Automatisierte Logik und Programmierung

Christoph Kreitz

Skript zur Vorlesung
Automatisierte Logik und Programmierung
Teil I: Formalisierung, Kalküle und interaktive Beweiser
Teil II: Automatisierte Programmentwicklung

Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstr. 10, 64283 Darmstadt

Inhaltsverzeichnis

1 Einführung **1**

1.1 Programmierung als logischer Prozeß . . . 2
1.2 Maschinell unterstützte Softwareentwicklung . . . 4
1.3 Formale Kalküle für Mathematik und Programmierung . . . 5
1.4 Strategien: Automatisierung von Entwurfsmethoden . . . 6
1.5 Aufbau der Veranstaltung . . . 7
1.6 Literaturhinweise . . . 8

2 Formalisierung von Logik, Berechenbarkeit und Typdisziplin **9**

2.1 Formale Kalküle . . . 9
2.1.1 Syntax formaler Sprachen . . . 10
2.1.2 Semantik formaler Sprachen . . . 11
2.1.3 Klassische und intuitionistische Mathematik . . . 12
2.1.4 Objekt- und Metasprache . . . 14
2.1.5 Definitorische Erweiterung . . . 14
2.1.6 Inferenzsysteme . . . 15
2.2 Prädikatenlogik . . . 22
2.2.1 Syntax . . . 23
2.2.2 Semantik . . . 26
2.2.3 Analytische Sequenzenbeweise . . . 29
2.2.4 Strukturelle Regeln . . . 31
2.2.5 Aussagenlogische Regeln . . . 32
2.2.6 Quantoren, Variablenbindung und Substitution . . . 36
2.2.7 Gleichheit . . . 41
2.2.8 Eigenschaften des Kalküls . . . 42
2.2.9 Beweismethodik . . . 44
2.3 Der λ-Kalkül . . . 45
2.3.1 Syntax . . . 48
2.3.2 Operationale Semantik: Auswertung von Termen . . . 49
2.3.3 Standard-Erweiterungen . . . 52

2.3.4 Ein Kalkül zum Schließen über Berechnungen . . . 58
2.3.5 Die Ausdruckskraft des λ-Kalküls . . . 59
2.3.6 Semantische Fragen . . . 61
2.3.7 Eigenschaften des λ-Kalküls . . . 62
2.4 Die einfache Typentheorie . . . 66
2.4.1 Syntax . . . 69
2.4.2 Semantik und Typzugehörigkeit . . . 70
2.4.3 Formales Schließen über Typzugehörigkeit . . . 72
2.4.4 Ein Typechecking Algorithmus . . . 75
2.4.5 Eigenschaften typisierbarer $\lambda - Terme$ . . . 78
2.4.6 Schließen über den Wert getypter $\lambda - Terme$ . . . 87
2.4.7 Die Curry-Howard Isomorphie . . . 87
2.4.8 Die Ausdruckskraft typisierbarer $\lambda - Terme$ . . . 88
2.5 Diskussion . . . 90
2.6 Ergänzende Literatur . . . 91

3 Die Intuitionistische Typentheorie 93
3.1 Anforderungen an einen universell verwendbaren Kalkül . . . 94
3.1.1 Typentheorie als konstruktive mathematische Grundlagentheorie . . . 94
3.1.2 Ausdruckskraft . . . 95
3.1.3 Eigenschaften von Berechnungsmechanismen und Beweiskalkül . . . 98
3.2 Systematik für einen einheitlichen Aufbau . . . 99
3.2.1 Syntax formaler Ausdrücke . . . 100
3.2.2 Semantik: Werte und Eigenschaften typentheoretischer Ausdrücke . . . 106
3.2.3 Inferenzsystem . . . 113
3.2.4 Grundprinzipien des systematischen Aufbaus . . . 126
3.3 Logik in der Typentheorie . . . 127
3.3.1 Die Curry-Howard Isomorphie . . . 127
3.3.2 Der leere Datentyp . . . 130
3.3.3 Konstruktive Logik . . . 134
3.3.4 Klassische Logik . . . 136
3.4 Programmierung in der Typentheorie . . . 137
3.4.1 Beweise als Programme . . . 138
3.4.2 Zahlen und Induktion . . . 140
3.4.3 Listen . . . 147
3.4.4 Teilmengen . . . 150
3.4.5 Quotienten . . . 155

3.4.6 Strings . . . 159
3.5 Rekursion in der Typentheorie . . . 159
3.5.1 Induktive Typen . . . 162
3.5.2 (Partiell) Rekursive Funktionen . . . 169
3.5.3 Unendliche Objekte . . . 173
3.6 Ergänzungen zugunsten der praktischen Anwendbarkeit . . . 174
3.7 Diskussion . . . 176
3.8 Ergänzende Literatur . . . 178

4 Automatisierung des formalen Schließens 181
4.1 Grundbausteine interaktiver Beweisentwicklungssysteme . . . 183
4.1.1 ML als formale Beschreibungssprache . . . 183
4.1.2 Implementierung der Objektsprache . . . 187
4.1.3 Bibliothekskonzepte . . . 190
4.1.4 Die Kommandoebene . . . 191
4.1.5 Der Text- und Termeditor . . . 192
4.1.6 Der Beweiseditor . . . 193
4.1.7 Definitionsmechanismus . . . 195
4.1.8 Der Programmevaluator . . . 197
4.1.9 Korrektheit der Implementierung . . . 198
4.2 Taktiken – programmierte Beweisführung . . . 198
4.2.1 Grundkonzepte des taktischen Beweisens . . . 199
4.2.2 Verfeinerungstaktiken . . . 200
4.2.3 Transformationstaktiken . . . 201
4.2.4 Programmierung von Taktiken . . . 203
4.2.5 Korrektheit taktisch geführter Beweise . . . 208
4.2.6 Erfahrungen im praktischen Umgang mit Taktiken . . . 208
4.3 Entscheidungsprozeduren – automatische Beweisführung . . . 209
4.3.1 Eine Entscheidungsprozedur für elementare Arithmetik . . . 210
4.3.2 Schließen über Gleichheit . . . 217
4.3.3 Andere Entscheidungsverfahren . . . 220
4.3.4 Grenzen der Anwendungsmöglichkeiten von Entscheidungsprozeduren . . . 221
4.4 Diskussion . . . 222
4.5 Ergänzende Literatur . . . 222

5 Automatisierte Softwareentwicklung 225
5.1 Verifizierte Implementierung mathematischer Theorien . . . 226
5.2 Grundkonzepte der Programmsynthese . . . 227

5.3 Programmentwicklung durch Beweisführung . . . 229
5.4 Programmentwicklung durch Transformationen . . . 230
5.4.1 Synthese durch Transformation logischer Formeln . . . 230
5.4.2 Die LOPS Strategie . . . 237
5.4.3 Integration in das allgemeine Fundament . . . 247
5.4.4 Synthese durch Tranformationen im Vergleich . . . 247
5.5 Programmentwicklung durch Algorithmenschemata . . . 248
5.5.1 Algorithmenschemata als Algebraische Theorien . . . 249
5.5.2 Globalsuch-Algorithmen . . . 256
5.5.3 Einbettung von Entwurfsstrategien in ein formales Fundament . . . 267
5.5.4 Divide & Conquer Algorithmen . . . 274
5.5.5 Lokalsuch-Algorithmen . . . 279
5.5.6 Vorteile algorithmischer Theorien für Syntheseverfahren . . . 280
5.6 Nachträgliche Optimierung von Algorithmen . . . 281
5.6.1 Simplifikation von Teilausdrücken . . . 281
5.6.2 Partielle Auswertung . . . 283
5.6.3 Endliche Differenzierung . . . 284
5.6.4 Fallanalyse . . . 285
5.6.5 Datentypverfeinerung . . . 286
5.6.6 Compilierung und sprachabhängige Optimierung . . . 287
5.7 Diskussion . . . 287

Literaturverzeichnis **289**

A Typentheorie: Syntax, Semantik, Inferenzregeln **I**
A.1 Parametertypen . . . I
A.2 Operatorentabelle . . . II
A.3 Redizes und Kontrakta . . . III
A.4 Urteile . . . IV
A.4.1 Typsemantik . . . IV
A.4.2 Elementsemantik – Universen . . . V
A.4.3 Elementsemantik . . . VI
A.5 Inferenzregeln . . . VII
A.5.1 Ganze Zahlen . . . VII
A.5.2 Void . . . VIII
A.5.3 Universen . . . VIII
A.5.4 Atom . . . IX
A.5.5 Funktionenraum . . . IX

A.5.6 Produktraum X
A.5.7 Disjunkte Vereinigung (Summe) XI
A.5.8 Gleichheit XII
A.5.9 Listen XII
A.5.10 Teilmengen XIII
A.5.11 Quotienten XIII
A.5.12 Induktive Typen XIV
A.5.13 Rekursive Funktionen XIV
A.5.14 Strukturelle Regeln und sonstige Regeln XV

B Konservative Erweiterungen der Typentheorie und ihre Gesetze XVII

B.1 Endliche Mengen XVII
B.2 Endliche Folgen XXIV
B.3 Endliche Abbildungen XXXI
B.4 Costas Arrays XXXI
B.5 Integer Segmente XXXI

Abbildungsverzeichnis

1.1 Berechnung der maximalen Segmentsumme: direkte Lösung 3
1.2 Berechnung der maximalen Segmentsumme: systematisch erzeugte Lösung 4

2.1 Frege–Hilbert–Kalkül für die Aussagenlogik 17
2.2 Kalkül des Natürlichen Schließens für die Aussagenlogik 19
2.3 Sequenzenkalkül für die Aussagenlogik 21
2.4 Sequenzenbeweis für die Kommutativität der Konjunktion 33
2.5 Sequenzenbeweis für das Distributivgesetz zwischen Allquantor und Konjunktion 40
2.6 Der analytische Sequenzenkalkül für die Prädikatenlogik erster Stufe 43
2.7 Sequenzenkalkül für die Gleichheit von λ-Termen 58
2.8 Sequenzenkalkül für die einfache Typentheorie 73
2.9 Sequenzenbeweis für eine Typisierung von $(\lambda f.\lambda x.\, fx)(\lambda z.z)$ 74
2.10 Typechecking Algorithmus für die einfache Typentheorie 76

3.1 Parametertypen und zugehörige Elemente 104
3.2 Operatorentabelle für Funktionenraum, Produktraum und Summe 106
3.3 Die Auswertungsprozedur der Typentheorie 108
3.4 Vollständige Operatorentabelle für Funktionenraum, Produktraum und Summe 108
3.5 Redex–Kontrakta Tabelle für Funktionenraum, Produktraum und Summe 109
3.6 Semantik der Urteile für Funktionenraum, Produktraum und Summe 112
3.7 Operatorentabelle für Universen und Gleichheit 115
3.8 Semantik der Urteile für Universen und Gleichheit 116
3.9 Regeln für Funktionenraum 122
3.10 Regeln für Produktraum 123
3.11 Regeln für disjunkte Vereinigung (Summe) 124
3.12 Regeln für Universen und Gleichheit 125
3.13 Strukturelle Regeln 126
3.14 Syntax, Semantik und Inferenzregeln des Typs void 132
3.15 Typisierungsbeweis für $\lambda T.\lambda z.\,(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$ 134
3.16 Interpretationen der typentheoretischen Urteile 138
3.17 Syntax und Semantik des Typs $\mathbb{Z}$ 143

3.18 Inferenzregeln des Typs $\mathbb{Z}$ (1) . . . 145
3.19 Inferenzregeln des Typs $\mathbb{Z}$ (2) . . . 146
3.20 Syntax, Semantik und Inferenzregeln des Listentyps . . . 148
3.21 Formaler Induktionsbeweis für die Existenz der maximalen Segmentsumme . . . 149
3.22 Syntax, Semantik und Inferenzregeln des Teilmengentyps . . . 152
3.23 Syntax und Semantik des Quotiententyps . . . 156
3.24 Inferenzregeln des Quotiententyps . . . 157
3.25 Syntax, Semantik und Inferenzregeln des Typs Atom . . . 160
3.26 Syntax, Semantik und Inferenzregeln induktiver Typen . . . 164
3.27 Syntax, Semantik und Inferenzregeln partiell rekursiver Funktionen . . . 172
3.28 Zusätzliche Inferenzregeln von NuPRL . . . 175

4.1 Darstellung eines Beweisknotens im Editorfenster . . . 194
4.2 Von der Taktik `simple_prover` generierter Beweis . . . 203
4.3 Wichtige vordefinierte Tacticals . . . 204
4.4 Implementierung der Taktik `simple_prover` . . . 207
4.5 Die Theorie $\mathcal{A}$ der elementaren Arithmetik . . . 212
4.6 Varianten der Monotoniegesetze von $\mathcal{A}$ . . . 213
4.7 Die Entscheidungsprozedur `arith` . . . 216
4.8 Der MERGE Algorithmus . . . 219
4.9 Entscheidungsalgorithmus zum Schließen über Gleichheiten . . . 220

5.1 Erweitertes Vokabular . . . 226
5.2 Synthese durch Erweiterung algebraischer Theorien . . . 249
5.3 Synthese mit Algorithmenschemata: Generelle Methode . . . 253
5.4 Hierarchie algorithmischer Theorien . . . 256

Es gibt selbstgefällige Programmierer, die die Vorstellung kultivieren, Informatik könne auf Mathematik oder Logik verzichten. Ebensogut kann ein Esel auf Beine verzichten. Er hat doch Zähne, sich vorwärtszuziehen.
(F.X. Reid)

Kapitel 1

Einführung

Seit mehr als 25 Jahren spricht man von einer Krise bei der Produktion kommerzieller Software. Auslöser dieser Krise ist genau diejenige Eigenschaft, welche den Einsatz von Software so attraktiv macht, nämlich die Vielseitigkeit und das komplexe Verhalten, das man erzeugen kann. Bis heute hat sich die Krise eher verschlimmert als verbessert, denn die Entwicklung von Methoden zur systematischen Softwareproduktion kann mit der stetig wachsenden Komplexität von Software, die auf dem Markt verlangt wird, nicht Schritt halten. Zwei Probleme treten dabei besonders hervor.

Das erste Problem sind die *Kosten von Software*, die in einem krassen Mißverhältnis zu dem Verfall der Hardwarepreise stehen. Im Gegensatz zu diesen entstehen Softwarekosten fast ausschließlich beim Entwurf — einer Phase, die Kreativität, Erfahrung und Disziplin verlangt. Zwar ist in allen Branchen die Entwurfsphase mit hohen Kosten verbunden. Bei der Softwareproduktion jedoch ist die Situation besonders schlecht, da die meisten Programme immer noch aus elementaren Konstrukten von Programmiersprachen anstatt aus größeren Modulen aufgebaut werden. Dies ist zu einem gewissen Teil eine Angewohnheit, die durch den wachsenden Einfluß objektorienterter Programmiersprachen abgebaut werden könnte. Prinzipiell aber liegt das Problem darin, daß selbst kleine Module bereits solch komplizierte Handlungsabläufe beschreiben müssen, daß hierzu bestehende Programmstücke nicht ohne Modifikationen übernommen werden können. Darüber hinaus werden die Kosten aber auch dadurch erhöht, daß der Erstentwurf eines Softwareprodukts nur selten die Absichten des Entwicklers erfüllt. Dies liegt daran, daß es – ganz im Gegensatz zur Hardwareentwicklung – bisher keinerlei Werkzeuge gibt, mit denen man ein bestimmtes Verhalten von Programmen erzwingen kann. Die notwendigen Tests und Korrekturen bis zur Erstellung des endgültigen Produkts dauern oft lange und machen dieses entsprechend teuer.

Aber auch nach der Auslieferung des Produkts entstehen weitere Kosten. Zum einen machen sich weitere Fehler in der Implementierung bemerkbar, da Tests niemals vollständig sein können. Zum anderen aber stellt sich nur allzu oft heraus, daß die gelieferte Software den wirklichen Wünschen der Kunden nicht vollständig entspricht, da diese Wünsche ursprünglich nicht genau genug formuliert waren. Aus diesem Grunde müssen nachträglich noch vielfältige Ergänzungen und Verbesserungen vorgenommen werden. Diese Modifikationen verlangen oft Änderungen, die mehr mit der abstrakten Entwurfsebene zu tun haben als mit dem unmittelbar vorliegenden Programmcode. Der ursprüngliche Entwicklungs*prozeß* ist jedoch nur selten ausführlich genug dokumentiert. Daher ist die Umsetzung dieser abstrakten Änderungen auf der Implementierungsebene fast genauso schwierig wie der Entwurf des ersten Prototyps. In der Tat sind *Wartung* von Software und eventuelle spätere *Erweiterungen* oft erheblich teurer als das Produkt selbst. All dies bezahlt im Endeffekt der Kunde – durch hohe Softwarepreise, Wartungsverträge oder die Notwendigkeit, "Upgrades" zu kaufen.

Das zweite und vielleicht bedeutendere Problem betrifft die *Zuverlässigkeit* von Software. Immer größer wird die Tendenz, die Steuerung komplizierter Prozesse in technischen Systemen einem Computerprogramm zu überlassen, da Menschen sie nicht mehr handhaben können oder wollen. Die Bandbreite der Anwendungen reicht von Waschmaschinenprogrammen und Autoelektronik (ABS, elektronische Motorsteuerung) bis hin zu Flugzeugsteuerung (Autopilot, "fly-by-wire"), Flugsicherung, U-Bahn-Netzen (automatischer Betrieb ohne Fahrer) und Sicherheitssystemen in Atomkraftwerken. Während man die Zuverlässigkeit der hierbei verwand-

ten Motoren, der mechanischen Steuerungen, der elektronischen Bauteile und der Computerhardware durch umfangreiche Tests sicherstellen kann, läßt die Unstetigkeit digitaler Systeme dies für die eingesetzte Software nicht zu. Bisher gibt es keinerlei praktikable Hilfsmittel, die Korrektheit von Software auf anderen Wegen zu garantieren. So verläßt man sich einfach darauf, daß die Produkte hinreichend gut sind, wenn während der Testphase keine der gängigen Fehler auftaucht.

Auch wenn es bisher kaum Unfälle gegeben hat, die eindeutig auf Softwarefehler zurückzuführen sind, glauben nur noch wenige Leute, daß im Angesicht der gegenwärtige Produktionsweise Software-verursachte Katastrophen gänzlich ausgeschlossen werden können. Spektakuläre Flugzeugunfälle zu Anfang der 90er Jahre wie der Absturz der Lauda Air Boeing 767 (Umkehrschub in der Luft) und der Air Inter French A 320 (unerklärlich zu tief geflogen) haben zu langen und gefährlichen Spekulationen über die tatsächlichen Ursachen geführt und Zweifel genährt, ob Software in sicherheitsrelevanten Bereichen überhaupt eingesetzt werden sollte.

Seit dem Aufkommen der Softwarekrise gibt es Bemühungen, zuverlässigere Methoden der Softwareentwicklung zu entwerfen, um den Einsatz von Software in diesen ökonomisch so wichtigen Bereichen zu ermöglichen. Methodiker beschrieben Programmierung als Kunst, Disziplin, Handwerk und Wissenschaft, um die verschiedenen Aspekte des Programmierprozesses hervorzuheben, zu systematisieren und somit zur Entwicklung besserer Software beizutragen.[1] Programmierung, so wird immer wieder hervorgehoben, ist weit mehr als die Beherrschung aller Feinheiten einer konkreten Programmiersprache, da beim eigentlichen Entwurf ganz andere Fähigkeiten eine Rolle spielen. Es kommt darauf an, kreativ Ideen zu entwerfen, sie auszuarbeiten und sich darüber klar zu werden, *warum* diese Idee eine Lösung des Problems darstellt. Neben Programmiererfahrung und Disziplin beim Präzisieren der Idee verlangt dies vor allem ein logisches Durchdenken und Analysieren von Problemen und Lösungen.

1.1 Programmierung als logischer Prozeß

Die Fähigkeit, zu *beweisen*, daß eine Idee tatsächlich auch funktioniert, spielt dabei im Hinblick auf die Zuverlässigkeit der erzeugten Programme ein fundamentale Rolle. Hierin aber liegt oft auch das größte Problem. Viele Programmierer haben keine klare Vorstellung davon, was Korrektheit wirklich bedeutet und wie man beweisen kann, daß ein Programm tatsächlich korrekt ist. Zudem hat das Wort "Beweis" für die meisten einen unangenehmen Beigeschmack, der viel Mathematik assoziiert. Sie sehen darin eher eine Behinderung ihrer Arbeit, die keinerlei Nutzen mit sich bringt. Wozu ein Programm beweisen, wenn es doch "funktioniert"?

Dies aber ist kurzsichtig gedacht. Zwar ist es richtig, daß es dem Entwickler wenig bringt, einen Beweis *nachträglich* zu führen. Es ist einfach zu schwer und oft auch nicht mehr praktisch durchführbar, denn bei der Beweisführung müssen alle Ideen, die zur Entwicklung des Programms beigetragen haben, rekonstruiert und mit dem endgültigen Programm in Beziehung gesetzt werden. Gewöhnt man sich aber an, bei der Implementierung *gleichzeitig* auch schon an einen möglichen Korrektheitsbeweis zu denken, so wird dies die Qualität des erstellten Programms erheblich steigern. Auch ist es effizienter, die Einsichten, die sich aus den Beweisideen ergeben, in die Implementierung einfließen zu lassen. Dies wollen wir an einem Beispiel illustrieren, auf das wir öfter noch zurückkommen werden.

Beispiel 1.1.1 (Maximale Segmentsumme)

Betrachten wir einmal das folgende einfache, aber doch realistische Programmierproblem

> *Zu einer gegebenen Folge $a_1, a_2, \ldots, a_n$ von n ganzen Zahlen soll die Summe $m = \sum_{i=p}^{q} a_i$ einer zusammenhängenden Teilfolge bestimmt werden, die maximal ist im Bezug auf alle möglichen Summen zusammenhängender Teilfolgen $a_j, a_{j+1} \ldots, a_k$.*

[1] Die Lehrbücher von Knuth [Knuth, 1968, Knuth, 1972, Knuth, 1975], Dijkstra [Dijkstra, 1976], Reynolds [Reynolds, 1981] und Gries [Gries, 1981] spielen in den Kursen über Programmierung heute noch eine wichtige Rolle. Lesenswert sind auch Abhandlungen von Wirth [Wirth, 1971] sowie Dahl, Dijkstra und Hoare [Dahl *et.al.*, 1972].

```
maxseg:INTEGER
  is
  local p, q, i, sum :INTEGER
  do
    from p := lower              -- variiere untere Grenze p
         Result := item(lower);
      until p >= upper
      loop
        from q := p              -- variiere obere Grenze q
          until q > upper
          loop
            from i := p ;        -- Berechne ∑_{i=p}^{q} a_i
                 sum := item(i) -- Initialwert der Summe
              until i = q
              loop
                  i := i+1;
                  sum := sum+item(i)
            end -- sum = ∑_{i=p}^{q} a_i
            if sum > Result
              then Result := sum
            end
            q := q+1
        end;
        p := p+1
    end
end
```

Abbildung 1.1: Berechnung der maximalen Segmentsumme: direkte Lösung

Derartige zusammenhängende Teilfolgen heißen *Segmente* und das Problem ist deshalb als das Problem der *maximalen Segmentsumme* bekanntgeworden. Für die Folge `-3, 2,-5, 3,-1, 2` ist zum Beispiel die maximale Segmentsumme die Zahl `4` und wird erreicht durch das Segment `3,-1, 2`.

Die vielleicht naheliegenste Lösung dieses Problems wäre, einfach alle möglichen Segmente und ihre Summen zu bestimmen und dann die größte Summe auszuwählen. Diese Lösung – in Abbildung 1.1 durch ein Eiffel-Programm beschrieben – ist jedoch weder elegant noch effizient, da die Anzahl der Rechenschritte für Folgen der Länge n in der Größenordnung von n^3 liegt – also 1 000 Schritte für Folgen der Länge 10 und schon 1 000 000 für Folgen der Länge 100. Es lohnt sich daher, das Problem systematisch anzugehen. Dazu formulieren wir es zunächst einmal in mathematischer Notation.

Zu einer Folge a der Länge n soll $\underline{M_n} = \texttt{max}(\{\sum_{i=p}^{q} a_i \mid 1 \leq \texttt{p} \leq \texttt{q} \leq \texttt{n}\})$ *berechnet werden.*

Da eine Lösung ohne Schleifen nicht möglich sein wird, bietet es sich an, nach einer induktiven Lösung zu suchen, bei der die Länge der Folge (wie in der direkten Lösung) natürlich eine Rolle spielen wird. Für einelementige Folgen ist nämlich die Lösung trivial – die maximale Segmentsumme ist der Wert des einzigen Elementes – und es besteht eine gewisse Hoffnung, daß wir das Problem einheitlich lösen können, wenn wir die (betrachtete) Folge um ein weiteres Element ergänzen.[2]

Nehmen wir also an, wir hätten für eine Folge der Länge n die maximale Segmentsumme M_n bereits bestimmt. Wenn wir nun ein neues Element a_{n+1} hinzufügen, dann ist die neue maximale Segmentsumme M_{n+1} entweder die Summe eines Segmentes, welches a_{n+1} enthält oder die Summe eines Segmentes, welches a_{n+1} nicht enthält.

Im ersten Fall müssen wir wissen, was die maximale Summe eines Segments ist, welches das letzte Element a_{n+1} enthält. Wir nennen diese Summe L_{n+1} und untersuchen sie später. Im zweiten Fall ist die Lösung einfach, da das letzte Element keine Rolle spielt – sie ist identisch mit M_n. Insgesamt wissen wir also, daß M_{n+1} das Maximum von L_{n+1} und M_n ist.

[2]Wir haben daher den Maximalwert M_n mit einem Index n versehen, der auf die Abhängigkeit von der Länge hinweist.

```
maxseg:INTEGER
  is
  local n, L_n :INTEGER
  do
    from n := lower;
         Result := item(n);
         L_n := item(n)
      until n >= upper
      invariant n<=upper                  -- ∧ Result=M_n ∧ L_n=L_n
      variant  upper - n
      loop
         if L_n > 0
            then L_n := L_n + item(n+1)
            else L_n := item(n+1)
         end;                          --  L_n = L_{n+1}
         if L_n > Result
            then  Result := L_n
         end;                          -- Result = M_{n+1}
         n := n+1
      end
end
```

Abbildung 1.2: Berechnung der maximalen Segmentsumme: systematisch erzeugte Lösung

Nun müssen wir noch L_{n+1} bestimmen, also die maximale Summe einer Teilfolge $a_i, a_{i+1} \ldots, a_{n+1}$. Da wir induktiv vorgehen, können wir davon ausgehen, daß L_n bereits aus dem vorhergehenden Schritt bekannt ist. Ist L_n negativ, dann ist die maximale Summe einer Folge, welche a_{n+1} enthält, die Summe der einelementigen Folge a_{n+1} (jede längere Folge hätte nur eine kleinere Summe). Andernfalls können wir "gewinnen", wenn wir die Folge a_{n+1} um das Segment ergänzen, dessen Summe L_n war, und erhalten L_{n+1} als Summe von L_n und a_{n+1}. Da L_1 offensichtlich a_1 ist, haben wir die Induktion auch verankert.

Dieses Argument sagt uns genau, wie wir eine Lösung berechnen und als korrekt nachweisen können. Der Algorithmus, der sich hieraus ergibt, ist in Abbildung 1.2 wiederum durch ein Eiffel-Programm beschrieben. Er läßt sich jetzt leicht verifizieren (wir müssen nur obiges Argument präzisieren) und ist darüber hinaus auch viel effizienter als der vorherige. Die Anzahl der Rechenschritte für Folgen der Länge n liegt nun in der Größenordnung von $3n$ – also nur etwa 300 für Folgen der Länge 100.

Dieses Beispiel zeigt, daß es sehr sinnvoll ist, in Programm und seinen Korrektheitsbeweis gleichzeitig zu entwickeln, wobei der *Beweis* die Vorgehensweise bestimmen sollte. Dabei muß ein Beweis im Prinzip weder mathematisch noch formal sein, sondern im wesentlichen nur *ein Argument, das den Leser von der Wahrheit eines Sachverhalts überzeugt*. Die Tatsache, daß Programmierer einen Großteil ihrer Zeit damit verbringen, Fehler aus ihren Programmen zu eliminieren – *obwohl* sie von der Richtigkeit ihrer Ideen überzeugt waren – zeigt jedoch, daß die derzeit vorherrschende 'Methode', sich von der Korrektheit eines Programms zu überzeugen, völlig unzureichend ist.

Der Grund hierfür wird relativ schnell klar, wenn man sich anschaut, welche Fähigkeiten bei der Entwicklung zuverlässiger Programme eigentlich gefragt sind. Das Beispiel zeigt, daß dies – neben Kreativität – vor allem die Fähigkeit ist, logische Schlußfolgerungen zu ziehen und dabei Wissen über Programme, Anwendungsbereiche und Programmiermethoden zu verwenden. Angesichts der Komplexität der zu erzeugenden Software ist diese Aufgabe für Menschen kaum zu bewältigen, denn sie neigen dazu, Flüchtigkeitsfehler in scheinbar unbedeutenden Details zu machen, die sich nachher fatal auswirken.

1.2 Maschinell unterstützte Softwareentwicklung

Die Aufgabe, komplexe Zusammenhänge fehlerfrei bis ins letzte Detail auszuarbeiten, stellt für menschliche Programmierer ein fast unlösbares Problem dar. Da Präzision bei der Behandlung komplexer Details aber

genau eine der wesentlichen Stärken von Computern ist, liegt der Gedanke nahe, den Programmierprozeß durch Computer unterstützen zu lassen und auf diese Art sicherzustellen, daß ein Programm die gewünschten Eigenschaften tatsächlich auch besitzt.

Ein erster Schritt im Hinblick auf eine rechnergestützte Entwicklung von Programmen ist das sogenannte *Computer-aided software engineering* (CASE). In speziellen Anwendugen können *Code-Generatoren* bereits benutzt werden, um Routinesoftware zu erzeugen. Werkzeuge für *Analyse und Entwurf* helfen, die Konsistenz modularer Systeme sicherzustellen. *Projekt Management Systeme* können dafür genutzt werden, größere Projekte unter Berücksichtigung aller Zusammenhänge zu planen und ihren Verlauf zu überwachen. Für die Praxis sind CASE Tools etwas sehr Nützliches, zumal die Technologie mittlerweile schon recht weit ausgereift ist. Dennoch sind sie keine Antwort auf das eigentliche Problem. Sie unterstützen nur spezielle Problemstellungen, nicht aber eine *logische* Verarbeitung von Wissen auf dem Weg von der Problemstellung zur Lösung. Es besteht somit keine Möglichkeit, zu dokumentieren, welche Ideen und Techniken bei der Entwicklung des Programms eine Schlüsselrolle gespielt haben. Derartiges Wissen aber ist wichtig, um die Korrektheit eines Softwaresystems zu überprüfen, Änderungen und Erweiterungen durchzuführen und einmal gewonnene Erkenntnisse bei der Entwicklung neuer Software wiederzuverwenden. Man benötigt weit mehr als die CASE Technologie, um eine wirkliche Unterstützung bei der Produktion zuverlässiger Software bereitzustellen.

Im Prinzip müßte man eine Automatisierung des gesamten Programmierprozesses anstreben und Werkzeuge für *wissensbasiertes Software Engineering*[3] entwickeln. Neben der Erzeugung *formaler* Spezifikationen aus informalen Anforderungen, was wir hier nicht weiter betrachten wollen,[4] bedeutet dies eine rechnergestützte *Synthese* von Programmen aus formalen Spezifikationen. Da Computer aber nur mit formalen Objekten operieren können, müssen hierzu im wesentlichen zwei Teilaufgaben bewältigt werden. Zum einen ist es nötig, jegliche Form von Wissen, das bei der Programmentwicklung eine Rolle spielt, zu *formalisieren* und *Kalküle* bereitzustellen, welche erlauben, das logische Schließen durch eine Manipulation formaler Objekte zu simulieren. Zum zweiten ist es sinnvoll, *Strategien* zu entwickeln, welche das formalisierte Wissen verwenden, um Schlüsse zum Teil automatisch auszuführen und einen Programmierer bei der Bearbeitung von formalen Detailfragen und Routineproblemen zu entlasten. Dabei ist es durchaus wünschenswert, einfache Programme vollautomatisch aus ihren Spezifikationen erzeugen zu können. Mit beiden Teilaufgaben werden wir uns im folgenden auseinandersetzen.

1.3 Formale Kalküle für Mathematik und Programmierung

Für eine rechnergestützte Synthese von Programmen ist es notwendig, ein breites Spektrum von Wissen aus Mathematik und Programmierung in formalisierter Form bereitzustellen. Ein Programmsynthesesystem sollte daher auf einem universell verwendbaren logischen Kalkül aufbauen, in dem man – zumindest im Prinzip – jegliche mathematische Aussage, einschließlich solcher über das Verhalten von Programmen, formal ausdrücken und ihren Wahrheitsgehalt untersuchen kann.

Die Entwicklung eines derartigen Kalküls, der es ermöglicht, natürlichsprachliche Aussagen in eine formale Sprache (*lingua rationalis*) zu übertragen und ihre Korrekheit mittels eines Berechnungsschemas (*calculus rationaticor*) zu überprüfen ist ein uralter Traum der Mathematiker, der mindestens auf die Zeit von G. W. Leibnitz (1646–1716) zurückgeht. Die Hoffnung war, auf diese Art Streitigkeiten unter Wissenschaftlern

[3] Dieser Begriff wurde erstmals geprägt in [Balzer *et.al.*, 1983].

[4] Es sei an dieser Stelle angemerkt, daß die Erstellung formaler Spezifikationen keineswegs trivial ist und daß eine Menge wichtiger Arbeiten auf diesem Gebiet geleistet wurden. Detaillierte Informationen hierzu kann man z.B. in [Hayes, 1987, Jones & Shaw, 1990, Kelly & Nonnenmann, 1991, Johnson & Feather, 1991] und [Lowry & Duran, 1989, Abschnitt B und D] finden. Wenn wir Softwareentwicklung als die Aufgabe betrachten, eine Brücke zwischen den Anforderungen und dem Programmcode zu bauen, so kann man die formale Spezifikation als einen Stützpfeiler in der Mitte ansehen. Indem wir uns die Konstruktion der zweiten Hälfte zum Ziel setzen, welche sich leichter formalisieren und durch Computer unterstützen läßt, unterstützen wir den Bau der ersten, da Zuverlässigkeitsanalyse und Wartung nun auf der Ebene der Spezifikation anstelle des Programmcodes durchgeführt werden können.

dadurch beseitigen zu können, daß man einfach durch Ausrechnen entscheidet, welche Aussagen wahr sind und welche nicht. In der Denkweise der heutigen Zeit bedeutet dies, daß man versuchte Maschinen zu bauen, die *entscheiden*, ob eine mathematische Aussage wahr ist oder nicht. Seit den dreißiger Jahren dieses Jahrhunderts weiß man jedoch, daß dies im Allgemeinfall nicht möglich ist. Probleme wie die Terminierung von Algorithmen oder das Entscheidungsproblem selbst sind unentscheidbar. Beweisverfahren, die mathematische Probleme entscheiden, sind also nur für spezielle Teiltheorien der Mathematik möglich. Will man dagegen *jegliche* Art mathematischer Aussagen formal untersuchen, so muß man sich darauf beschränken, Kalküle zu entwickeln, in denen formale Beweise geführt und durch Rechner *überprüft* werden können.[5]

Die axiomatische Mengentheorie, die Grundlage der modernen Mathematik, ist für diese Zwecke leider ungeeignet, da sie hochgradig unkonstruktiv ist, also den Begriff des *Algorithmus* kaum erfassen kann. Aus diesem Grunde wurden seit den siebziger Jahren *konstruktive Kalküle* entwickelt, die in Hinblick auf Universalität und Anwendungsbereich der klassischen Mengentheorie gleichkommen. Die wichtigsten Vertreter sind Martin-Löf's *intuitionistische prädikative Mengentheorie* (siehe [Martin-Löf, 1982, Martin-Löf, 1984]), der *Kalkül der Konstruktionen* von Coquand und Huet [Coquand & Huet, 1985, Coquand & Huet, 1988], Platek und Scott's *Logik berechenbarer Funktionen* und die *Lineare Logik* von Girard [Girard, 1987]. Im Gegensatz zur Mengentheorie, die im wesentlichen nur den Begriff der Menge und ihrer Elemente kennt, enthalten diese Theorien bereits formale Gegenstücke für die wichtigsten Standardkonzepte aus Mathematik und Programmierung – einschließlich eines Klassifizierungskonzepts wie Datentypen bzw. Sorten – und ermöglichen eine direkte Formalisierung von mathematischen Aussagen und Aussagen über Programme *und* ihre Eigenschaften.

Parallel zur Entwicklung von Kalkülen hat man sich bemüht, Rechnerunterstützung für das Erstellen und Überprüfen formaler Beweise zu schaffen. Beginnend mit dem *AUTOMATH* Projekt in Eindhoven (siehe [Bruijn, 1980]) wurden generische *proof-checker* implementiert, mit denen man der Automatisierung der Mathematik wieder ein Stück näher kam. In späteren Systemen wie *Edinburgh LCF* [Gordon *et.al.*, 1979], *PP*λ [Paulson, 1987]), *NuPRL* [Constable *et.al.*, 1986] und *ISABELLE* [Paulson, 1990] wurde dieses Konzept schrittweise zu *Beweiseditoren* ausgebaut, die man zur interaktiven Entwicklung von Beweisen verwenden kann, wobei es möglich ist, *Taktiken* (d.h. Suchprogramme) bei der Suche nach Beweisen zuhilfe zu nehmen.

Beweiseditoren für ausdruckstarke Kalküle, die es erlauben, jegliche Form logischen Schließens über mathematische Aussagen und Eigenschaften von Programmen auf dem Computer zu simulieren, gekoppelt mit der Möglichkeit, das logische Schließen durch programmierte Taktiken zum Teil zu automatisieren, scheinen für die rechnergestützte Entwicklung zuverlässiger Software das ideale Handwerkzeug zu sein.

Im Laufe dieser Veranstaltung werden wir uns im wesentlichen mit der *intuitionistischen Typentheorie* des NuPRL Systems befassen, die eine Weiterentwicklung der Martin-Löf'schen Theorie ist.[6] Im Verhältnis zu anderen ist die Theorie sehr reichhaltig und vor allem das zugehörige System ist schon sehr weit entwickelt, auch wenn es immer noch viele Möglichkeiten zu weiteren Verbesserungen gibt.

1.4 Strategien: Automatisierung von Entwurfsmethoden

Für die Implementierung von Strategien, die logische Schlüsse zum Teil selbständig ausführen und Routineprobleme eventuell vollautomatisch lösen können, spielt das eben erwähnte Konzept der *Taktiken* eine wesentliche Rolle. Taktiken sind ("Meta"-)Programme, welche die Anwendung elementarer Schlußregeln in einem ansonsten voll interaktiven Beweiseditor steuern und einen Anwender von der Bearbeitung trivialer, aber aufwendiger Detailprobleme entlasten. Diese Vorgehensweise sichert die Korrektheit aller ausgeführten logischen Schritte

[5] Durch den Gödelschen Unvollständigkeitssatz wurde nachgewiesen, daß es auch hier Grenzen gibt: zu jedem formalen System, das eine Formalisierung der *gesamten* Arithmetik erlaubt, kann man Aussagen formulieren, die in diesem System weder bewiesen noch widerlegt werden können. Diese Aussagen haben zum Glück wenig praktische Auswirkungen.

[6] Der Name resultiert daher, daß bei dieser Theorie die Datentypen (bzw. die Klassifizierung eines mathematischen Objektes) eine fundamentale Rolle spielen und bewußt eine Anlehnung an die elementaren Datenstrukturen von Programmiersprachen vorgenommen wurde.

und gibt dennoch eine große Flexibilität zum Experimentieren mit Beweis- und Programmentwicklungsstrategien.[7]

Die meisten Beweis- und Programmsyntheseverfahren wurden bisher allerdings unabhängig von den im letzten Abschnitt genannten Kalkülen entwickelt und implementiert. *Theorembeweiser*, die in der Lage sind, mathematische Beweise eigenständig zu finden, sind in ihrem Anwendungsbereich eingeschränkt auf Probleme, die sich gut in der Prädikatenlogik – eventuell ergänzt durch Gleichheit und einfache Induktion (siehe Abschnitt 2.2) – formulieren lassen. Viele *Programmsynthesestrategien* wurden zunächst als eher informale Methoden auf dem Papier entwickelt und dann als eigenständige Programme zur Manipulation von Formeln implementiert. Diese Vorgehensweise ist aus praktischen Gesichtspunkten zunächst einfacher, effizienter und erfolgversprechender. Das KIDS System [Smith & Lowry, 1990, Smith, 1991a, Smith & Parra, 1993] ist sogar schon nahe daran, für Routineprogrammierung verwendbar zu sein. Dennoch bleibt bei einer von formalen Kalkülen losgelösten Implementierung immer die Gefahr, daß Strategien, die auf dem Papier nachweisbar gute Eigenschaften besitzen, bei ihrer Implementierung Fehler enthalten und somit nicht die zuverlässige Software generieren, die man von ihnen erwartet. Es besteht daher die Notwendigkeit, diese Strategien durch eine Kombination von Taktiken und einer Formalisierung der "Theorie der Programmierung" in das Konzept der formalen Kalküle zu integrieren. Auf welche Art dies sicher und effizient geschehen kann, wird derzeit noch erforscht.

1.5 Aufbau der Veranstaltung

Die *Automatisierte Logik und Programmierung* ist als zweiteilige Veranstaltung ausgelegt. Der Schwerpunkt des ersten Teils wird das theoretische Fundament sein, also formale Kalküle, ihre Eigenschaften und der Bau von interaktiven Beweissystemen.

Im Kapitel 2 werden wir formale Kalküle als solche vorstellen und zur Formalisierung von Logik, Berechnung und Datentypen zunächst drei separate Kalküle vorstellen, die unabhängig voneinander entstanden sind. Wir werden dabei einige grundlegende Eigenschaften von Kalkülen besprechen, die notwendig sind, um die Sicherheit zu garantieren, die wir von formalen Systemen erwarten.

Die Typentheorie des NuPRL Systems, die wir im Kapitel 3 ausführlich diskutieren, bringt dann alles in einem einheitlichen Rahmen zusammen und ergänzt es um Formalisierungen der meisten Grundkonzepte, die in Programmiersprachen eine Rolle spielen. Da diese Theorie verhältnismäßig umfangreich ist, werden wir auch über eine Systematik beim Aufbau formaler Konzepte sprechen und über das sensible Gleichgewicht zwischen der Eleganz und der "Kontrollierbarkeit" formaler Theorien.

Im Kapitel 4 werden wir dann besprechen, wie auf der Basis eines formalen Kalküls zuverlässige interaktive Systeme gebaut werden können, mit denen ein Anwender beweisbar korrekte Schlüsse über Programme und ihre Eigenschaften ziehen kann.

In diesem ersten Teil werden theoretische Aspekte, Formalismen und Logik eine große Rolle spielen. Sie sollen lernen, Stärken und Schwächen sowie Grenzen und Möglichkeiten von Kalkülen einzuschätzen, Probleme zu formalisieren, formale Beweise zu führen und Beweiseditoren zu benutzen, auszubauen und im Hinblick auf Automatisierung zu ergänzen. In einem zweiten Teil wird es dann darum gehen, wie man auf der Grundlage dieses Fundaments mathematisches Wissen und eine formale Theorie der Programmierung "implementieren" kann und mit welchen konkreten Strategien sich eine Entwicklung korrekter Programme aus Spezifikationen unterstützen läßt.

[7] Dieses Konzept wurde erstmalig für *Edinburgh LCF* [Gordon *et.al.*, 1979] entwickelt und später in vielen Systemen übernommen und erfolgreich angewandt wie z.B. in *Cambridge LCF* [Paulson, 1987], *NuPRL* [Constable *et.al.*, 1986], *λ-PROLOG* [Felty & Miller, 1988, Felty & Miller, 1990], *OYSTER* [Bundy, 1989, Bundy *et.al.*, 1990], *ISABELLE* [Paulson, 1989, Paulson, 1990] *KIV* [Heisel *et.al.*, 1988, Heisel *et.al.*, 1990, Heisel *et.al.*, 1991], und *DEVA* [Lafontaine, 1990, Weber, 1990].

1.6 Literaturhinweise

Zur Typentheorie und ihren Wurzeln gibt es eine Fülle von Literatur, aber so gut wie keine Lehrbücher. Wir stützen uns im wesentlichen auf zwei zusammenfassende Darstellungen:

- Per Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Napoli, 1984.
- R.L. Constable et.al, *Implementing Mathematics with the NuPRL Proof Development System*, Prentice Hall, Englewood Cliffs, 1986.

In beiden Büchern werden die Grundgedanken und viele Details der Typentheorie und des NuPRL Systems ausführlich erklärt. In den letzten 10 Jahren haben sich aber sowohl die Theorie als auch das System erheblich weiterentwickelt. Detailaspekte sind in vielen verschiedenen Schriften dokumentiert [Constable, 1984, Constable & Mendler, 1985, Mendler, 1987b, Constable & Smith, 1987, Allen, 1987a, Constable & Smith, 1988, Constable, 1989, Constable & Howe, 1990b]. Der aktuelle Stand ist derzeit nur in diesem Buch und den folgenden Manuals zu finden.

- The Nuprl Proof Development System, Version 4.1: Introductory Tutorial
- The Nuprl Proof Development System, Version 4.1: Reference Manual and User's Guide
- NuPRL's Metalanguage ML: Reference Manual and User's Guide

Es mag sich herausstellen, daß die Manuals noch Fehler aufweisen oder für Sie unverständlich sind. Für entsprechende konkrete Hinweise wäre ich sehr dankbar.

Für Interessierte lohnt es, die Entwicklung der Typentheorie, der konstruktiven Mathematik und der Logik weiter zurückzuverfolgen. Die Typentheorie von Martin-Löf [Martin-Löf, 1970, Martin-Löf, 1982] geht in ihren Zielen zurück auf die konstruktive Mathematik von L.E.J. Brouwer (siehe hierzu [Brouwer, 1908b, Brouwer, 1908a, Brouwer, 1924b, Brouwer, 1924a, Brouwer, 1925, Brouwer, 1926, Brouwer, 1927] – zu finden in [Brouwer, 1975]), A. Heyting [Heyting, 1934, Heyting, 1971], E. Bishop [Bishop, 1967, Bridges, 1979] und A.S. Troelsta [Troelsta, 1969, Troelsta, 1977]. Lesern, die sich für die historische Entwicklung von Mathematik und Logik als solche interessieren, sei die Lektüre von [Kneale & Kneale, 1962] nahegelegt.

Die frühesten Wurzeln der Typentheorien findet man schon bei Russel [Russel, 1908] und später in konstruktiver Form bei Church [Church, 1940]. Dennoch hat sich aufgrund der verschiedenen Detailprobleme immer noch keine "optimale" Formulierung herauskristallisiert. Die Unterschiede der verschiedenen konstruktiven Theorien liegen in Notationen, Effizienz der herleitbaren Algorithmen, Ausdruckskraft der Basissprache und Größe des Notwendigen formalen Apparates. Wir werden kaum dazu kommen, alle Varianten vorzustellen, die in manchen Bereichen sicherlich eleganter sind als die Typentheorie von NuPRL, in anderen dafür aber wiederum Schwächen aufweisen. Die wichtigsten Veröffentlichungen hierzu sind [Smith, 1984, Andrews, 1986, Girard, 1986, Coquand & Huet, 1988, Girard *et.al.*, 1989, Horn, 1988, Backhouse *et.al.*, 1988a, Backhouse *et.al.*, 1988b, Backhouse, 1989, Paulin-Mohring, 1989, Coquand, 1990, Galmiche, 1990] und das Buch [Nordström *et.al.*, 1990].

Weitere Literaturhinweise werden wir zu den einzelnen Themen am Ende jedes Kapitels geben.

Kapitel 2

Formalisierung von Logik, Berechenbarkeit und Typdisziplin

In diesem Kapitel wollen wir die mathematischen Grundlagen vorstellen, die für ein Verständnis formaler Methoden beim Beweis mathematischer Aussagen und bei der Entwicklung korrekter Programme notwendig sind. Wir werden eine Einführung in die Grundkonzepte formaler Beweisführung geben und dies an drei fundamentalen Formalismen illustrieren.

Die *Prädikatenlogik* (siehe Abschnitt 2.2) ist den meisten als abkürzende Schreibweise für logische Zusammenhänge bekannt. Als formales System erlaubt sie, Aussagen zu beweisen, deren Gültigkeit ausschließlich aus der logischen Struktur folgt, ohne daß man dazu die Bedeutung der Einzelaussagen kennen muß. Sie ist ein sehr mächtiges Handwerkzeug, wenn es gelingt, alle notwendigen (und gültigen) Voraussetzungen als logische Teilaussagen zu formulieren. Gerechnet werden kann innerhalb der Prädikatenlogik allerdings nicht.

Der *λ-Kalkül* (siehe Abschnitt 2.3) ist eines der einfachsten Modelle für die Klasse aller berechenbaren Funktionen. Er erlaubt symbolisches Rechnen auf Termen und bietet gleichzeitig ein klar definiertes und einfaches Konzept, um logische Schlußfolgerungen über das Verhalten von Programmen zu ziehen.

Die *einfache Typentheorie* (siehe Abschnitt 2.4) wurde geschaffen, um den vielen Unentscheidbarkeitsfragen zu begegnen, die ein Turing-mächtiger Formalismus wie der λ-Kalkül mit sich bringt. Durch Hinzunahme einer Typdisziplin, welche die Selbstanwendbarkeit von Operationen – die Hauptursache der Unentscheidbarkeiten – deutlich einschränkt, lassen sich viele Programmeigenschaften typisierbarer Algorithmen leichter beweisen.

2.1 Formale Kalküle

Eines der größten Probleme, das normalerweise entsteht, wenn Mathematiker oder Programmierer versuchen, jemand anderen von der Richtigkeit ihrer Behauptungen zu überzeugen, ist die Mehrdeutigkeit der natürlichen Sprache, die sie hierbei verwenden. Dies liegt zum einen daran, daß in der natürlichen Sprache häufig Worte und Formulierungen verwendet werden, denen keine präzise, allgemeingültige Bedeutung zugewiesen werden kann. Das Wort *Baum*, zum Beispiel, hat sehr verschiedene Bedeutungen, je nachdem, in welchem Kontext es verwandt wird. Darüberhinaus gibt es aber auch Sätze, die mehrere Interpretationen zulassen. Was bedeutet zum Beispiel der folgende Satz?

Wenn eine gerade Zahl eingegeben wird, so gibt mein Programm eine Eins aus.

Man sollte meinen, daß dies eine präzise Beschreibung eines einfachen Sachverhalts ist. Aber können wir nun schließen, daß bei ungeraden Zahlen *keine* Eins ausgegeben wird? Das ist legitim, sofern mit *wenn* in Wirklichkeit *nur wenn* gemeint ist, ansonsten aber nicht. Gerade die bisherige Praxis der Validierung von Software zeigt, daß die Verwendung natürlicher Sprache beim logischen Schließen immer wieder die Gefahr von Mißverständnissen und Trugschlüssen in sich birgt. Um diesem Problem zu begegnen, hat man in der Mathematik schon frühzeitig *Kalküle* entwickelt.

Von ihrem ursprünglichen Sinn her sind Kalküle nichts anderes als formale Rechenvorschriften, welche es ermöglichen, bei der Lösung von Problemen nach einem einmal bewiesenen Schema vorzugehen, ohne über dessen Rechtfertigung weiter nachzudenken. Ein tieferes Verständnis ist nicht mehr nötig. Alles, was man zu tun hat, ist, stur einer vorgegebenen Menge von *Regeln* zu folgen, bis man am gewünschten Ziel ist.

Die meisten von Ihnen werden aus der Schule die Kalküle der Differential- und Integralrechnung kennen. Die Summenregel $d/dx(u+v) = du/dx + dv/dx$, die Produktregel $d/dx(u*v) = du/dx*v + u*dv/dx$ und andere Regeln zur Bestimmung von Ableitungsfunktionen wurden ein für allemal bewiesen und ab dann nur noch schematisch angewandt. Oft werden Sie die Rechtfertigung dieser Regeln längst vergessen haben und dennoch in der Lage sein, sie anzuwenden.

Genau besehen sind diese Regeln nichts anderes als syntaktische Vorschriften, wie man symbolische Objekte manipulieren bzw. erzeugen darf, um zum Ziel zu kommen. Auch wenn man diese Symbole im konkreten Fall mit einer Bedeutung verbindet – also bei Anwendung der Produktregel anstelle der Symbole u und v konkrete Funktionen im Blickfeld hat – so ist es bei der Verarbeitung eigentlich gar nicht notwendig, diese Bedeutung zu kennen. Alles, was man zu beachten hat, ist daß die Regeln korrekt angewandt werden, d.h. die Symbole korrekt manipuliert werden. So ist es möglich, *komplexere Aufgaben alleine durch symbolisches Rechnen zu lösen*, wie zum Beispiel bei der Berechnung der folgenden Ableitung.

$$\frac{d}{dx}(x^2 + 2x + 4) = \frac{d}{dx}x^2 + \frac{d}{dx}2x + \frac{d}{dx}4 = \ldots = 2x + 2$$

Dieses schematische Anwenden von Regeln zur Manipulation symbolischer Objekte als Ersatz für mathematische Schlußfolgerungen ist genau das Gebiet, für das Computer geschaffen wurden. Alle "Berechnungen", die ein Computerprogramm durchführt, sind im Endeffekt nichts anderes als Manipulationen von Bits (Worten, Symbolen), die mit bestimmten Bedeutungen (Zahlen, Texten etc.) in Verbindung gebracht werden. Die Idee, dieses Anwendungsgebiet auf das mathematische Schließen auszudehnen, um Unterstützung für komplexere Anwendungsgebiete zu schaffen, ist deshalb relativ naheliegend. Programme wie MACSYMA oder MATHEMATICA sind genau aus der Überlegung entstanden, daß für das Lösen bestimmter Aufgaben keine Intelligenz, sondern nur formale Umformungen erforderlich sind.[1]

Die Beschreibung eines formalen Kalküls gliedert sich in eine *formale Sprache*, bestehend aus *Syntax* und *Semantik*, und ein System von *(Inferenz-)Regeln*. Die Syntax der formalen Sprache legt fest, welche äußere Struktur formale Ausdrücke haben müssen, um – unabhängig von ihrer Bedeutung – überhaupt als solche akzeptiert werden zu können. Die Semantik der Sprache ordnet dann den syntaktisch korrekten Ausrücken eine Bedeutung zu. Die Regeln geben an, wie syntaktisch korrekte Grundausdrücke (*Axiome*, *Atome*, o.ä.) zu bilden sind bzw. wie aus bestehenden Ausdrücke neue Ausdrücke erzeugt werden können. Dabei ist beabsichtigt, daß alle Regeln die Semantik der Ausdrücke berücksichtigen. Im Falle logischer Kalküle heißt das, daß alle durch Regeln erzeugbare Ausdrücke *wahr* im Sinne der Semantik sein sollen. Mit Hilfe von formalen Kalkülen wird es also möglich, daß Computer mathematische Aussagen *beweisen*.

Im folgenden wollen wir nun diese Konzepte im Detail besprechen und am Beispiel der (hoffentlich) bekannten Prädikatenlogik erster Stufe, die im nächsten Abschnitt dann vertieft wird, illustrieren.

2.1.1 Syntax formaler Sprachen

Mit der Syntax wird die formale Struktur einer Sprache beschrieben. Sie legt fest, in welcher textlichen Erscheinungsform ein informaler Zusammenhang innerhalb einer formalen Sprache repräsentiert werden kann. Normalerweise verwendet man hierzu eine *Grammatik*, welche die "syntaktisch korrekten" Sätze über einem vorgegebenen *Alphabet* charakterisiert. Häufig wird diese Grammatik in *Backus-Naur Form* oder einem ähnlichen Formalismus notiert, weil dies eine Implementierung leichter macht. Ebenso gut – und für das Verständnis

[1] Die meisten der bisher bekannten Programmpakete dieser Art sind für Spezialfälle konzipiert worden. Wir werden uns im folgenden auf universellere Kalküle konzentrieren, die – zumindest vom Prinzip her – in der Lage sind, das logische Schließen als solches zu simulieren. Natürlich setzen wir dabei wesentlich tiefer an als bei den Spezialfällen und entsprechend dauert es länger bis wir zu gleich hohen Ergebnissen kommen. Dafür sind wir dann aber vielseitiger.

besser – ist aber auch die Beschreibung durch mathematische Definitionsgleichungen in natürlichsprachlichem Text. Wir werden diese Form im folgenden bevorzugt verwenden.

Beispiel 2.1.1 (Syntax der Prädikatenlogik – informal)

Die Sprache der Prädikatenlogik erster Stufe besteht aus Formeln, die aus Termen, Prädikaten, und speziellen logischen Symbolen (*Junktoren* und *Quantoren*) aufgebaut werden.

Terme bezeichnen mathematische *Objekte* wie Zahlen, Funktionen oder Mengen. Zu ihrer Beschreibung benötigt man eine (unendliche) Menge von *Variablen* sowie eine Menge von *Funktionssymbolen*, wobei jedes Funktionssymbol eine *Stelligkeit* $n \geq 0$ besitzt. Ein *Term* ist entweder eine Variable oder eine Funktionsanwendung der Form $f(t_1, \ldots, t_n)$, wobei f ein n-stelliges Funktionssymbol ist und alle t_i Terme sind. Nullstellige Funktionssymbole werden auch *Konstanten(-symbole)* genannt; der Term $c()$ wird in diesem Fall zur Abkürzung einfach als c geschrieben.

Formeln beschreiben mathematische *Aussagen* wie z.B. die Tatsache, daß das Quadrat einer geraden Zahl durch 4 teilbar ist. Hierzu benötigt man eine Menge von *Prädikatssymbolen*, wobei jedes Prädikatssymbol eine *Stelligkeit* $n \geq 0$ besitzt. Eine *atomare Formel* hat die Gestalt $P(t_1, \ldots, t_n)$, wobei P ein n-stelliges Prädikatssymbol ist und alle t_i Terme sind. Eine *Formel* ist entweder eine atomare Formel oder von der Gestalt $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $\forall x.A$, $\exists x.A$, oder (A), wobei A und B Formeln sind und x eine Variable ist. Jede Formel enthält somit zumindest ein Prädikatssymbol.

Ein paar *Konventionen* ermöglichen Abkürzungen beim Aufschreiben von Formeln. Bei Formeln mit nullstelligen Prädikatssymbolen wird wie bei den nullstelligen Funktionen die Klammer weggelassen. Statt $P()$ schreiben wir kurz P. *Prioritäten* sind ein weiteres wichtiges Hilfsmittel: $\neg$ bindet am stärksten, dann folgen $\wedge$, $\vee$, und $\Rightarrow$ in absteigender Reihenfolge; der *Bindungsbereich* eines Quantors reicht so weit nach rechts wie möglich, d.h. Quantoren binden am schwächsten. Diese Konventionen erlauben es, den syntaktischen Aufbau der Formel $\forall$`x. g(x)` $\wedge \neg$`v(x)` $\Rightarrow$ `v(sqr(x))` eindeutig zu rekonstruieren. Sie entspricht der Formel $\forall$`x. (` `(g(x)` $\wedge$ `(`$\neg$`v(x))` $\Rightarrow$ `v(sqr(x)) ).`

Die Sprache der Prädikatenlogik erster Stufe wird dazu verwandt, mathematische Aussagen zu präzisieren und auf ihren logischen Kern zu reduzieren. Diese Reduktion gilt insbesondere für Funktions- und Prädikatssymbole, die innerhalb dieser Sprache – im Gegensatz zu Junktoren und Quantoren – keine Bedeutung besitzen, selbst wenn die benutzten Namen eine solche suggerieren. Dies genau ist auch der Vorteil der Prädikatenlogik, denn bei den logischen Schlüssen darf es auf die konkrete Bedeutung der Terme und Prädikate nicht ankommen. Was zählt, ist ausschließlich die logische Struktur.[2]

Eine präzisierte Version dieser Beschreibung werden wir in Abschnitt 2.2.1 geben.

Als abkürzende Schreibweise für logische Zusammenhänge ist die Prädikatenlogik ein weit verbreitetes Mittel. Dennoch findet man eine Fülle von verschiedenen Notationen. So wird zuweilen auch A`&`B oder A `AND` B statt $A \wedge B$ geschrieben, A`|`B oder A `OR` B statt $A \vee B$, und $A \rightarrow B$ oder A `IMPLIES` B statt $A \Rightarrow B$. Bei den Quantoren entfällt zuweilen der Punkt (wie in $\exists x\ A$) oder er wird durch Klammern ersetzt (wie in $(\exists x)A$). Der Bindungsbereich von Quantoren reicht manchmal nur bis zum *ersten* $\wedge$, $\vee$, oder $\Rightarrow$.

Dies macht deutlich, wie wichtig es ist, eine eindeutige Syntax für die verwendete formale Sprache zu vereinbaren, damit es beim Lesen von Formeln nicht zu Mißverständnissen kommen kann. Für den Einsatz von Computerunterstützung ist es ohnehin notwendig, die Syntax der Sprache präzise festzulegen.

2.1.2 Semantik formaler Sprachen

Die Semantik ordnet den syntaktisch korrekten Sätzen einer formalen Sprache eine Bedeutung zu. Sie gibt uns die Handhabe, darzustellen, daß wir den Satz $\forall$`x. teilbar(x,2)` $\Rightarrow$ `teilbar(x`^{2}`,4)` für wahr halten,

[2] Deshalb ist es durchaus legitim, den Satz "*das Quadrat einer geraden Zahl ist durch 4 teilbar*" auf verschiedene Arten zu formalisieren, z.B. als $\forall$`x. teilbar(x,2)` $\Rightarrow$ `teilbar(x`^{2}`,4)` oder als $\forall$`x. gerade(x)` $\Rightarrow$ `vierteilbar(x`^{2}`)` oder gar ganz kurz $\forall$`x. g(x)` $\Rightarrow$ `v(sqr(x))`. Die Gültigkeit dieses Satzes hängt nicht von der Wahl der Symbole ab, sondern davon, welche Formeln mit diesen Symbolen bereits als gültig vorausgesetzt werden.

den Satz $\forall$`x. teilbar(x,2)` $\Rightarrow$ `teilbar(x`^{2}`,5)` aber nicht. Die Semantik einer formalen Sprache wird üblicherweise durch eine *Interpretation* der einzelnen Symbole beschrieben.[3] Eine solche Interpretation stellt die Beziehung zwischen einer zu definierenden formalen Sprache (*Quellsprache*) zu einer zweiten (*Zielsprache*) her. In den meisten Fällen ist letztere eine weniger formale Sprache, von der man aber annimmt, daß sie eine klare Bedeutung besitzt.[4]

Fast alle mathematischen Theorien werden in der Cantorschen Mengentheorie interpretiert: jeder Term entspricht einer Menge von Objekten und jede Formel einer Aussage über solche Mengen. Die Mengentheorie selbst wird als Grundlage aller Mathematik betrachtet und nicht weiter in Frage gestellt. Ihre Axiome stützen sich auf informale, aber allgemein anerkannte Intuitionen wie "*Es gibt Mengen*" und "*die Vereinigung zweier Mengen ist wieder eine Menge*".[5]

Beispiel 2.1.2 (Semantik der Prädikatenlogik – informal)

Eine Interpretation der Prädikatenlogik weist jedem logischen Symbol ein mengentheoretisches Objekt zu. Variablen, Funktionen und Prädikate werden alle mit Bezug auf eine feste Menge, das sogenannte *Universum* erklärt. Jede Variable wird einem einem Element dieses Universums zugeordnet, jedes n-stellige Funktionssymbol einer konkreten n-stelligen Funktion auf Elementen des Universums und jedes n-stellige Prädikatssymbol einer n-stelligen Relation über Elementen des Universums. Vorschriften, *welche* Elemente, Funktionen oder Relationen den einzelen Symbolen zugeordnet werden müssen, gibt es nicht.

Eine Interpretation weist somit jedem Term einen Wert (ein Element) zu und macht jede atomare Formel entweder *wahr* oder *falsch*. Der Wahrheitsgehalt zusammengesetzter Formeln wird nach festen Regeln ermittelt, die unabhängig von der konkreten Interpretation gelten:

Die *Negation* $\neg A$ ist genau dann wahr, wenn A falsch ist.
Die *Konjunktion* $A \wedge B$ ist genau dann wahr, wenn sowohl A als auch B wahr sind.
Die *Disjunktion* $A \vee B$ ist genau dann wahr, wenn eine der beiden Formeln A oder B wahr ist.
Die *Implikation* $A \Rightarrow B$ ist genau dann wahr, wenn B wahr ist, wann immer A wahr ist.
Die *Universelle Quantifizierung* $\forall x\,.\,A$ ist genau dann wahr, wenn A für jedes mögliche x wahr ist.
Die *Existentielle Quantifizierung* $\exists x\,.\,A$ ist genau dann wahr, wenn A für mindestens ein x wahr ist.

Dabei sind A und B beliebige Formeln und x eine Variable.

2.1.3 Klassische und intuitionistische Mathematik

Die oben angegebene Semantik der Prädikatenlogik ist in der Mathematik allgemein akzeptiert. Jedoch läßt die natürlichsprachliche Formulierung einige Fragen offen, die sich vor allem die Art der Beweise auswirkt, welche als legitim anerkannt werden. Kann man zum Beispiel davon ausgehen, daß *jede* Formel entweder wahr oder falsch ist? Folgt die Wahrheit einer Formel $A \vee B$ aus der Tatsache, daß nicht beide falsch sein können? Ist es für die Wahrheit einer Formel der Gestalt $\exists x\,.\,A$ essentiell, das konkrete Element x benennen zu können, welches die Formel A wahr macht? Formal ausgedrückt ist das die Frage nach der Allgemeingültigkeit der folgenden logischen Gesetze:

[3] Dies ist die sogenannte *denotationale* Semantik. Es sei aber erwähnt, daß auch weitere Formen existieren wie z.B. die *operationale* Semantik des λ-Kalküls (siehe Abschnitt 2.3.2), welche beschreibt, wie λ-Terme auszurechnen sind.

[4] Um eine derartige Annahme wird man nicht herumkommen, auch wenn man sich bemüht, die Semantik der Zielsprache zu präzisieren. Im Endeffekt wird man sich immer auf eine Zielsprache abstützen müssen, die ihre Bedeutung der natürlichen Sprache entnimmt.

[5] Daß diese intuitive Abstützung nicht ohne Gefahren ist, zeigt der Zusammenbruch des gesamte Gedankengebäudes der Mathematik zu Beginn dieses Jahrhunderts, als das Russelsche Paradoxon das bis dahin vorherrschende Mengenkonzept zerstörte. Die Arbeit vieler Mathematiker – wie z.B. die von G. Frege [Frege, 1879, Frege, 1892], der die Mathematik zu formalisieren versuchte – wurde hierdurch zunichte gemacht. Bei der heutigen Formulierung der Mengentheorie ist man sich aber sehr sicher, daß sie keine Widersprüche zuläßt.

Es gilt $A \vee \neg A$ *(Gesetz vom ausgeschlossenen Dritten)*
Aus $\neg(\neg A)$ folgt A
Aus $\neg(\neg A \wedge \neg B)$ folgt $A \vee B$
Aus $\neg(\forall x\,.\,\neg A)$ folgt $\exists x\,.\,A$

Während diese Gesetze unstreitig für die *meisten* logischen Aussagen gültig sind, die man für A und B einsetzen kann, gibt es doch Fälle, in denen die Anwendung dieser Gesetze für viele "mathematisch unverbildete" Menschen zumindest problematisch ist.

Beispiel 2.1.3

In der Analysis wurde der Satz *"Es gibt zwei irrationale Zahlen x und y so, daß x^y rational ist"* zunächst wie folgt bewiesen

> *Die Zahl $\sqrt{2}^{\sqrt{2}}$ ist entweder rational oder irrational. Ist sie rational, so können wir $x{=}y{=}\sqrt{2}$ wählen, denn $\sqrt{2}$ ist bekanntermaßen irrational. Anderenfalls wählen wir $x{=}\sqrt{2}^{\sqrt{2}}$ und $y{=}\sqrt{2}$, die nun beide irrationale Zahlen sind, und bekommen $x^y{=}2$, also eine rationale Zahl.*

Dieser Beweis macht massiv Gebrauch vom Gesetz des ausgeschlossenen Dritten, läßt aber die meisten Leser unzufrieden. Selbst wenn wir die Beweisführung akzeptieren, wissen wir immer noch nicht, welches die beiden Zahlen sind, deren Existenz bewiesen wurde.[6]

Intuitionismus (*intuitionistische Mathematik*) ist eine mathematische Denkrichtung, welche diese Art von Beweisführung und die Allgemeingültigkeit des Gesetzes vom ausgeschlossenen Dritten ablehnt. Anders als die sogenannte *klassische* Mathematik, die sich mittlerweile weitgehend in Schulen und Universitäten durchgesetzt hat, geht sie davon aus, daß mathematische Aussagen *konstruktiv* zu verstehen sind. In dieser Sicht besagt eine Existenzaussage der Form $\exists x\,.\,A$, daß man im Endeffekt ein konkretes Element angeben kann, welches die Aussage A erfüllt, und eine Disjunktion $A \vee B$, daß man sagen kann, welcher der beiden Fälle gültig ist.

Diese Forderung macht mathematische Beweise natürlich schwieriger als solche, bei denen es ausreicht, das Gegenteil einer Aussage zu widerlegen. In der Mathematik des Endlichen – solange man also ausschließlich über natürliche und rationale Zahlen, Listen, *endliche* Mengen (von Zahlen o.ä.) usw. spricht – ergeben sich hierduch auch keine anderen Ergebnisse,[7] so daß die Forderung eher als eine lästige Einschränkung erscheint. Sobald aber unendliche Objekte wie *unendliche* Mengen (wie z.B. die Menge der natürlichen Zahlen), reelle Zahlen, Funktionen oder Programme in den Aussagen vorkommen, werden auch die Ergebnisse unterschiedlich und der Streit über die Richtigkeit mancher Aussage ist heute noch nicht geklärt.

Für das Schließen über Programme wird dieses Problem besonders deutlich. Bei der Verifikation eines gegebenen Programmes mag es noch angehen, wegen der einfacheren Beweise klassische Logik zu verwenden, auch wenn dies in vielen Teilen bereits recht problematisch wird. Es macht aber wenig Sinn, die Existenz eines Programms mit einer gewünschten Eigenschaft dadurch zu beweisen, daß man die Behauptung widerlegt, alle Programme würden diese Eigenschaft nicht besitzen.

Daher werden wir uns im folgenden mit beiden Denkrichtungen, der klassischen und der intuitionistischen Logik beschäftigen müssen. Zum Glück ist der Unterschied *formal* sehr gering. Klassische Logik ergibt sich aus der intuitionistischen durch Hinzufügen eines einzigen Gesetzes, da alle logischen Gesetze, die einen Unterschied zwischen den beiden ausmachen, unmittelbare Folgerungen des Gesetzes vom ausgeschlossenen Dritten sind.

[6] Es gibt übrigens auch einen direkten Beweis mit $x = \sqrt{2}$, $y = 2 * ld3$ und $x^y = 3$.

[7] Im Endlichen ist es rein hypothetisch immer möglich, *alle* möglichen Fälle hinzuschreiben und einzeln zu untersuchen. Wenn ich also gezeigt habe, daß nicht alle Elemente x eine Eigenschaft A nicht haben (d.h. $\neg(\forall x\,.\,\neg A)$), dann weiß ich, daß ich bei der Untersuchung aller Möglichkeiten dieses Element finden werde. Sobald es aber unendlich viele Möglichkeiten gibt, besteht diese Chance im allgemeinenn nicht mehr.

2.1.4 Objekt- und Metasprache

In den vorhergehenden Teilabschnitten haben wir bereits öfter Aussagen über Formeln gemacht. Wir haben gesagt, daß $A \vee B$ eine Formel ist, wenn A und B Formeln sind, und über Gesetze wie das vom ausgeschlossenen Dritten gesprochen, was wir kurz mit "*Es gilt* $A \vee \neg A$" ausgedrückt haben. Dabei war klar, daß die Symbole A und B nicht selbst Formeln (also atomare Formeln, die nur das nullstellige Prädikatssymbol A bzw. B enthalten) sind, sondern *Platzhalter* für beliebige Formeln. Diese Platzhalter erlauben uns, Aussagen *über* logische Formeln – wie z.B. logische Gesetze – kurz und prägnant zu beschreiben, anstatt sie durch lange und kaum verständliche Sätze auszudrücken. Der Satz "*Aus der Negation der Konjunktion der Negation zweier Formeln folgt die Disjunktion der beiden Formeln*" besagt dasselbe wie "*aus* $\neg(\neg A \wedge \neg B)$ *folgt* $A \vee B$", ist aber erheblich schwerer zu verstehen.

Die Verwendung von Platzhaltern für Formeln, Terme und Variablen macht aber auch klar, daß man im Zusammenhang mit formalen Kalkülen zwei Ebenen, nämlich die *Objektsprache* und die *Metasprache* des Kalküls, unterscheiden muß.

- Die *Objektsprache* eines Kalküls ist die formale Sprache, in welcher die Objekte formalisiert werden, über die wir formale Schlüsse ziehen wollen. Bei der Prädikatenlogik ist dies also eine Sprache, die aus Termen und Formeln besteht.
- Die *Metasprache*[8] des Kalküls ist diejenige Sprache, die wir benutzen, wann immer wir Aussagen *über* den Kalkül und seine Objektsprache machen. Sie wird benötigt, um den Aufbau der Objektsprache und vor allem auch die Inferenzregeln zu beschreiben. Normalerweise wird hierfür eine natürliche Sprache verwendet. Wir benutzen einfaches Deutsch um über Kalküle zu reden. Da dies aber aus den obengenannten Gründen manchmal unverständlich wird, verwenden wir zusätzlich eine mathematische Notation, in der Symbole der Objektsprache (wie $\forall$ und $\wedge$) und Platzhalter für objektsprachliche Ausdrücke – sogenannte *syntaktische Metavariablen* – vorkommen.

Um Objekt- und Metasprache voneinander zu trennen, werden wir hierzu verschiedene Zeichensätze verwenden. Alle Ausdrücke der Objektsprache werden im `typewriter`-font geschrieben – so, wie sie auch im Computer Einsatz finden werden. Dabei ist jedoch zu bedenken, daß 8-bit fonts auch mathematische Symbole wie $\forall$ und $\wedge$ enthalten können. Zugunsten der besseren Lesbarkeit werden wir Schlüsselworte der Sprache durch **Fettdruck** in einem ähnlichen Zeichensatz hervorheben.

Syntaktische Metavariablen werden im mathematischen Zeichensatz *kursiv* gedruckt. Dabei werden wir bestimmte Symbole bevorzugt für bestimmte Arten von Objekten verwenden.

- Namen für Variablen der Objektsprache sind normalerweise $x, y, z, x_1, x_2, \ldots$
- Namen für Funktionssymbole sind $f, g, h, \ldots$
- Namen für Terme sind $t, u, v, r, s, \ldots$
- Namen für Prädikatssymbole sind $P, Q, R, \ldots$
- Namen für Formeln sind $A, B, C, \ldots$

Diese Konventionen[9] ersparen uns, syntaktische Metavariablen immer wieder neu deklarieren zu müssen.

2.1.5 Definitorische Erweiterung

Bei der Entwicklung eines formalen Kalküls versucht man normalerweise, mit einem möglichst kleinen Grundgerüst auszukommen. Dies erleichtert die Definition von Syntax und Semantik und macht vor allem den

[8] Die griechische Vorsilbe *meta* steht für "über", "jenseits von" und ähnliches.

[9] Diese Liste wird später implizit um weitere Konventionen erweitert. Die Symbole n, m, i, j, k, l bezeichnen Zahlen, griechische Buchstaben Γ, Δ, Θ stehen für Mengen und Listen von Formeln, T, S für Bereiche (Typen), usw.

Nachweis leichter, daß die Ableitungsregeln tatsächlich der Semantik der formalen Sprache entsprechen und nicht etwa Widersprüche enthalten. Bei der Formalisierung natürlichsprachlicher Zusammenhänge möchte man dagegen eine möglichst umfangreiche Sprache zur Verfügung haben, damit eine Formulierung nicht derart umfangreich wird, daß sie nicht mehr zu verstehen ist.

Ein einfaches Mittel, sowohl der theoretischen Notwendigkeit für einen kleinen Kalkül als auch der praktischen Anforderung nach einer großen Menge vordefinierter Konzepte gerecht zu werden ist die sogenannte *konservative Erweiterung* einer formalen Sprache durch *definitorische Abkürzungen*. Dieses in der Mathematik gängige Konzept, bei dem ein komplexer Zusammenhang durch einen neuen abkürzenden Begriff belegt wird, läßt sich größtenteils auch auf formale Theorien übertragen. Es wird einfach eine Definition aufgestellt, die einen Ausdruck der bestehenden formalen Sprache durch einen neuen abkürzt, wobei syntaktische Metavariablen eingesetzt werden dürfen.

Beispiel 2.1.4 (Definitorische Erweiterungen der Prädikatenlogik)

In der Syntax der Prädikatenlogik (siehe Beispiel 2.1.1 auf Seite 11) vermißt man die *Äquivalenz* "*A gilt, genau dann wenn B gilt*", die üblicherweise mit $A \Leftrightarrow B$ bezeichnet wird. Sie läßt sich aber auf zwei Implikationen zurückführen, nämlich "*A gilt, wenn B gilt*" und "*B gilt, wenn A gilt*". Damit können wir $A \Leftrightarrow B$ als Abkürzung für $(A \Rightarrow B) \wedge (B \Rightarrow A)$ betrachten. Wir benutzen hierfür folgende Schreibweise.

$$A \Leftrightarrow B \quad \equiv \quad (A \Rightarrow B) \ \wedge \ (B \Rightarrow A)$$

In ähnlicher Weise könnte man die Prädikatenlogik um eine exklusive Disjunktion $A \dot{\vee} B$ erweitern, die ausdrückt, daß genau eine der beiden Formeln A und B wahr ist.

Durch die konservative Erweiterung einer formalen Sprache gewinnen wir also Flexibilität – zumal für die textliche Form der Abkürzungen keine Einschränkungen gelten außer, daß es in einem Zeichensatz beschreibbar ist und alle syntaktischen Metavariablen der rechten Seite auch links vorkommen müssen. Die guten Eigenschaften des bisherigen Kalküls, an dem wir ja überhaupt nichts ändern, bleiben dagegen erhalten.

2.1.6 Inferenzsysteme

Durch die Definition der Semantik ist die Bedeutung von objektsprachlichen Ausdrücken eindeutig festgelegt. Rein hypothetisch wäre es möglich, diese Bedeutung dadurch zu bestimmen, daß man die Interpretation präzisiert und dann den "Wert" eines Ausdruck ausrechnet. Bei dieser Vorgehensweise wären formale Kalküle jedoch nur wenig mehr als eine abkürzende Schreibweise, da das Bewerten einer Aussage im wesentlichen doch wieder von Hand geschehen muß. Sinnvoller ist es, syntaktische Aussagen *direkt* zu manipulieren, ohne ihren Wert zu bestimmen, und dabei die Möglichkeiten für eine syntaktische Manipulation so einzugrenzen, daß dem Wert der Ausdrücke Rechnung getragen wird, selbst wenn sich das textliche Erscheinungsbild ändert.

Es gibt zwei Möglichkeiten zur syntaktischen Manipulation, *Konversion* und *Inferenz*. Bei der *Konversion* werden Ausdrücke der Objektsprache in semantisch äquivalente umgeformt.[10] Im Falle der Prädikatenlogik werden also logische Formeln in andere Formeln mit gleichem Wahrheitswert umgewandelt, also zum Beispiel $A \wedge (A \vee B)$ in A. Im Gegensatz dazu geht es bei der *Inferenz* darum, die Gültigkeit einer mathematischen Aussage zu *beweisen* ohne dabei über deren Bedeutung nachzudenken. Inferenzregeln haben daher die Gestalt "*aus* A_1 *und ... und* A_n *darf ich* C *schließen*", wie zum Beispiel "*aus* A *und* $A \Rightarrow B$ *folgt* B" ("*modus ponens*").

Auch wenn Konversionsregeln eher über *Werte* und Inferenzregeln eher über *Gültigkeit* reden, können Konversionsregeln meist auch als Inferenzregeln geschrieben werden. Jedoch kann man Konversionsregeln in beide Richtungen lesen, während Inferenzregeln nur in einer festen Richtung anwendbar sind. Für das logische Schließen sind Inferenzregeln vielseitiger, denn sie erlauben auch, Aussagen abzuschwächen.

Inferenzsysteme, oft auch einfach *Kalkül* genannt, bestehen aus einer Menge von Regeln zur Manipulation objektsprachlicher Ausdrücke. Eine Regel der Art "*aus* A_1 *und ... und* A_n *folgt* C" wird häufig in schematischer Form aufgeschrieben:

$$\frac{A_1, \ldots, A_n}{C}$$

[10] Der Kalkül der Differentialrechnung ist ein Konversionskalkül, bei dem Gleichheiten die Umformungsregeln bestimmen.

Dabei sind die Aussagen $A_1, \ldots, A_n$ die *Prämissen* dieser Regel und C die *Konklusion*. Die Anzahl n der Prämissen darf auch Null sein. In diesem Falle nennt man die Regel auch ein *Axiom*, da die Konklusion ohne jede Voraussetzung Gültigkeit hat. Axiome werden oft auch separat von den "eigentlichen" Regeln betrachtet, da sie mit allgemeingültigen Grundaussagen des Kalküls identifiziert werden können.

Genaugenommen bezeichnet die Schreibweise $\frac{A_1,\ldots,A_n}{C}$ ein Regel*schema*, da die Prämissen und die Konklusion, wie bereits erwähnt, syntaktische Metavariablen enthalten. Die Anwendung eines Regelschemas auf konkrete Prämissen wird mit $\underline{\Gamma \vdash_{rs} C}$ bezeichnet. Dabei ist Γ die Menge der konkreten Prämissen und C die konkrete Konklusion und rs die Bezeichnung für das jeweilige Regelschema. $\Gamma \vdash_{rs} C$ gilt, wenn die syntaktischen Metavariablen der Regel rs so durch Formeln ersetzt werden können, daß die Prämissen von rs in der Menge Γ enthalten sind und C die Konklusion ist.

Theoreme sind mathematische Aussagen, die sich durch Anwendung von endlich vielen Regeln *ableiten* lassen. Dabei ist Ableitbarkeit wie folgt definiert.

Definition 2.1.5 (Ableitbarkeit)

Es sei RG die Menge der Regeln und AX die Menge der Axiome eines Kalküls. Γ sei eine (endliche) Menge von Formeln der Logiksprache und C eine Formel der formalen Sprache. Dann heißt C aus Γ ableitbar, wenn es eine Folge $rs_1 \ldots rs_n$ von Regeln aus RG und eine eine Folge von Formeln $C_1 \ldots C_n$ mit $C_n = C$ gibt, derart, daß gilt

$$\Gamma_1 \vdash_{rs_1} C_1, \quad \ldots, \quad \Gamma_j \vdash_{rs_j} C_j, \quad \ldots, \quad \Gamma_n \vdash_{rs_n} C_n$$

wobei Γ_j eine Menge von Formeln bezeichnet, welche alle Axiome des Kalküls, alle Formeln aus Γ und alle bisher abgeleiteten Formeln C_l mit $1 \leq l < j$ enthält. (Also $\Gamma_j = AX \cup \Gamma \cup \{C_l | 1 \leq l < j\}$, $\Gamma_1 = AX \cup \Gamma$)

Falls Γ ausschließlich Axiome enthält, dann bezeichnen wir C als ableitbar oder als Theorem.

Bei einer formalen Darstellung läßt sich eine Ableitung als ein Baum beschreiben, an dessen Wurzel das Theorem steht und dessen Knoten durch Regeln markiert sind. Die Blätter dieses Baumes müssen entweder aus Axiomen – oder bei einer stärkeren Strukturierung – aus anderen Theoremen bestehen.

Mit Hilfe von formalen Kalkülen wird es also möglich, daß Computer mathematische Aussagen *beweisen*. Wir müssen aber dennoch eine scharfe Trennung zwischen *Wahrheit* und *Beweisbarkeit* ziehen. Im Idealfall sollte jedes bewiesene Theorem wahr im Sinne der Semantik sein und umgekehrt jede wahre Aussage beweisbar sein. Das ist jedoch nicht immer möglich. Deshalb müssen wir die folgenden zwei Begriffe prägen.

Definition 2.1.6 (Korrektheit und Vollständigkeit)

1. *Eine Inferenzregel heißt korrekt, wenn aus der Gültigkeit aller Prämissen immer auch die Gültigkeit der Konklusion (semantisch) folgt. Ein Kalkül heißt korrekt, wenn jede seiner Inferenzregeln korrekt ist.*

2. *Ein Kalkül heißt vollständig, wenn jede semantisch wahre Aussage ein Theorem, also mit den Inferenzregeln des Kalküls beweisbar ist.*

In korrekten Kalkülen ist also jedes Theorem des Kalküls semantisch wahr. Dies ist eine Grundvoraussetzung, die man an Kalküle stellen muß, denn Kalküle mit inkorrekten Regeln haben für die Beweisführung einen geringen praktischen Wert.[11] Vollständigkeit dagegen ist bei reichhaltigen formalen Sprachen praktisch nicht mehr zu erreichen. Wenn eine Theorie die gesamte Arithmetik umfaßt, dann kann es – so der Gödelsche Unvollständigkeitssatz – hierfür keine vollständigen Kalküle mehr geben.

[11] Da Korrektheit relativ zu einer Semantik definiert ist, muß der Begriff zuweilen relativiert werden, wenn die Semantik nicht eindeutig fixiert ist. Im Falle der Prädikatenlogik z.B. hängt die Semantik von der konkreten Interpretation ab. Ein korrekter Kalkül sollte für jede mögliche Interpretation korrekt sein. Ein Kalkül heißt *konsistent* (*widerspruchsfrei*) , wenn die Regeln einander nicht widersprechen, also noch mindestens eine Interpretation zulassen. Konsistenz wird immer dann interessant, wenn sich die Semantik einer formalen Sprache noch nicht genau angeben läßt. In diesem Falle bestimmt der Kalkül, welche Semantik überhaupt noch möglich ist.

Axiomenschemata:

(A1)	$A \Rightarrow A$	(A11)	$(A \wedge B \vee C) \Rightarrow (A \vee C) \wedge (B \vee C)$
(A2)	$A \Rightarrow (B \Rightarrow A)$	(A12)	$(A \vee C) \wedge (B \vee C) \Rightarrow (A \wedge B \vee C)$
(A3)	$(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))$	(A13)	$(A \vee B) \wedge C \Rightarrow (A \wedge C \vee B \wedge C)$
(A4)	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$	(A14)	$(A \wedge C \vee B \wedge C) \Rightarrow (A \vee B) \wedge C$
(A5)	$A \Rightarrow A \vee B$	(A15)	$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$
(A6)	$A \Rightarrow B \vee A$	(A16)	$A \wedge \neg A \Rightarrow B$
(A7)	$(A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \vee B \Rightarrow C))$	(A17)	$(A \wedge (A \Rightarrow B)) \Rightarrow B$
(A8)	$A \wedge B \Rightarrow A$,	(A18)	$(A \wedge C \Rightarrow B) \Rightarrow (C \Rightarrow (A \Rightarrow B))$
(A9)	$A \wedge B \Rightarrow B$	(A19)	$(A \Rightarrow (A \wedge \neg A)) \Rightarrow \neg A$
(A10)	$(C \Rightarrow A) \Rightarrow ((C \Rightarrow B) \Rightarrow (C \Rightarrow A \wedge B))$		

Ableitungsregel:

(mp) $\dfrac{A \ , \ A \Rightarrow B}{B}$

Abbildung 2.1: Frege–Hilbert–Kalkül für die Aussagenlogik

Es sollte an dieser Stelle auch erwähnt werden, daß ein Kalkül nur die Regeln festlegt, mit denen Beweise geführt werden dürfen, um zu garantieren, daß alle formalen Beweise auch korrekten logischen Schlüssen entsprechen. Ein Kalkül ist dagegen keine Methode, um Beweise zu *finden*.[12]

Es gibt zwei Ausrichtungen bei Kalkülen. Die meisten von ihnen arbeiten *synthetisch* (bottom-up), d.h. sie beschreiben, wie man von einer vorgegebenen Menge von Axiomen auf eine neue Aussage schließen kann. Dies erscheint wenig zielgerichtet, aber dafür sind die Regeln oft einfacher formuliert, da man präzsiser beschreiben kann, wie man vorzugehen hat. Bei einem *analytischen* Kalkül beschreiben die Regeln den Weg von einer Zielaussage zu den Axiomen. Die Beweise sind zielgerichtet (top-down), aber die Regeln benötigen mehr Kontrolle, da festzulegen ist, welche Informationen noch gebraucht werden und welche weggeworfen werden dürfen. Generell ist zu berücksichtigen, daß es *den* Kalkül für eine formale Sprache nicht gibt. Regeln und Axiome können so gewählt werden, daß die gestellte Aufgabe bestmöglich gelöst werden kann. Eine andere Aufgabenstellung kann einen anderen Kalkül sinnvoll erscheinen lassen. Wir werden im folgenden die wichtigsten Kalkülarten kurz vorstellen.

2.1.6.1 Frege–Hilbert–Kalküle

Die ersten logischen Kalküle wurden von Gottlob Frege gleichzeitig mit der Entwicklung der Sprache der Prädikatenlogik entworfen. Diese wurden später von David Hilbert (siehe zum Beispiel [Hilbert & Bernays, 1934, Hilbert & Bernays, 1939]) weiterentwickelt, weshalb man sie heute als *Frege–Hilbert–Kalküle* oder auch als *Hilberttypkalküle* bezeichnet. Sie bestehen typischerweise aus relativ vielen Axiomen und wenigen "echten" Kalkülregeln. Sie sind synthetischer Natur, d.h. es werden Formeln ausgehend von Axiomen mittels Schlußregeln abgeleitet. Im Prinzip sind sie sehr mächtig, da sie kurze Beweise von Theoremen ermöglichen. Jedoch ist wegen der synthetischen Natur die *Suche* nach einem Beweis für ein gegebenes Problem recht aufwendig.

[12] Die rechnergestützte *Suche* nach Beweisen ist ein Thema der *Inferenzmethoden*, deren Schwerpunkte in gewissem Sinne komplementär zu denen dieser Veranstaltung sind. Die dort besprochenen Verfahren wie Konnektionsmethode und Resolution sind allerdings bisher nur auf die Prädikatenlogik erster Stufe (und Modallogiken) anwendbar. Zu komplexeren Kalkülen wie der Typentheorie ist bisher kein allgemeines Beweissuchverfahren bekannt und es ist auch unwahrscheinlich, daß es ein solches je geben wird. Wir werden uns daher damit begnügen müssen, zu jedem Teilkalkül/-aspekt methodische Hinweise zu geben, die wir später dann zum Teil als Beweistaktiken automatisieren.

Als Beispiel für einen Frege–Hilbert–Kalkül haben wir in Abbildung 2.1 einen Ableitungskalkül für die *Aussagenlogik* – also die Prädikatenlogik ohne Quantoren – zusammengestellt. Dabei haben wir bei den Axiomen darauf verzichtet, die leere Liste der Prämissen mit aufzuführen. Axiom (A1) schreiben wir daher kurz $A \Rightarrow A$ anstatt $\frac{}{A \Rightarrow A}$.

Der Kalkül in Abbildung 2.1 ist korrekt und vollständig für die *intuitionistische Aussagenlogik*. Für die klassische Aussagenlogik muß man die Axiome (A17) bis (A19) entfernen und stattdessen das Gesetz vom ausgeschlossenen Dritten $A \vee \neg A$ als neues Axiom (A17) hinzufügen. Es sei nochmals erwähnt, daß für die syntaktischen Metavariablen jede beliebige Formel eingesetzt werden darf.

2.1.6.2 Natürliche Deduktion

Mit den Frege-Hilbert–Kalkülen konnte das logische Schließen soweit präzisiert werden, daß Zweifel an der Gültigkeit von Aussagen zumindest im Prinzip ausgeschlossen werden können. Sie sollten es daher ermöglichen, mathematische Beweise auf eine völlig formale Gestalt zu bringen. Gerhard Gentzen hat jedoch darauf hingewiesen, daß Mathematiker eine Form von Beweisen verwenden, die sich von den Frege–Hilbert–Ableitungen wesentlich unterscheiden, so daß sich mathematische Beweise nicht leicht in solcher Weise formalisieren lassen. Von Gentzen stammt daher ein Kalkül [Gentzen, 1935], der dieses natürliche Schließen wesentlich unmittelbarer widerspiegelt. Bei diesem *Kalkül des natürlichen Schließens* (von Gentzen mit *NJ* bzw. *NK*[13] bezeichnet) handelt es sich ebenfalls um einen synthetischen Kalkül, der sich daher nicht besonders gut für die das Finden von Beweisen eignet. Die Nähe zu der "natürlichen" Form des mathematischen Schließens läßt ihn jedoch zu einem guten Bindeglied zwischen automatisch gestützten und menschlichen Schließen werden.

Die Grundidee dieses Kalküls ist die folgende. Wenn ein Mathematiker eine Formel der Gestalt $A \wedge B$ beweisen will, dann wird er hierzu separat A und B beweisen. Will er eine Formel der Gestalt $A \Rightarrow B$ beweisen, dann nimmt er zunächst an, A gelte, und versucht dann, aus dieser Annahme die Gültigkeit von B abzuleiten. Solche Annahmen spielen eine wichtige Rolle, denn sie enthalten Informationen, die bei der weiteren Beweisführung noch verwendet werden können. Hat eine Annahme zum Beispiel die Gestalt $B \wedge C$, so können wir sowohl die Formel B als auch C im Beweis benutzen. Gentzen hat den Kalkül des natürlichen Schließens sehr symmetrisch ausgelegt. Zu jedem logischen Symbol gibt es zwei Arten von Regeln, eine *Einführungsregel* (–I) und eine *Eliminationsregel* (–E):

- Die Einführungsregel beantwortet die Frage "*unter welchen Voraussetzungen kann ich auf die Gültigkeit einer Formel schließen?*". Die Einführungsregel $\wedge$–I für die Konjunktion $\wedge$ besagt zum Beispiel, daß man auf die Gültigkeit von $A \wedge B$ schließen kann, wenn man A und B als Prämissen hat: $\frac{A \quad B}{A \wedge B}$
- Die Eliminationsregel beantwortet die Frage "*welche Informationen folgen aus einer gegebenen Formel?*". Die beiden Eliminationsregeln $\wedge$–E besagen zum Beispiel, daß man aus $A \wedge B$ sowohl A als auch B folgern kann, was zu den zwei Regeln $\frac{A \wedge B}{A}$ und $\frac{A \wedge B}{B}$ führt.

Jedes logische Zeichen ist durch die Beantwortung dieser beiden Fragen eindeutig charakterisiert. Im Normalfall sind Einführungs- und Eliminationsregel invers zueinander, d.h. wenn wir eine Konjunktion $A \wedge B$ eliminieren und anschließend wieder einführen haben wir weder Informationen gewonnen noch verloren.

Anders als in Frege–Hilbert–Kalkülen behandeln alle Regeln jeweils nur *ein* logisches Zeichen. Dadurch wird der Kalkül sehr klar und einfach. Die Details des Kalküls *NJ* für die Aussagenlogik sind in Abbildung 2.2 zusammengestellt. Das Zeichen Λ steht hierbei für den logischen *Widerspruch* ("falsch").[14] Auch dieser Kalkül ist korrekt und vollständig für die intuitionistische Aussagenlogik. Bemerkenswerterweise benötigt er nicht ein einziges Axiom.[15] Der Grund hierfür ist die Tatsache, daß manche Regeln es ermöglichen, einmal getroffene Annahmen wieder zu entfernen.

[13] Die Abkürzungen stehen für die Natürliche Deduktion für intuitionistische (*J*) bzw. klassische (*K*) Logik.

[14] Es wird benötigt, um in dem von Gentzen angestrebten klaren Stil die Bedeutung der Negation zu erklären.

[15] Für die klassische Aussagenlogik müsste man das Axiom $A \vee \neg A$ ergänzen. Ansonsten sind die Kalküle identisch, was ein weiterer Vorteil der Gentzen'schen Kalküle des Natürlichen Schließens gegenüber Frege–Hilbert–Kalkülen ist.

$$\Lambda\text{-E}\quad \frac{\Lambda}{A}$$

$$\neg\text{-I}\quad \frac{\begin{matrix}[A]\\ \Lambda\end{matrix}}{\neg A} \qquad \neg\text{-E}\quad \frac{\neg A \;\; A}{\Lambda}$$

$$\wedge\text{-I}\quad \frac{A \;\; B}{A \wedge B} \qquad \wedge\text{-E}\quad \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$$\vee\text{-I}\quad \frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \qquad \vee\text{-E}\quad \frac{A \vee B \quad \begin{matrix}[A]\\ C\end{matrix} \quad \begin{matrix}[B]\\ C\end{matrix}}{C}$$

$$\Rightarrow\text{-I}\quad \frac{\begin{matrix}[A]\\ B\end{matrix}}{A \Rightarrow B} \qquad \Rightarrow\text{-E}\quad \frac{A \;\; A \Rightarrow B}{B}$$

Abbildung 2.2: Kalkül des Natürlichen Schließens für die Aussagenlogik

Wir wollen dies am Beispiel der Einführungsregel für die Implikation erläutern. Diese Regel, die dem Modus Ponens gleichkommt, drückt aus, daß die Implikation $A \Rightarrow B$ genau dem Gedanken "*aus A folgt B*" entspricht. Sie verlangt als Prämisse nur, daß B *unter der Annahme*, daß A wahr ist, erfüllt sein muß (was wir dadurch kennzeichnen, daß wir $[A]$ in eckigen Klammern über B stellen). Hieraus darf man $A \Rightarrow B$ schließen, ohne daß es hierfür noch der Annahme A bedarf. Von den bisher getroffenen Annahmen kann A also entfernt werden.

Zur Illustration geben wir eine Ableitung der Formel `((A ⇒ B) ∧ (B ⇒ C)) ⇒ (A ⇒ C)` (dies entspricht im Prinzip dem Axiom (A3) im Frege–Hilbert–Kalkül) in diesem System an. Zunächst zeigen wir, wie der zugehörige Beweis eines Mathematikers in etwa aussehen würde.

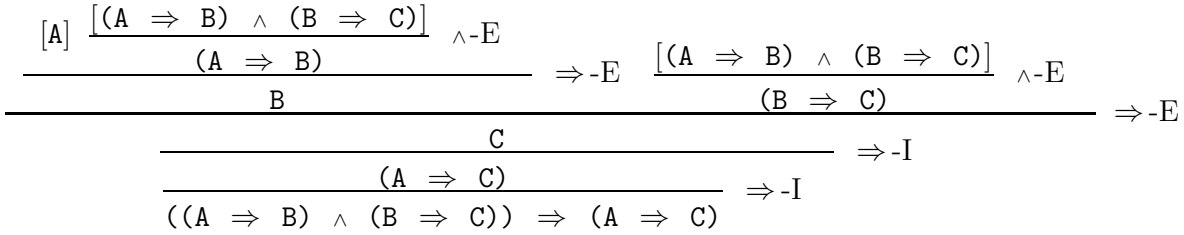
1. Wir nehmen an `(A ⇒ B) ∧ (B ⇒ C)` sei erfüllt
2. Wir nehmen weiter an, daß `A` gilt.
3. Aus der ersten Annahme folgt `(A ⇒ B)`
4. und mit der zweiten dann auch `B`.
5. Aus der ersten Annahme folgt auch, daß `(B ⇒ C)` gilt
6. und mit der vierten dann auch `C`.
7. Es ergibt sich also, daß `C` unter der Annahme `A` gilt. Also folgt `A ⇒ C`
8. Insgesamt ergibt sich also, daß `A ⇒ C` unter der Annahme `(A ⇒ B) ∧ (B ⇒ C)` gilt. Damit folgt die Behauptung: es gilt `((A ⇒ B) ∧ (B ⇒ C)) ⇒ (A ⇒ C)`

Die entsprechende Ableitung im Kalkül des natürlichen Schließens ist die folgende:

(1)	`(A ⇒ B) ∧ (B ⇒ C)`	Annahme
(2)	`A`	Annahme
(3)	`(A ⇒ B)`	∧-E angewendet auf (1)
(4)	`B`	⇒-E angewendet auf (2) und (3)
(5)	`(B ⇒ C)`	∧-E angewendet auf (1)
(6)	`C`	⇒-E angewendet auf (4) und (5)
(7)	`(A ⇒ C)`	⇒-I angewendet auf (2) und (6) — *Annahme (2) entfällt*
(8)	`(A ⇒ B) ∧ (B ⇒ C) ⇒ (A ⇒ C)`	⇒-I angewendet auf (1) und (7) —*Annahme (1) entfällt*

Es ist offensichtlich, daß jeder der acht Schritte des natürlichen Beweises in unmittelbarer Weise von dem entsprechenden Schritt des formalen Beweises wiedergegeben wird und umgekehrt. Man kann diesen

noch weiter formalisieren und verkürzen, indem man eine Baumschreibweise verwendet, also einfach alle Ableitungsschritte übereinander schreibt. Der Beweis sieht dann wie folgt aus.



Auch wenn der Zusammenhang zum informalen mathematischen Schließen deutlich zu erkennen ist, ist der Kalkül des natürlichen Schließens nicht ganz leicht zu handhaben. Besonders die Verwendung von Regeln, die einmal getroffene Annahmen wieder entfernen können, bereitet Schwierigkeiten. Sie machen es notwendig, bei der Beweisführung jederzeit den gesamten bisherigen Beweisbaum zu berücksichtigen und eine Übersicht darüber zu behalten, welche Annahmen noch gelten und welche schon entfernt wurden.

2.1.6.3 Sequenzenkalküle

Gentzen selbst hatte bereits die globale Sicht mit der notwendigen Verwaltung von Annahmen als ein Problem bei den Kalkülen der natürlichen Deduktion erkannt. Er entwickelte daher in der gleichen Arbeit [Gentzen, 1935] einen Kalkül, der die Behandlung von Annahmen während einer Ableitung überflüssig machte, aber die grundsätzliche Sichtweise des natürlichen Schließens beibehielt. Die Modifikationen sind verhältnismäßig einfach. Anstatt Formeln als Kern des logischen Schließens zu betrachten, stellte Gentzen sogenannte *Sequenzen* in den Mittelpunkt, die aus einer Formel zusammen mit der Liste der aktuell gültigen Annahmen bestehen. Sie haben die äußere Form

$$A_1, \ldots, A_n \vdash C \qquad (n \geq 0)$$

und bedeuten, daß die Formel C von den Annahmen $A_1,\ldots,A_n$ abhängt.[16] Regeln handhaben nunmehr Sequenzen anstelle logischer Formeln. Auf den ersten Blick scheint dies nicht mehr als ein formaler Trick zu sein, der nur zusätzliche Schreibarbeit einbringt. Der Vorteil aber ist, daß *Sequenzenkalküle* eine *lokale* Sicht auf Beweise ermöglichen. In jedem Beweisschritt ist es nun nur noch notwendig, die aktuelle Sequenz zu kennen.[17]

Bis auf diese Modifikation sind Sequenzenbeweise praktisch identisch mit Beweisen im Kalkül des natürlichen Schließens und die Regeln des Sequenzenkalküls ergeben sich unmittelbar aus den Regeln des natürlichen Schließens. Wie zuvor dienen die Einführungsregeln für ein logisches Zeichen dazu, Formeln mit dem entsprechenden Zeichen zu erzeugen. Da manche Einführungsregeln aber auch Annahmen – also Formeln der linken Seite – entfernen, benötigen wir nun Regeln, welche zu jedem logischen Zeichen eine entsprechende Formel auf der Annahmenseite erzeugen. Diese Regeln übernehmen die Rolle der Eliminationsregeln, denn sie bearbeiten wie sie die Frage "*welche Informationen folgen aus einer gegebenen Formel?*".

Wir wollen dies am Beispiel der Regeln für die Konjunktion erläutern. Die Eliminationsregeln des logischen Schließens besagt, daß man aus $A \wedge B$ sowohl A als auch B folgern kann, also $\frac{A \wedge B}{A}$ und $\frac{A \wedge B}{B}$. Im Sequenzenkalkül wird dieser Gedanke nun mit dem Konzept der Konsequenzen von Annahmen verbunden. Da A aus $A \wedge B$ gefolgert werden darf, dürfen wir schließen, daß jede Aussage, die aus A und einer Menge anderer Annahmen – bezeichnet mit Γ – folgt, mit Sicherheit auch aus $A \wedge B$ und Γ folgt: $\frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C}$.

[16] Semantisch gesehen entspricht eine Sequenz einem *Urteil*, welches besagt, daß eine mathematische Aussage (bezeichnet durch die Formel C) aus einer Reihe von anderen Aussagen folgt. Der Kalkül simuliert also eigentlich nicht das logische Schließen über Aussagen sondern liefert Begründungen für die Gültigkeit bestimmter logischer Schlüsse. Allerdings kann man mathematische Aussagen als spezielle Urteile ansehen, nämlich daß die Aussage ohne Voraussetzungen folgt.

[17] Sequenzenkalküle sind daher ideal für eine Kooperation von Mensch und Computer beim Führen formaler Beweise. Die lokale Sicht ermöglicht es dem Menschen, den Beweis zu steuern, während der Computer für die korrekte Anwendung der Regeln sorgt.

Axiom	$\dfrac{}{\Gamma, A \vdash A}$	Schnitt	$\dfrac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}$
		Λ-E	$\dfrac{}{\Gamma, \Lambda \vdash A}$
$\neg$-I	$\dfrac{\Gamma, A \vdash \Lambda}{\Gamma \vdash \neg A}$	$\neg$-E	$\dfrac{\Gamma \vdash A}{\Gamma, \neg A \vdash \Lambda}$
$\wedge$-I	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$	$\wedge$-E	$\dfrac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C} \quad \dfrac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}$
$\vee$-I	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$	$\vee$-E	$\dfrac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}$
$\Rightarrow$-I	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\Rightarrow$-E	$\dfrac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C}$

Abbildung 2.3: Sequenzenkalkül für die Aussagenlogik

In erster Näherung werden Eliminationsregeln also genau in umgekehrter Reihenfolge aufgeschrieben. Statt die Formel auf der rechten Seite zu eliminieren wird sie auf der linken Seite erzeugt.[18] Abbildung 2.3 faßt die entsprechenden Regeln für die intuitionistische Aussagenlogik zusammen.[19] Man beachte dabei, daß die syntaktischen Metavariablen Γ und Δ beliebige *Mengen* von Formeln bezeichnen. Die hervorgehobenen Formeln einer Regel dürfen also an jeder beliebigen Stelle in der Liste der Annahmen stehen.[20] Die Axiom-Regel ist nötig, um Annahmen tatsächlich auch verwenden zu können. Die Schnittregel erlaubt eine bessere Strukturierung von Beweisen (man beweist erst ein Teilziel und verwendet es dann), ist aber nicht unbedingt erforderlich. In seinem Hauptsatz [Gentzen, 1935] zeigte Gentzen, daß alle Schnittregeln aus einem Beweis gefahrlos entfernt werden können. Allerdings vergrößert sich der Beweis dadurch erheblich.

Wie die bisher vorgestellten Kalküle sind Sequenzenkalküle eigentlich synthetischer Natur. Ein Beweis wird von den Axiomen ausgehend entwickelt bis man am gewünschten Ziel ist. Allerdings lassen sich die Regeln verhältnismäßig leicht so umwandeln, daß sie einen analytischen Sequenzenkalkül liefern. Man muß hierzu lediglich die Richtung der Regeln umkehren und gegebenenfalls Steuerungsparameter ergänzen, welche diejenigen Informationen geben, die nach einer synthetischen Anwendung einer Regel verlorengegangen sind.

So muß zum Beispiel bei der Umkehrung der Schnittregel $\dfrac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}$ angegeben werden, welche Formel A für den Schnitt benutzt wird, da diese ja nachher nicht mehr vorhanden ist. Ebenso kann man in der Zielsequenz die Aufteilung der Annahmen in Γ und Δ nicht erkennen. Um den Kalkül nicht unnötig zu verkomplizieren, übernimmt man bei einem analytischen Vorgehen einfach alle Annahmen und entscheidet später, welche man davon benötigt. Insgesamt bekommt die Regel die Gestalt

$$\frac{\Gamma \vdash C}{\Gamma \vdash A \quad \Gamma, A \vdash C} \;\; \texttt{cut}\ A.$$

Weitere Details, eine computernähere Präsentation und eine semantische Rechtfertigung der Regeln des analytischen Sequenzenkalküls werden wir im folgenden zusammen mit der Prädikatenlogik besprechen.

[18] Entsprechend bezeichnete Gentzen die Regeln nun als Linksregeln (*-l*) und Rechtsregeln (*-r*). Diese Konvention wird von vielen Mathematikern übernommen während Computersysteme meist auf den Begriffen Einführung und Elimination basieren.

[19] Der ursprüngliche Sequenzenkalkül *LK* für die klassische Aussagenlogik ist etwas komplizierter als der hier angegebene intuitionistische Kalkül *LJ*, da er mehrere Formeln auch auf der rechten Seite zuläßt, die als mögliche Alternativen (also eine Art Disjunktion) betrachtet werden. Der Kalkül *LJ* kann allerdings auch durch Abschwächung der Regel Λ-E zu $\frac{\Gamma, \neg A \vdash \Lambda}{\Gamma \vdash A}$ in einen Kalkül für die klassische Aussagenlogik umgewandelt werden.

[20] Es sei angemerkt, daß Gentzens ursprüngliche Sequenzenkalküle *Folgen* von Annahmen verarbeiten, bei denen es von Bedeutung ist, wo und wie oft eine Formel in den Annahmen auftritt. Für die Vollständigkeit dieses Kalküls werden daher neben den in Abbildung 2.3 angegebenen logischen Regeln auch sogenannte *Strukturregeln* zur Vertauschung, Kontraktion und Verdünnung von Annahmen benötigt. Diese Vorgehensweise ist bei einer mechanischen Abarbeitung einer Sequenz als Text (z.B. in einem Computer) erforderlich, wird bei einer formalen Beweisführung durch Menschen jedoch eher als unnötiger Ballast angesehen.

2.2 Prädikatenlogik

Unter allen formalen Sprachen ist die *Prädikatenlogik* am weitesten verbreitet. Für den eingegrenzten Bereich logischer Schlüsse, deren Gültigkeit sich ausschließlich auf die Struktur mathematischer Aussagen stützt, ist sie ein überaus praktisches und mächtiges Handwerkzeug. Im Prinzip läßt sich jede mathematische Aussage in der Sprache der Prädikatenlogik formulieren. Man muß hierzu nur die entsprechenden mathematischen Begriffe in der Form von Prädikats- und Funktionssymbolen beschreiben. Da diese Symbole allerdings keine festgelegte Bedeutung haben – selbst wenn die Namensgebung dies suggerieren mag – ist nicht festgelegt, bis zu welchem Detail man die Aussage vor ihrer Formalisierung mit Hilfe logischer Strukturierungsmechanismen zerlegen sollte. Die optimale Form ergibt sich aus dem Verwendungszweck.

Beispiel 2.2.1 (Formalisierung mathematischer Aussagen)

Faßt man die Aussage "*Alle Studenten, die ihre Prüfungsleistungen innerhalb von zwei Jahren ablegen, haben die Diplomhauptprüfung bestanden*" als eine logische Einheit der Diplomprüfungsordnung auf, so wird man sie vielleicht nicht weiter unterstrukturieren und sie vielleicht durch das Prädikat

`DPO_18_4`

ausdrücken. Diese Formulierung enthält allerdings nicht die geringsten Hinweise auf die intendierte Bedeutung und wird nur sehr eingeschränkt verwendbar sein. Etwas mehr Klarheit gewinnt man schon, wenn man fixiert, daß es hier um eine Bestimmung geht, die für alle Studenten Gültigkeit hat und zudem eine "wenn-dann" Beziehung enthält. In diesem Sinne präzisiert lautet die Aussage "*Für alle Studenten gilt: wenn sie ihre Prüfungsleistungen innerhalb von zwei Jahren ablegen, dann haben sie die Diplomhauptprüfung bestanden*''. Bei der Formalisierung verwendet man hierzu als logische Symbole den Allquantor $\forall$ sowie die Implikation $\Rightarrow$ und schreibt:

$\forall$`student. Prüfungen_in_zwei_Jahren_abgelegt(student)` $\Rightarrow$ `Diplom_bestanden(student)`

oder mit etwas kürzeren Prädikatsnamen

$\forall$`s. P_abgelegt(s)` $\Rightarrow$ `Diplom(s)`

Bei der zweiten Form zeigt sich ein Dilemma. Durch die Verwendung weniger suggestiver Variablen- und Prädikatsnamen geht ein Teil der Informationen verloren, die vorher vorhanden waren. Genau besehen waren diese jedoch niemals vorhanden, da die Prädikatenlogik über die Bedeutung ihrer Variablen-, Funktionen- und Prädikatsnamen nichts aussagt, wenn man nicht weitere Informationen beisteuert.

Wieviele dieser Informationen man explizit hinzugibt, ist reine Ermessenssache. So kann es für die Bestimmung relevant sein, daß es sich bei den Objekten, die mit dem Symbol `s` bezeichnet werden, um Studenten handelt. Es mag aber sein, daß die Bestimmung auf andere Objekte niemals angewandt werden wird, und dann ist es eigentlich unbedeutend, wofür `s` nun genau steht. Sicherlich von Interesse aber ist, auszudrücken, daß die Dauer der Prüfung nicht mehr als zwei Jahre betrug. So liest sich unsere obige Aussage nun wie folgt: "*Für alle Objekte* `s` *gilt: wenn* `s` *ein Student ist und* `s` *seine Diplomprüfungen abgelegt hat und die Dauer dieser Prüfungen maximal 2 Jahre war, dann hat* `s` *die Diplomhauptprüfung bestanden*". Hier wird einerseits ein funktionaler Zusammenhang (die Dauer der Prüfung) benutzt und eine Verkettung von Voraussetzungen. Bei der Formalisierung verwendet man hierzu Funktionssymbole sowie die Konjunktion $\wedge$ und schreibt:

$\forall$`s. Student(s)` $\wedge$ `abgelegt(s,prüfung)` $\wedge$ $\leq$`(dauer(prüfung), 2)` $\Rightarrow$ `Diplom(s)`

Man könnte die Strukturierung jetzt noch weitertreiben, da sie immer noch einige Fragen offen läßt. Endgültige Eindeutigkeit wird man jedoch nie erreichen können. An irgendeinem Punkt muß man sich darauf verlassen, daß die Formulierung auf der Basis der bisher gewählten Begriffe (und möglicher Zusatzinformationen über diese) für den Anwendungszweck hinreichend klar ist.[21]

[21] Die Typentheorie, die wir in Kapitel 3 vorstellen werden geht einen anderen Weg. Sie geht – wie die Mengentheorie – davon aus, daß wenige Objekte wie natürliche Zahlen und Funktionen elementarer Natur sind, deren Regeln erforscht werden müssen. Alle anderen Konzepte sind dann auf der Basis einer festen Formalisierung dieser Objekte auszudrücken. Das heißt, daß jede Formalisierung eine feste Bedeutung erhält (sofern sie nicht Variable eines Quantors ist).

Die übliche Prädikatenlogik verwendet zur Strukturierung mathematischer Aussagen die in Beispiel 2.1.2 auf Seite 12 erklärten Operatoren *Negation*, *Konjunktion*, *Disjunktion*, *Implikation*, *Universelle Quantifizierung* und *Existentielle Quantifizierung*, deren Bedeutung intuitiv leicht zu erklären ist. Nur bei der Bedeutung der Quantoren bleiben einige Fragen offen, da es erlaubt ist, über beliebige Objekte zu quantifizieren, deren Natur völlig im unklaren liegt. Dies ist sicherlich legitim, wenn man die Logik verwendet, um immer wieder über den gleichen Bereich – also zum Beispiel natürliche Zahlen – zu reden. Hier stimmt die Annahme, daß alle Variablen, Funktionen und Prädikate mit Bezug auf eine festes *Universum* erklärt werden können, und man ist sich darüber im klaren, daß der Satz $\forall x.\ x{=}0 \vee x{\geq}1$ eine hinreichend präzise Formalisierung einer mathematischen Grundeigenschaft ist. Das wird aber sofort anders, wenn der gleiche Satz als Eigenschaft reeller Zahlen angesehen wird, denn nun fehlen ein paar Voraussetzungen. Denkt man bei der Interpretation gar an Objekte wie Graphen, Mengen oder Liste, so wird der Satz schlicht sinnlos.

In der Prädikatenlogik behilft man sich zuweilen damit, daß man Informationen über den Bereich, aus dem ein Objekt gewählt werden darf, als Voraussetzung in die Aussage hineinnimmt. Man schreibt zum Beispiel

$$\forall x.\ nat(x) \Rightarrow (x{=}0 \vee x{\geq}1)$$

und nimmt separat die Axiome der natürlichen Zahlen hinzu. Auf diese Art kann man die bisherige Sprache beibehalten, handelt sich aber für das formale Beweisen eine große Menge zu verarbeitender Formeln ein.

Eine Alternative hierzu ist die Erweiterung der Prädikatenlogik zu einer *Sortenlogik*, indem man die Informationen über die Natur der Objekte als eine Art *Typdeklaration* in den Quantor mit hineinnnimmt und dann die Formel wie folgt aufschreibt:

$$\forall x{:}\mathbb{N}.\ x{=}0 \vee x{\geq}1$$

Die obige Schreibweise ist nicht nur eleganter, sondern ermöglicht auch eine klarere Charakterisierung der Semantik einer Aussage. Zudem wird sie der Denkweise moderner Programmiersprachen gerechter, bei denen man eine Typdisziplin für Variablen nicht mehr missen möchte. Dafür muß der Formalismus allerdings die Bezeichner für die Sorten mitschleppen, selbst wenn diese nicht weiter verarbeitet werden. Da wir spätestens bei der Besprechung der Typentheorie in Kapitel 3 die Bereiche, aus denen ein Objekt gewählt werden darf, ohnehin zum zentralen Thema machen werden, wollen wir bei der nun folgenden Präzisierung der Prädikatenlogik die Sorten mit in den Formalismus mit aufnehmen. Die bekannte unsortierte Prädikatenlogik, die wir im Abschnitt 2.1 bereits informal erklärt haben, ergibt sich hieraus einfach durch Herausstreichen jeglicher Sorteninformationen.

Ein weiterer Nachteil der Prädikatenlogik gegenüber der gängigen mathematischen Notation ist die strikte Verwendung der Präfixschreibweise auch dort, wo die Mathematik Infixoperatoren wie $+$, $-$, $*$ usw. benutzt.[22] Die einzige Ausnahme von dieser Regelung wird zuweilen bei der *Gleichheit* gemacht, da Gleichheit ein derart fundamentales Konzept ist, daß man ihre Bedeutung nicht freistellen sollte. Deshalb nehmen wir – wie viele Formulierungen der Prädikatenlogik – auch die Gleichheit als fest definiertes Prädikatssymbol in Infixschreibweise mit in die formale Sprache auf, deren Syntax und Semantik wir nun festlegen wollen.

2.2.1 Syntax

Die Sprache der Prädikatenlogik besteht aus Formeln, die wiederum aus Termen, Prädikaten, und speziellen logischen Symbolen (*Junktoren* und *Quantoren*) aufgebaut werden. Zur Präzisierung von Termen benötigt man Funktions- und Variablensymbole, die aus bestimmten, nicht näher präzisierten Alphabeten ausgewählt werden.

[22] Auch dieses Problem werden wir erst im Zusammenhang mit der Besprechung der Typentheorie in Kapitel 3 lösen.

Definition 2.2.2 (Terme)

Es sei $\mathcal{V}$ ein Alphabet von Variablen(-symbolen). Es sei $\mathcal{F}^i$ ein Alphabet von i-stelligen Funktionssymbolen für jedes $i \geq 0$ und $\mathcal{F} = \bigcup_{i=0}^{\infty} \mathcal{F}^i$.

Die Terme der Sprache der Prädikatenlogik sind induktiv wie folgt definiert.

- *Jede Variable $x \in \mathcal{V}$ ist ein Term.*
- *Ist $f \in \mathcal{F}^0$ ein beliebiges 0-stelliges Funktionssymbol, dann ist f ein Term (eine Konstante).*
- *Sind $t_1, \ldots, t_n$ Terme ($n \geq 1$) und ist $f \in \mathcal{F}^n$ ein beliebiges n-stelliges Funktionssymbol, dann ist $f(t_1, \ldots, t_n)$ ein Term.*

Man beachte hierbei, daß Variablen- und Funktionssymbole nicht unbedingt aus einzelnen Buchstaben bestehen müssen. Es ist durchaus legitim, Namen wie `x14`, $\mathtt{f}_1$, `student` und ähnliches zu verwenden. Auch ist nicht unbedingt gefordert, daß die Alphabete $\mathcal{V}$ und $\mathcal{F}$ disjunkt sein müssen. Meist geht es aus dem Kontext hervor, um welche Art von Symbolen es sich handelt. Es hat sich jedoch eingebürgert, die auf Seite 14 eingeführten Konventionen für syntaktische Metavariablen in ähnlicher Form auch als Konventionen für die Namensgebung von Symbolen der Objektsprache zu verwenden, um Mißverständnisse zu vermeiden. Wir wollen die Bildung von Termen gemäß der obigen Regeln nun anhand einiger Beispiele veranschaulichen.

Beispiel 2.2.3

Die folgenden Ausdrücke sind korrekte Terme im Sinne von Definition 2.2.2

`x`, `24`, `peter`, `vater(peter)`, `max(2,3,4)`, `max(plus(4,plus(5,5)),23,5)`

Dabei ist die Rolle von `x`, `24`, `peter` ohne unsere Namenskonventionen nicht eindeutig festzustellen. Üblicherweise betrachten wir `x` als Variable und `24` genauso wie `peter` als nullstelliges Funktionssymbol, d.h. als Konstante. `vater`, `max` und `plus` müssen Funktionszeichen sein, wobei `vater` einstellig, `max` dreistellig, und `plus` zweistellig ist.

Der Ausdruck `max(2,3)` wäre an dieser Stelle erlaubt, aber problematisch. Zwar ist prinzipiell nicht ausgeschlossen, daß ein Symbol mit verschiedenen Stelligkeiten benutzt wird (die $\mathcal{F}^i$ müssen nicht disjunkt sein). Eine Verwendung von `max` als zwei- und als dreistelliges Symbol wird aber die meisten Inferenzsysteme überlasten, so daß man auf derartige Freiheiten besser verzichten sollte.

Die Definition prädikatenlogischer Formeln ist strukturell der eben aufgestellten Definition von Termen sehr ähnlich. Wir definieren sie als atomare und als induktiv zusammengesetzte Formeln.

Definition 2.2.4 (Formeln)

Es sei für jedes $i \geq 0$ $\mathcal{P}^i$ ein Alphabet von i-stelligen Prädikatssymbolen, $\mathcal{P} = \bigcup_{i=0}^{\infty} \mathcal{P}^i$ und $\mathcal{T}$ ein Alphabet von Bereichssymbolen (Typen).

Die Formeln der Sprache der Prädikatenlogik sind induktiv wie folgt definiert.

- *Λ ist eine (atomare) Formel.*
- *Ist $P \in \mathcal{P}^0$ ein 0-stelliges Prädikatssymbol, dann ist P eine atomare Formel (oder Aussagenvariable).*
- *Sind t_1 und t_2 Terme, dann ist $t_1 = t_2$ eine atomare Formel.*
- *Sind $t_1, \ldots, t_n$ Terme ($n \geq 1$) und ist $P \in \mathcal{P}^n$ ein beliebiges n-stelliges Prädikatssymbol, dann ist $P(t_1, \ldots, t_n)$ eine atomare Formel.*
- *Sind A und B Formeln, $x \in \mathcal{V}$ ein Variablensymbol und $T \in \mathcal{T}$ ein Bereichssymbol, dann sind*

  $$\neg A, \quad A \wedge B, \quad A \vee B, \quad A \Rightarrow B, \quad \forall x{:}T\,.\,A, \quad \exists x{:}T\,.\,A \quad \text{und} \quad (A)$$

  Formeln.

Man beachte, daß im Unterschied zur Definition 2.2.2 die Argumente von Prädikatssymbolen nicht Formeln, sondern Terme sind. Das Symbol Λ ist ein spezielles 0-stelliges Prädikatssymbol, das für die atomare Aussage "*falsch*" verwendet wird. Das Symbol `=` ist ein spezielles 2-stelliges Prädikatssymbol, das die gewöhnliche Gleichheit ausdrückt und in Infixnotation verwendet wird. Auch hierzu wollen wir ein paar Beispiele ansehen.

Beispiel 2.2.5

Die folgenden Ausdrücke sind korrekte Formeln im Sinne von Definition 2.2.4

4=plus(2,3), $\leq$(max(2,3,4),7), (4=5) $\Rightarrow \Lambda$, Sein $\vee \neg$Sein, lange_währt $\Rightarrow$ endlich_gut,
$\forall$x:$\mathbb{N}$.$\exists$x:$\mathbb{Z}$. x=x, $\forall$x:$\mathbb{N}$.$\exists$y:$\mathbb{Z}$. $\leq$(*(y,y), x) $\wedge$ <(x, *(plus(y,1),plus(y,1)))

Man beachte daß die syntaktische Korrektheit einer Formel nichts mit ihrer Wahrheit zu tun hat, denn es muß ja auch möglich sein, falsche Aussagen zu *formulieren*. Deshalb ist 4 = 5 eine korrekte Formel, auch wenn wir damit keine wahre Aussage verbinden. Keine korrekten Formeln sind dagegen

plus(plus(2,3),4), $\wedge$ so_weiter, $\forall$x:$\mathbb{N}$. x(4)=x, $\forall$x. x=x.

Der erste Ausdruck ist ein Term, aber keine Formel, im zweiten Ausdruck fehlt eine Formel auf der linken Seite des Symbols $\wedge$ und im dritten Beispiel wird eine Variable unzulässigerweise als Funktionszeichen benutzt. Das vierte Beispiel ist eine Formel der unsortierten Prädikatenlogik, die aber im Sinne der Definition 2.2.4 syntaktisch nicht korrekt ist.

Die Verwendung von nullstelligen Funktions- und Prädikatssymbolen als Terme bzw. atomare Formeln, ohne daß hierzu Klammern verwendet werden müssen, entspricht der in Beispiel 2.1.1 auf Seite 11 angesprochenen *Konvention* zur Abkürzung formaler Ausdrücke. Die anderen dort angesprochenen Konventionen sind für eine syntaktische Korrektheit von Formeln eigentlich überflüssig. Ausdrücke wie

$\exists$y:$\mathbb{N}$. gerade(y) $\wedge$ $\geq$(y,2) $\Rightarrow$ y=2 $\wedge$ >(y,20)

sind syntaktisch absolut korrekt. Jedoch läßt sich die Frage, welche Aussagen sie beschreiben, nicht eindeutig beantworten, da unklar ist, was die wirkliche Struktur dieses Ausdrucks ist. Sie könnte gelesen werden als

$\exists$y:$\mathbb{N}$. (gerade(y) $\wedge$ $\geq$(y,2)) $\Rightarrow$ (y=2 $\wedge$ >(y,20)) oder als

$\exists$y:$\mathbb{N}$. (gerade(y) $\wedge$ ($\geq$(y,2) $\Rightarrow$ y=2)) $\wedge$ >(y,20) oder als

$\exists$y:$\mathbb{N}$. gerade(y) $\wedge$ ($\geq$(y,2) $\Rightarrow$ (y=2 $\wedge$ >(y,20))

Mit diesen drei Lesarten verbinden wir sehr unterschiedliche mathematische Aussagen. Die erste Aussage wird z.B. wahr, wenn wir eine ungerade Zahl für y einsetzen. Die zweite Aussage ist falsch, denn wir müssten eine gerade Zahl einsetzen, die größer als 20 ist, aber den Wert 2 annimmt, wenn sie größer oder gleich 2 ist. In der dritten können wir nur die Zahl 0 als Wert für y verwenden. Es wäre nun lästig, wenn wir die Eindeutigkeit einer Formalisierung nur durch die Verwendung von Klammern erreichen können. Aus diesem Grunde gibt es Konventionen, welche die Struktur ungeklammerter Ausdrücke eindeutig festlegen.[23]

Definition 2.2.6 (Konventionen zur Eindeutigkeit prädikatenlogischer Ausdrücke)

Für die Bindungskraft der logischen Operatoren gilt die folgende Reihenfolge:

$\neg$ *bindet stärker als* $\wedge$*, dann folgt* $\vee$*, dann* $\Rightarrow$*, dann* $\exists$ *und zum Schluß* $\forall$*.*

In einer Formel braucht eine durch einen stärker bindenden Operator gebildete Teilformel nicht geklammert zu werden. Bei gleicher Bindungskraft gilt (im Zweifel) Rechtsassoziativität.

Diese Vereinbarung erlaubt es uns, überflüssige Klammern fallen zu lassen. Die Vereinbarung der Rechtsassoziativität ist eigentlich nur für die Implikation von Bedeutung. Auch hier geben wir ein paar Beispiele.

Beispiel 2.2.7

A $\wedge$ $\neg$B	entspricht	A $\wedge$ ($\neg$B)
A $\wedge$ B $\vee$ C	entspricht	(A $\wedge$ B) $\vee$ C
A $\Rightarrow$ B $\vee$ C	entspricht	A $\Rightarrow$ (B $\vee$ C)
A $\Rightarrow$ B $\Rightarrow$ C	entspricht	A $\Rightarrow$ (B $\Rightarrow$ C)
$\exists$x:$\mathbb{N}$. A $\wedge$ B(x) $\vee$ C(x)	entspricht	($\exists$x:$\mathbb{N}$. (A $\wedge$ B(x)) $\vee$ C(x))

Bei der obigen Beispielformel entspricht die erste Lesart der Konvention aud Definition 2.2.6.

[23] Diese Konventionen sind besonders von großer Bedeutung, wenn man die formale Sprache mit dem Computer verarbeiten möchte und dennoch eine gewisse Freiheit bei der Formalisierung haben will. Zusammen mit den Definition 2.2.2 und 2.2.4 führen diese Konventionen zu einer *eindeutigen* Grammatik für die Prädikatenlogik, die sich von einem Parser leichter verarbeiten läßt.

2.2.2 Semantik

Obwohl es sich bei logischen Formeln eigentlich nur um reine Zeichenfolgen ohne jede Bedeutung handelt, assoziieren wir natürlich bestimmte Begriffe mit Konstanten wie `peter` oder `24` und Prädikaten wie $\leq$`(4,29)`. Um dieser gedachten Zuordnung zwischen formalen Ausdrücken und semantischen Inhalten Ausdruck zu verleihen, bedient man sich einer *Interpretationsfunktion*, die jedem Variablen-, Funktions-, Prädikats- und Bereichssymbol ein mengentheoretisches Objekt zuweist, und setzt diese dann auf Terme und logische Formeln fort. In mathematischen Begriffen definiert man die Semantik von Formeln dann wie folgt.

Definition 2.2.8 (Interpretation prädikatenlogischer Formeln)

1. *Eine Interpretation der Prädikatenlogik ist ein Paar $(\iota,\mathcal{U})$, wobei $\mathcal{U}$ eine Menge (das Universum) und ι eine Abbildung ist, für die gilt*
   - *ι weist jeder Variablen $x \in \mathcal{V}$ ein Objekt aus $\mathcal{U}$ zu.*
   - *ι weist jedem Funktionssymbol $f \in \mathcal{F}^n$ eine n-stellige Funktion $\phi : \mathcal{U}^n \to \mathcal{U}$ zu.*
   - *ι weist jedem Bereichssymbol $T \in \mathcal{T}$ eine Menge zu, und es gilt $\bigcup_{T \in \mathcal{T}} \iota(T) \subseteq \mathcal{U}$.*
   - *ι weist jedem Prädikatssymbol $P \in \mathcal{P}^n$ eine charakteristische Funktion $\Pi : \mathcal{U}^n \to \{\mathsf{wahr}, \mathsf{falsch}\}$ zu.*

2. *Eine Interpretationsfunktion ι wird auf prädikatenlogische Terme und Formeln gemäß der folgenden Regeln homomorph fortgesetzt.*
   - *Sind $t_1, \ldots, t_n$ Terme ($n \geq 1$) und ist $f \in \mathcal{F}^n$ ein beliebiges n-stelliges Funktionssymbol, dann ist $\iota(f(t_1,\ldots,t_n)) = \iota(f)(\iota(t_1),\ldots,\iota(t_n))$.*
   - $\iota(\Lambda) = \mathsf{falsch}$
   - *Sind t_1 und t_2 Terme, dann ist*
     $$\iota(t_1 = t_2) = \begin{cases} \mathsf{wahr} & \text{falls die Objekte } \iota(t_1) \text{ und } \iota(t_2) \text{ in } \mathcal{U} \text{ gleich sind} \\ \mathsf{falsch} & \text{sonst} \end{cases}$$
   - *Sind $t_1, \ldots, t_n$ Terme ($n \geq 1$) und ist $P \in \mathcal{P}^n$ ein beliebiges n-stelliges Prädikatssymbol, dann ist $\iota(P(t_1,\ldots,t_n)) = \iota(P)(\iota(t_1),\ldots,\iota(t_n))$.*
   - *Sind A und B Formeln, $x \in \mathcal{V}$ ein Variablensymbol und $T \in \mathcal{T}$ ein Bereichssymbol, dann ist*
     $$\iota(\neg A) = \begin{cases} \mathsf{wahr} & \text{falls } \iota(A) = \mathsf{falsch} \\ \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{wahr} \end{cases}$$
     $$\iota(A \wedge B) = \begin{cases} \mathsf{wahr} & \text{falls } \iota(A) = \mathsf{wahr} \text{ und } \iota(B) = \mathsf{wahr} \\ \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{falsch} \text{ oder } \iota(B) = \mathsf{falsch} \end{cases}$$
     $$\iota(A \vee B) = \begin{cases} \mathsf{wahr} & \text{falls } \iota(A) = \mathsf{wahr} \text{ oder } \iota(B) = \mathsf{wahr} \\ \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{falsch} \text{ und } \iota(B) = \mathsf{falsch} \end{cases}$$
     $$\iota(A \Rightarrow B) = \begin{cases} \mathsf{wahr} & \text{falls aus } \iota(A) = \mathsf{wahr} \text{ immer } \iota(B) = \mathsf{wahr} \text{ folgt} \\ \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{wahr} \text{ und } \iota(B) = \mathsf{falsch} \end{cases}$$
     $$\iota(\forall x{:}T\,.\,A) = \begin{cases} \mathsf{wahr} & \text{falls } \iota^u_x(A) = \mathsf{wahr} \text{ für alle } u \in \iota(T) \text{ ist} \\ \mathsf{falsch} & \text{falls } \iota^u_x(A) = \mathsf{falsch} \text{ für ein } u \in \iota(T) \text{ ist} \end{cases}$$
     $$\iota(\exists x{:}T\,.\,A) = \begin{cases} \mathsf{wahr} & \text{falls } \iota^u_x(A) = \mathsf{wahr} \text{ für ein } u \in \iota(T) \text{ ist} \\ \mathsf{falsch} & \text{falls } \iota^u_x(A) = \mathsf{falsch} \text{ für alle } u \in \iota(T) \text{ ist} \end{cases}$$
     $$\iota((A)) = \iota(A)$$
     Dabei bezeichnet ι^u_x diejenige Interpretationsfunktion, die identisch mit ι für alle von der Variablen x verschiedenen Symbole ist und für die $\iota^u_x(x)$ der Wert u ist.[24]

Wir sehen also, daß durch die Interpretation im Grunde nur von der Symbolwelt in eine andere mathematische Denkwelt, nämlich die Mengentheorie, abgebildet wird. Der Unterschied ist lediglich, daß wir von der Mengentheorie eine relativ klare Vorstellung haben, die nicht weiter erklärt werden muß. Die Interpretation versetzt uns also in die Lage, den Wahrheitsgehalt einer Formel genau zu beurteilen. Zu beachten ist dabei, daß nullstellige Funktionen mit Elementen ihres Bildbereiches identifiziert werden. Nullstelligen Funktionssymbolen weist eine Interpretation also ein konstantes Objekt aus $\mathcal{U}$ zu und nullstelligen Prädikatssymbolen einen Wahrheitswert aus {wahr,falsch}.

Das folgende Beispiel zeigt, wie man die Bedeutung eines Ausdrucks für eine vorgegebene Interpretation "ausrechnen" kann.

Beispiel 2.2.9

Die Interpretationsfunktion ι interpretiere Zahlsymbole in Dezimaldarstellung in der bekannten Weise, d.h. ι(2) ist die Zahl *zwei*, ι(12) ist die Zahl *zwölf* usw. ι(max) sei die Funktion ϕ_{max}, welche das Maximum dreier Zahlen bestimmt. $\iota(\leq)$ sei die charakteristische Funktion $\Pi_{\leq}$ der "*kleiner-gleich*"-Relation, also genau dann wahr, wenn das erste Argument nicht größer als das zweite ist. $\iota(\mathbb{N})$ sei die Menge der natürlichen Zahlen.[25] Dann wird der Ausdruck $\leq$(max(2,3,4),7) wie folgt interpretiert

$$\begin{aligned}
& \iota(\leq(\texttt{max}(2,3,4),7)) \\
= \;& \iota(\leq)(\iota(\texttt{max}(2,3,4)),\iota(7)) \\
= \;& \Pi_{\leq}(\iota(\texttt{max})(\iota(2),\iota(3),\iota(4)),\ \textit{sieben})) \\
= \;& \Pi_{\leq}(\phi_{max}(\textit{zwei},\textit{drei},\textit{vier}),\ \textit{sieben}) \\
= \;& \Pi_{\leq}(\textit{vier},\ \textit{sieben}) \\
= \;& \mathsf{wahr}
\end{aligned}$$

Sei zusätzlich – der Vollständigkeit halber – ι(x) die Zahl *dreizehn*. Dann wird der prädikatenlogische Ausdruck $\exists$x:$\mathbb{N}$. $\leq$(max(2,3,4),x) wie folgt interpretiert:

$$\begin{aligned}
& \iota(\exists \texttt{x}{:}\mathbb{N}.\ \leq(\texttt{max}(2,3,4),\texttt{x})) \\
= \;& \mathsf{wahr} \text{ genau dann, wenn } \iota^u_x(\leq(\texttt{max}(2,3,4),\texttt{x})) = \mathsf{wahr} \text{ für ein } u \in \iota(\mathbb{N}) \text{ ist} \\
= \;& \vdots \\
= \;& \mathsf{wahr} \text{ genau dann, wenn } \Pi_{\leq}(\textit{vier},\iota^u_x(\mathrm{x})) = \mathsf{wahr} \text{ für eine natürliche Zahl } u \text{ ist.} \\
= \;& \mathsf{wahr} \text{ genau dann, wenn } \Pi_{\leq}(\textit{vier},u) = \mathsf{wahr} \text{ für eine natürliche Zahl } u \text{ ist.} \\
= \;& \mathsf{wahr} \qquad\qquad (\text{Wir wählen } u \text{ als } \textit{fünf})
\end{aligned}$$

Einen Sonderfall wollen wir hier noch ansprechen. Es ist durchaus legitim, ein und dieselbe Variable zweimal innerhalb von Quantoren zu verwenden. Der Ausdruck

$$\forall \texttt{x}{:}\mathbb{N}.\ \geq(\texttt{x},0)\ \wedge\ \exists \texttt{x}{:}\mathbb{Z}.\ \texttt{x=x}$$

ist eine absolut korrekte, wenn auch ungewöhnliche Formel der Prädikatenlogik. Damit muß es auch möglich sein, dieser eine präzise Bedeutung zuzuordnen. Die Intuition sagt uns, daß der zuletzt genannte Quantor in einer Formel Gültigkeit haben muß, daß also das x in x=x sich auf den Existenzquantor bezieht. Der Gedanke dabei ist, daß es bei quantifizierten Formeln auf den konkreten Namen einer Variablen eigentlich nicht ankommen darf, sondern daß diese nur ein Platzhalter dafür ist, daß an ihrer Stelle etwas beliebiges eingesetzt werden darf. Die Formel $\exists$x:$\mathbb{Z}$. x=x muß also dasselbe bedeuten wie $\exists$y:$\mathbb{Z}$. y=y und damit muß obige Formel die gleiche Bedeutung haben wie

$$\forall \texttt{x}{:}\mathbb{N}.\ \geq(\texttt{x},0)\ \wedge\ \exists \texttt{y}{:}\mathbb{Z}.\ \texttt{y=y}$$

Unsere Definition der Semantik wird dieser Forderung gerecht, denn sie besagt, daß *innerhalb* einer quantifizierten Formel eine Interpretation ι bezüglich der quantifizierten Variable x geändert werden darf. Welcher Wert außerhalb für x gewählt wurde, hat innerhalb also keine Gültigkeit mehr.

[24] Durch die Forderung, daß $\iota^u_x(A)$ den Wert wahr für alle bzw. ein $u \in \iota(T)$ haben muß, drücken wir aus, daß die Interpretation der Formel A wahr sein muß für alle Werte (bzw. einen Wert), die wir für (die Interpretation von) x einsetzen können.

[25] Es sei vereinbart, daß die natürlichen Zahlen die Null enthalten.

In Definition 2.2.8 haben wir die Semantik der logischen Symbole durch natürlichsprachliche Formulierungen erklärt, über deren Bedeutung es klare Vorstellungen zu geben scheint. Leider gehen im Falle der Disjunktion $\vee$, der Implikation $\Rightarrow$ und des Existenzquantors $\exists$ die Meinungen darüber auseinander, was die Worte "*oder*", "*folgt*" und "*es gibt ein...*" nun genau bedeuten. Viele Mathematiker würden die folgenden Charakterisierungen als gleichbedeutend zu denen aus Definition 2.2.8 ansehen:

$$\iota(A \vee B) = \begin{cases} \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{falsch} \text{ und } \iota(B) = \mathsf{falsch} \\ \mathsf{wahr} & \text{sonst} \end{cases}$$

$$\iota(A \Rightarrow B) = \begin{cases} \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{wahr} \text{ und } \iota(B) = \mathsf{falsch} \\ \mathsf{wahr} & \text{sonst} \end{cases}$$

$$\iota(\exists x{:}T\,.\,A) = \begin{cases} \mathsf{falsch} & \text{falls } \iota^u_x(A) = \mathsf{falsch} \text{ für } \underline{\text{alle}}\ u \in \iota(T) \text{ ist} \\ \mathsf{wahr} & \text{sonst} \end{cases}$$

Dahinter verbirgt sich die Vorstellung, daß *jede* mathematische Aussage einen eindeutig bestimmten Wahrheitsgehalt haben *muß*. Damit sind bei der Disjunktion und der Implikation jeweils nur vier Möglichkeiten zu betrachten und dies führt dann dazu, daß die obigen Charakterisierungen wirklich äquivalent zu den in Definition 2.2.8 gegebenen werden. Auch stimmt es, daß die Interpretation einer Formel $\exists x{:}T\,.\,A$ mit Sicherheit genau dann falsch ist, wenn für $\underline{\text{alle}}\ u \in \iota(T)$ $\iota^u_x(A) = \mathsf{falsch}$ ist.

Im Endeffekt ist aber die Annahme, daß eine Aussage wahr ist, wenn sie nicht falsch ist, ein *Postulat* – also eine Forderung, die sich nicht beweisen läßt. Eine Möglichkeit, andere von der Richtigkeit oder Falschheit dieses Postulats zu überzeugen, gibt es nicht. Ob man die obengemachte Annahme akzeptiert oder ablehnt, ist also eher eine Frage der "Weltanschauung" und weniger die Frage danach, was denn nun wahr ist.[26] Da sich im formalen Kalkül der Unterschied zwischen diesen Weltanschauungen durch Hinzunahme oder Entfernung einer einzigen Regel ausdrücken läßt, werden wir die klassische und die intuitionistische Prädikatenlogik im folgenden simultan behandeln.

Es sei angemerkt, daß die klassische Prädikatenlogik gemäß der oben angegebenen äquivalenten Charakterisierungen eigentlich auf Disjunktion, Implikation und Existenzquantor verzichten könnte, da sich diese durch $\neg$, $\wedge$ und $\forall$ simulieren lassen und somit als definitorische Erweiterung im Sinne von Abschnitt 2.1.5 (Seite 14) angesehen werden können.

Wir wollen nochmals darauf hinweisen, daß – mit Ausnahme der Gleichheit – alle Prädikats- und Funktionssymbole (insbesondere die vertrauten Symbole aus der Arithmetik) keine feste Bedeutung besitzen. Die Bedeutung dieser Symbole hängt stattdessen von der konkreten Interpretation ab und wird bei der Frage nach der Gültigkeit logischer Gesetze und ihrer Darstellung als Inferenzregeln keine Rolle spielen. In der Prädikatenlogik kann man nur strukturelle Informationen verarbeiten, aber nicht rechnen.

[26] In der Tat hat es über diese Frage seit Anfang dieses Jahrhunderts unter den Mathematikern Streitigkeiten gegeben, die schon fast religiöse Züge angenommen haben, da sowohl die klassische als auch die intuitionistische Mathematik für sich in Anspruch nahmen, die *reine Wahrheit zu verkünden* und die Vertreter der jeweils anderen Denkrichtung als Ketzer ansahen.

Für die *klassische* – Ihnen aus der Schule vertrautere – Mathematik ist die Wahrheit einer Aussage dann gesichert, wenn man weiß, daß sie nicht falsch ist. Insbesondere werden Objekte mit einer bestimmten Eigenschaft als existent angesehen, wenn man weiß, daß nicht alle Objekte diese Eigenschaft nicht besitzen. Das strapaziert zuweilen zwar die Anschauung, macht aber mathematische Beweise so viel einfacher, daß sich die klassische Mathematik weitestgehend durchgesetzt hat.

Die *intuitionistische* Mathematik dagegen geht davon aus, daß nur die natürlichen Zahlen – wie der Name schon sagt – etwas Naturgegebenes sind, von dem jeder Mensch ein Verständnis besitzt. Alle anderen mathematischen Objekte dagegen – also ganze Zahlen, Brüche, reelle und komplexe Zahlen usw. – sind reine Hilfskonstruktionen des Menschen, deren einzige Aufgabe es ist *bestimmte natürliche Phänomene eleganter zu beschreiben*. Es sind also abkürzende *Definitionen* oder – wie man sagt – *mentale Konstruktionen*. Die Frage nach der Existenz von abstrakten Objekten, die sich nicht einmal gedanklich konstruieren lassen, ist in diesem Sinne reine Spekulation, deren Beantwortung etwas "metaphysisches" beinhaltet. Religion aber – so sagen die Intuitionisten – ist Privatsache des einzelnen Menschen und hat mit Mathematik nichts zu tun.

Beide Argumente sind an und für sich einsichtig, vertragen einander aber nicht. Für das logische Schließen über Algorithmen, deren Natur ja gerade die Konstruktion ist, scheint jedoch die intuitionistische angemessener zu sein, auch wenn wir dadurch auf einen Teil der Eleganz der klassischen Mathematik verzichten. Im Zusammenhang mit der Besprechung der Typentheorie in Kapitel 3 werden wir allerdings auch zeigen, wie wir beide Konzepte in einem Formalismus gleichzeitig behandeln können. Wir können in der intuitionistischen Typentheorie klassisch schließen, müssen dies aber explizit sagen.

2.2.3 Analytische Sequenzenbeweise

Formale Kalküle sollen, wie in Abschnitt 2.1.6 erwähnt, durch syntaktische Manipulation logisch korrekte Schlußfolgerungen simulieren. Ein Kalkül für die Prädikatenlogik muß also Formeln so in andere Formeln umwandeln, daß der Semantik aus Definition 2.2.8 Rechnung getragen wird. Dabei sollte der Kalkül natürlich unabhängig von der konkreten Interpretation sein, denn er muß ja alleine auf Formeln operieren.

In diesem Sinne sind logisch korrekte Schlüsse solche, die wahre Aussagen wieder in wahre Aussagen überführen. Für den Kalkül bedeutet dies, daß Formeln, die für jede Interpretation wahr sind, wieder in "allgemeingültige" Formeln umgewandelt werden müssen. Um dies zu präzisieren, wollen wir ein paar Begriffe prägen.

Definition 2.2.10 (Modelle und Gültigkeit)

Es sei A eine beliebige prädikatenlogische Formel.

1. *Eine Interpretation $(\iota,\mathcal{U})$ mit $\iota(A) =$ wahr heißt Modell von A*
2. *A heißt*
   - *erfüllbar, wenn es ein Modell für A gibt,*
   - *widerlegbar, wenn es ein Modell für $\neg A$ gibt,*
   - *widersprüchlich oder unerfüllbar, wenn es für A kein Modell gibt,*
   - *(allgemein)gültig, wenn jede Interpretation ein Modell für A ist.*

Wir wollen diese Begriffe an einigen Beispielen erläutern.

Beispiel 2.2.11

1. Die Formel $(\leq(4,+(3,1)) \Rightarrow \leq(+(3,1),4)) \Rightarrow \leq(+(3,1),4)$ ist erfüllbar aber nicht gültig.

   Interpretieren wir nämlich $\leq$ durch die übliche "kleiner-gleich" Relation, + durch die Addition und die Zahlsymbole wie gewöhnlich, dann erhalten wir eine wahre Aussage. Interpretieren wir dagegen $\leq$ durch die Relation "kleiner", so wird die Aussage falsch. Denn die innere Implikation wird wahr (aus $4 < 4$ folgt $4 < 4$), aber die rechte Seite der äußeren Implikation ($4 < 4$) ist falsch.

2. Die Formel $\leq(4,+(3,1)) \wedge \neg\leq(4,+(3,1))$ ist unerfüllbar, da die beiden Aussagen der Konjunktion einander widersprechen und nie gleichzeitig wahr sein können.

3. Die Formel $(\leq(4,+(3,1)) \wedge \leq(+(3,1),4)) \Rightarrow \leq(+(3,1),4)$ ist allgemeingültig.

   Nach der Semantik der Konjunktion in Definition 2.2.8 gilt nämlich für jede Interpretation ι, daß $\iota(\leq(+(3,1),4)) =$ wahr ist, wenn $\iota(\leq(4,+(3,1)) \wedge \leq(+(3,1),4)) =$ wahr ist. Damit ist auch die Voraussetzung für die Wahrheit der gesamten Implikation erfüllt.

Gültige Formeln spielen bei der Simulation logisch korrekter Folgerungen eine Schlüsselrolle. Beim Entwurf der Inferenzregeln eines Kalküls muß man sicherstellen, daß aus gültigen Formeln wieder gültige Formeln hergeleitet werden. Dieser Aspekt wird bei der Begründung der Regeln von besonderer Bedeutung sein.

Wir wollen im folgenden nun einen analytischen Sequenzenkalkül (vgl. Abschnitt 2.1.6.3, Seite 20 ff) für die Prädikatenlogik entwickeln. Dabei werden wir auf eine computergerechte Formulierung der Inferenzregeln achten, damit diese Regeln auch in einem Beweiseditor wie dem NuPRL System Verwendung finden können. Die wesentliche Änderung betrifft dabei die Menge Γ der Annahmen in einer Sequenz $\Gamma \vdash C$. Sie wird nun als Liste dargestellt und erlaubt somit, auf einzelne Annahmen über deren Position in der Liste zuzugreifen. Wir wollen als erstes den Begriff der Sequenzen und Beweise definieren.

Definition 2.2.12 (Sequenzen, Regeln und Beweise)

1. *Eine Deklaration hat die Gestalt* $x{:}T$, *wobei* $x \in \mathcal{V}$ *eine Variable und* $T \in \mathcal{T}$ *ein Bereichsbezeichner ist.*

2. *Eine Sequenz hat die Gestalt* $\Gamma \vdash C$, *wobei* Γ – *die Hypothesenliste* – *eine Liste von durch Komma getrennten Formeln oder Deklarationen und* C – *die Konklusion* – *eine Formel ist.*

3. *Eine (analytische) Inferenzregel ist eine Abbildung, welche eine Sequenz – das Beweisziel – in eine endliche Liste von Sequenzen – die Unter- oder Teilziele – abbildet.*

4. *Ein Beweis ist ein Baum, dessen Knoten eine Sequenz und eine Regel enthalten. Dabei müssen die Sequenzen der Nachfolger eines Knotens genau die Teilziele sein, welche sich durch Anwendung der Regel auf die Knotensequenz ergeben.*

   In einem unvollständigen Beweis darf es vorkommen, daß die Blätter nur eine Sequenz, aber keine Regel enthalten. Ein Beweis ist vollständig, wenn seine Blätter Regeln enthalten, welche bei Anwendung auf die zugehörigekn Sequenzen keine neuen Teilziele erzeugen.

5. *Eine Formel* C *ist ein Theorem, wenn es einen vollständigen Beweis gibt, dessen Wurzel die Sequenz* '$\vdash C$' – *also eine Sequenz ohne Hypothesen – enthält.*

Semantisch besehen ist eine Sequenz $A_1, \ldots, A_n \vdash C$ keine Formel, sondern ein *Urteil* über eine logische Folgerung. Üblicherweise liest man diese Sequenz als

Aus den Annahmen $A_1, \ldots, A_n$ *folgt die Konklusion* C.

Die folgende Definition der Semantik einer Sequenz entspricht dieser Lesart.

Definition 2.2.13 (Interpretation von Sequenzen)

Eine Interpretationsfunktion ι *wird auf Sequenzen wie folgt fortgesetzt*

$$\iota(A_1, \ldots, A_n \vdash C) = \begin{cases} \mathsf{wahr} & \textit{falls aus } \iota(A_1) = \mathsf{wahr} \textit{ und } \ldots \iota(A_n) = \mathsf{wahr} \textit{ immer } \iota(C) = \mathsf{wahr} \textit{ folgt} \\ \mathsf{falsch} & \textit{sonst} \end{cases}$$

Dabei gilt – der Einfachheit halber – $\iota(x{:}T) = \mathsf{wahr}$ *für jede Deklaration* $x{:}T$.[27]

Zu beachten ist, daß diese Definition der Rolle einer Sequenz als Formalisierung von logischen Urteilen nicht ganz gerecht wird, da genaugenommen ein Wahrheitsbegriff auf Urteile nicht anwendbar ist. Wir haben in Definition 2.2.13 implizit das sogenannte *Deduktionstheorem* angewandt, welches besagt, daß ein logischer Schluß von den Annahmen $A_1, \ldots, A_n$ auf eine Konklusion C genau dann korrekt ist, wenn die Formel $A_1 \wedge \ldots \wedge A_n \Rightarrow C$ gültig ist. Syntaktisch betrachtet spielt innerhalb eines Kalküls der Unterschied zwischen einem Urteil und einer Implikation keine besondere Rolle.

Inferenzregeln werden üblicherweise nicht als Funktionen beschrieben, sondern als Regel*schemata*, welche die Syntax des Beweiszieles und der entstehenden Teilziele mit Hilfe syntaktischer Metavariablen angeben. Die Regel ist auf eine gegebene Sequenz anwendbar, wenn sich dieses aus dem schematischen Beweisziel durch Einsetzen konkreter Formeln anstelle der syntaktischen Metavariablen ergibt. Die neuen Unterziele ergeben sich entsprechend aus den schematischen Teilzielen. Neben einigen strukturellen Regeln – wie zum Beispiel der Regel, daß eine Konklusion C gilt, wenn sie in der Hypothesenliste auftaucht – gibt es im Sequenzenkalkül für jedes logische Symbol zwei Arten von Regelschemata.

[27] Deklarationen haben im Rahmen der Prädikatenlogik vorerst keine semantische Bedeutung. Diese wird erst im Zusammenhang mit der Diskussion der Typentheorie, beginnend ab Abschnitt 2.4 deutlich werden. Bis dahin sind sie nicht mehr als ein syntaktischer Kontrollmechanismus für die Namensgebung von Variablen.

- Einführungsregeln beantworten die Frage "*was muß man zeigen, um eine Konklusion C zu beweisen?*" und haben üblicherweise die Gestalt

  $\Gamma \vdash C$ **by** *symbolname*_i *parameter*
  $\Gamma, \ldots \vdash C_1$
  $\Gamma, \vdots \ldots \vdash C_n$

  *symbolname*_i ist der Name der Regel. Er ist zum Beispiel anzugeben, wenn man die Regel mit Hilfe eines Beweiseditors ausführen möchte. *parameter* beschreibt eine (meist leere) Liste von Kontrollparametern, die in manchen Fällen – wie bei dem Existenzquantor – erforderlich sind, um die Regel ausführen zu können. $\Gamma \vdash C$ ist das Beweisziel und darunter stehen eingerückt die sich ergebenden Teilziele. In manchen Fällen – wie bei der Implikation – werden dabei Annahmen in die Hypothesenliste verlagert.

- Eliminationsregeln beantwortet die Frage "*welche Annahmen folgen aus einer gegebenen Formel A in der Hypothesenliste?*" und haben üblicherweise die Gestalt

  $\Gamma, A, \Delta \vdash C$ **by** *symbolname*_e i *parameter*
  $\Gamma, \ldots, \Delta \vdash C_1$
  $\Gamma, \vdots \ldots, \Delta \vdash C_n$

  Dabei ist *symbolname*_e wieder der Name der auszuführenden Regel, ggf. gesteuert durch einige Parameter. Da es sich bei den Hypothesen um Listen handelt, können wir die zu eliminierende Formel durch die Angabe ihrer Position i in dieser Liste bestimmen.[28]

Analytische Einführungsregeln werden also dazu benutzt, um die Konklusion in Teilformeln zu zerlegen während man mit Eliminationsregeln eine bestehende Annahme zerlegt, um neue Annahmen zu generieren.

2.2.4 Strukturelle Regeln

Zusätzlich zu den sogenannten "logischen" Regeln, welche die Semantik einzelner logischer Symbole charakterisieren, benötigt der Sequenzenkalkül eine Regel, welche den Zusammenhang zwischen den Hypothesen einer Sequenz und ihrer Konklusion beschreibt. Diese Regel besagt, daß eine Sequenz $\Gamma \vdash C$ gültig ist, wann immer die Konklusion C in den Annahmen auftaucht. Diese Regel, deren Rechtfertigung sich aus dem oben erwähnten Deduktionstheorem ableitet, ist praktisch das Axiom des Sequenzenkalküls. Formal drückt man dies so aus, daß die Hypothesenliste sich aufspalten läßt in drei Teile Γ, C, Δ, wobei Γ und Δ beliebig lange (auch leere!) Hypothesenlisten sind. Ist C nun die i-te Hypothese in der gesamten Liste (d.h. Γ hat i-1 Elemente) und die Konklusion der Sequenz, dann können wir sagen, daß die Konklusion aus der i-ten Hypothese folgt. Als formale Inferenzregel schreibt sich dies wie folgt:

$\Gamma, C, \Delta \vdash C$ **by** `hypothesis` i

Die *Hypothesenregel* `hypothesis` ist die einzige strukturelle Regel, die für den Sequenzenkalkül unbedingt erforderlich ist. Auf die folgenden beiden Regeln – *Schnitt* (`cut`) und *Ausdünnung* (`thin`) – könnte man im Prinzip auch verzichten. Sie tragen jedoch dazu bei, Beweise besser zu strukturieren und durch Entfernen nicht mehr benötigter Annahmen lesbarer zu machen.

Die *Schnittregel* besagt, daß man Beweise durch Einfügen von Zwischenbehauptungen zerlegen darf. Dies entspricht der Methode, mathematische Beweise durch Lemmata zu strukturieren und übersichtlicher zu gestalten. Man stellt dazu eine Formel A auf, beweist deren Gültigkeit und darf diese dann innerhalb der Annahmen zum Beweis des eigentlichen Beweiszieles weiterverwenden:

$\Gamma, \Delta \vdash C$ **by** `cut` i A
$\Gamma, \Delta \vdash A$
$\Gamma, A, \Delta \vdash C$

Dabei ist i die neue Position der Formel A in der Hypothesenliste.

[28] Deklarationen können im Rahmen der Prädikatenlogik nicht eliminiert werden. Sie werden nur benötigt, um die Variablennamen in Hypothesen und Konklusion zu verwalten und tauchen entsprechend nur über Einführung bei Allquantoren und Elimination von Existenzquantoren auf, wo sie die sogenannte "*Eigenvariablenbedingung*" von Genzten [Gentzen, 1935] ersetzen.

Die *Ausdünnungsregel* ermöglicht es, überflüssige Annahmen aus der Hypothesenliste zu entfernen. Dies ist besonders dann sinnvoll, wenn die Liste der Annahmen in großen Beweisen unüberschaubar zu werden droht und das zu beweisende Teilziel mit manchen Hypothesen gar nichts zu tun hat. Zum Ausdünnen gibt man einfach die Position i der zu entfernende Annahme an.[29]

$\Gamma, A, \Delta \vdash C$ **by** `thin` i
$\Gamma, \Delta \vdash C$

Die Ausdünnungsregel ist jedoch mit Sorgfalt einzusetzen. Unterziele könnten unbeweisbar werden, wenn man unvorsichtigerweise Annahmen herausnimmt, die doch noch relevant sind.

2.2.5 Aussagenlogische Regeln

Konjunktion

Eine Konjunktion $A \wedge B$ ist genau dann wahr, wenn A und B wahr sind. Um also $A \wedge B$ zu beweisen, müssen wir A und B separat zeigen können. Die entsprechende Regel des Sequenzenkalküls lautet:

$\Gamma \vdash A \wedge B$ **by** `and_i`
$\Gamma \vdash A$
$\Gamma \vdash B$

Die Anwendung der Einführungsregel `and_i` für die Konjunktion auf die Sequenz $\Gamma \vdash A \wedge B$ läßt uns also als noch zu beweisende Unterziele zwei Sequenzen, nämlich $\Gamma \vdash A$ und $\Gamma \vdash B$.

Umgekehrt können wir aus einer Annahme $A \wedge B$ folgern, daß sowohl A als auch B gültige Annahmen sind. Wenn i die Position der Formel $A \wedge B$ in der Hypothesenliste kennzeichnet, dann können wir diese Schlußfolgerung durch Ausführung der Eliminationsregel "`and_e` i" simulieren.

$\Gamma, A \wedge B, \Delta \vdash C$ **by** `and_e` i
$\Gamma, A, B, \Delta \vdash C$

Beispiel 2.2.14

Durch Anwendung der Regeln für die Konjunktion läßt sich sehr leicht zeigen, daß die Konjunktion kommutativ ist. Zu beweisen ist hierbei die Sequenz

$$A \wedge B \vdash B \wedge A$$

wobei A und B Platzhalter für beliebige Formeln sind (d.h. das Beweismuster ist für alle Konjunktionen gleich[30]). Im ersten Schritt eliminieren wir die Annahme $A \wedge B$, welche die erste und einzige Formel in unserer Hypothesenliste ist. Nach Anwendung der Regel `and_e 1` erhalten wir

$$A,\ B \vdash B \wedge A$$

Nun zerlegen wir das Beweisziel durch die Einführungsregel `and_i` für die Konjunktion und erhalten zwei Teilziele

$$A,\ B \vdash B \quad \text{und} \quad A,\ B \vdash A$$

Beide Teilziele lassen sich durch Anwendung von `hypothesis 2` bzw. `hypothesis 1` vollständig beweisen. Der Beweis, den wir in Abbildung 2.4[31] zusammenfassend dargestellt haben, ist damit abgeschlossen.

[29] Deklarationen von Variablen, die noch verwendet werden, können nicht ausgedünnt werden. Die entsprechende Regel des NuPRL Systems würde bei einem derartigen Versuch eine Fehlermeldung auslösen.

[30] Wir könnten daher das Kommutativgesetz als eine "kombinierte" Regel hinzufügen, die im Endeffekt dasselbe tut, wie die hier vorgestellten vier Beweisschritte. Mit dem Taktik-Konzept, das wir aber erst in Kapitel 4 besprechen werden, können wir derartige Ergänzungen sogar als konservative Erweiterung des Regelsystems durchführen.

[31] Jede Zeile repräsentiert einen Bestandteil eines Knotens im Beweisbaum, also die Sequenz oder die Beweisregel. Das Resultat der Anwendung einer Beweisregel erscheint eingerückt unter dieser Regel und ist mit ihr graphisch verbunden. Das Maß der Einrückung kennzeichnet auch die Tiefe des Knotens im Beweisbaum.

```
A∧B  ⊢  B∧A
by and_e 1
 \
  A, B    ⊢  B∧A
  by and_i
  |\
  | A, B ⊢  B
  | by hypothesis 2
  |  \
  |   A, B ⊢  A
  |   by hypothesis 1
```

Abbildung 2.4: Sequenzenbeweis für die Kommutativität der Konjunktion

Disjunktion

Eine Disjunktion $A \vee B$ ist wahr, wenn A oder B wahr sind. Um also $A \vee B$ zu beweisen, muß man entweder A beweisen oder B beweisen können. Die Entscheidung, welches der beiden Ziele gezeigt werden soll, muß durch die Wahl einer der beiden Einführungsregeln getroffen werden.

$$\frac{\Gamma \vdash A \vee B \quad \textbf{by } \texttt{or\_i1}}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \vee B \quad \textbf{by } \texttt{or\_i2}}{\Gamma \vdash B}$$

Umgekehrt folgt eine Konklusion C aus $A \vee B$ wenn sie aus A folgt und aus B folgt. Die Elimination einer Disjunktion entspricht also einer Fallunterscheidung.

$$\frac{\Gamma, A \vee B, \Delta \vdash C \quad \textbf{by } \texttt{or\_e } i}{\Gamma, A, \Delta \vdash C \qquad \Gamma, B, \Delta \vdash C}$$

Beispiel 2.2.15

Hier ist ein Beweis für ein Distributivgesetz zwischen Konjunktion und Disjunktion:

```
A ∧ (B∨C) ⊢ (A∧B) ∨ (A∧C)
by and_e 1
 \
  A, (B∨C) ⊢ (A∧B) ∨ (A∧C)
  by or_e 2
  |\
  | A, B ⊢ (A∧B) ∨ (A∧C)
  | by or_i1
  |  \
  |   A, B ⊢ A∧B
  |   by and_i
  |   |\
  |   | A, B ⊢ A
  |   | by hypothesis 1
  |   |  \
  |   |   A, B ⊢ B
  |   |   by hypothesis 2
  |  \
  |   A, C ⊢ (A∧B) ∨ (A∧C)
  |   by or_i2
  |    \
  |     A, C ⊢ A∧C
  |     by and_i
  |     |\
  |     | A, C ⊢ A
  |     | by hypothesis 1
  |     |  \
  |     |   A, C ⊢ C
  |     |   by hypothesis 2
```

Implikation

Eine Implikation $A \Rightarrow B$ ist wahr, wenn B aus der Annahme A folgt. Um also $A \Rightarrow B$ zu beweisen, muß man A in die Annahmen verlagern und auf dieser Basis dann B zeigen. Umgekehrt folgt eine Konklusion C aus $A \Rightarrow B$, wenn sie aus B folgt und man A beweisen kann (man beachte, daß dies keine "genau dann, wenn" Beziehung ist).

$$\Gamma \vdash A \Rightarrow B \quad \textbf{by } \texttt{imp\_i}$$
$$\Gamma, A \vdash B$$

$$\Gamma, A \Rightarrow B, \Delta \vdash C \quad \textbf{by } \texttt{imp\_e } i$$
$$\Gamma, A \Rightarrow B, \Delta \vdash A$$
$$\Gamma, \Delta, B \vdash C$$

Die Eliminationsregel entspricht dem modus ponens "*aus A und $A \Rightarrow B$ folgt B*", ist aber kaum noch als dieser wiederzuerkennen. Liest man sie aber wie folgt

Wenn A unter der Annahme $A \Rightarrow B$ sowie C unter der Annahme B folgt und $A \Rightarrow B$ gilt, dann folgt C.

so wird klar, warum sie korrekt ist. Die Annahme $A \Rightarrow B$ muß im ersten Unterziel erhalten bleiben, da sie möglicherweise im Beweis noch benötigt wird.[32] Im zweiten Teilziel ist sie dagegen redundant und kann daher fallengelassen werden.

Beispiel 2.2.16

Hier ist ein Beweis für einen wichtigen Zusammenhang zwischen Implikation und Konjunktion:

$A \wedge B \Rightarrow C \vdash A \Rightarrow B \Rightarrow C$
by `imp_i`
\
$A \wedge B \Rightarrow C,\ A \vdash B \Rightarrow C$
by `imp_i`
\
$A \wedge B \Rightarrow C,\ A,\ B \vdash C$
by `imp_e 1`
|\
| $A \wedge B \Rightarrow C,\ A,\ B \vdash A \wedge B$
| **by** `and_i`
| |\
| | $A \wedge B \Rightarrow C,\ A,\ B \vdash A$
| | **by** `hypothesis 2`
| \
| $A \wedge B \Rightarrow C,\ A,\ B \vdash B$
| **by** `hypothesis 3`
\
$A,\ B,\ C \vdash C$
by `hypothesis 3`

Man beachte hierbei die Prioritätsregeln ungeklammerter Ausdrücke: $\wedge$ bindet stärker als $\Rightarrow$ und die Implikation ist rechtsassoziativ zu lesen.

Negation

Eine Negation $\neg A$ ist wahr, wenn A falsch ist. Was könnte einfacher und zugleich problematischer sein als Falschheit? In Anlehnung an Prawitz [Prawitz, 1965] stellen wir Falschheit als eine separate atomare Formel Λ dar und betrachten $\neg A$ als Abkürzung für $A \Rightarrow \Lambda$. Das Symbol Λ drückt den Gedanken des *Widerspruchs* aus und wird separat diskutiert. Diese Vorgehensweise hat den Vorteil, daß sie sowohl für klassische als auch für intuitionistische Logik eine einheitliche korrekte Sichtweise der Negation ermöglicht. Die Regeln der Negation ergeben sich unmittelbar aus denen der Implikation.

$$\Gamma \vdash \neg A \quad \textbf{by } \texttt{not\_i}$$
$$\Gamma, A \vdash \Lambda$$

$$\Gamma, \neg A, \Delta \vdash C \quad \textbf{by } \texttt{not\_e } i$$
$$\Gamma, \neg A, \Delta \vdash A$$
$$\Gamma, \Delta, \Lambda \vdash C$$

[32] Unter den Annahmen könnte sich z.B. bereits die Annahme $(A \Rightarrow B) \Rightarrow A$ befinden. Dann folgt A aus dieser Annahme und der verbliebenen Implikation.

Falschheit (Widerspruch)

Das Symbol Λ, dessen Semantik grundsätzlich den Wert falsch erhält, soll den logischen Gedanken des Widerspruchs ausdrücken. Um Λ zu beweisen, müssen wir einen Widerspruch herleiten können. Im Rahmen der Prädikatenlogik gibt es hierfür keine spezielle Einführungsregel `false_i`, da ein Widerspruch eben nur aus sich widersprechenden Annahmen – wie A und $\neg A$ (oder später auch arithmetischen Widersprüchen) – entstehen kann, niemals aber unabhängig davon. Anders sieht es aus mit der Verwendung von Widersprüchen. In der Arithmetik ist die Aussage `0=1` ein typischer Vertreter einer widersprüchlichen Aussage. Jeder Widerspruch kann im Endeffekt in Form dieser Aussage formuliert werden. Hat man eine derartige Aussage nun bewiesen, so folgt hieraus jede arithmetische Aussage. Alle Zahlen werden gleich und jede Aussage, die für eine einzige Zahl gültig ist, gilt nun für alle Zahlen. Aus diesem Grunde macht es Sinn zu fordern, daß aus einer falschen Annahme jede beliebige Aussage folgt: "*Ex falso quodlibet*".

$\Gamma,\Lambda,\Delta \vdash C$ **by** `false_e` i

Ist also Λ eine der Annahmen einer Sequenz, so ist die Sequenz gültig – unabhängig davon, wie ihre Konklusion konkret aussieht. Diese Regel gilt klassisch und intuitionistisch.

In einer alternativen Lesart läßt sich die Regel `false_e` so verstehen, daß sich die Sequenz $\Gamma \vdash C$ beweisen läßt, wann immer man $\Gamma \vdash \Lambda$ zeigen kann.[33] Die klassische Logik geht in ihrem Verständnis des Widerspruchs noch ein Stück weiter. Unter der Voraussetzung, daß eine Aussage nur wahr oder falsch sein kann, ist eine Sequenz $\Gamma \vdash C$ bereits dann gültig, wenn ein Widerspruch aus Γ und $\neg C$ – der Negation der Konklusion – folgt. Dies führt dann zu der folgenden modifizierten Eliminationsregel für Λ.

$\Gamma \vdash C$ **by** `classical_contradiction`
$\Gamma, \neg C \vdash \Lambda$

Diese Regel gilt *ausschließlich* für die *klassische Logik*, da sie die Gültigkeit des Gesetzes vom ausgeschlossenen Dritten voraussetzt. Zudem ist sie im Verhältnis zu den anderen Regeln 'unsauber', da sie als Unterziel eine Sequenz erzeugt, die komplexere Formeln enthält als die ursprüngliche Sequenz.

Beispiel 2.2.17

Hier ist ein klassischer Beweis für das Gesetz vom ausgeschlossenen Dritten:

```
⊢ A∨¬A
by classical_contradiction
 \
  ¬(A∨¬A)  ⊢ Λ
  by not_e 1
  |\
  | ¬(A∨¬A)  ⊢  A∨¬A
  | by or_i1
  |  \
  |   ¬(A∨¬A)  ⊢  A
  |   by classical_contradiction
  |    \
  |     ¬(A∨¬A), ¬A  ⊢ Λ
  |     by not_e 1
  |     |\
  |     | ¬(A∨¬A), ¬A  ⊢ A∨¬A
  |     | by or_i2
  |     |  \
  |     |   ¬(A∨¬A), ¬A  ⊢ ¬A
  |     |    by hypothesis 2
  |      \
  |       A, Λ ⊢ Λ
  |       by hypothesis 2
   \
    Λ ⊢ Λ
    by hypothesis 1
```

Wie man sieht, erlaubt die Regel `classical_contradiction` zuweilen seltsam anmutende Beweise.

[33] Dies ergibt sich, wenn man auf $\Gamma \vdash C$ zuerst `cut` i Λ anwendet und anschließend `false_e` i auf das zweite Teilziel.

2.2.6 Quantoren, Variablenbindung und Substitution

Die bisherigen Inferenzregeln waren verhältnismäßig leicht zu erklären, da sie sich nur auf aussagenlogische Junktoren bezogen. Mit den Quantoren $\forall$ und $\exists$ wird jedoch eine größere Dimension des logischen Schließens eröffnet und eine Reihe von Themen treten auf, die bisher nicht behandelt werden mußten: *Substitution*, *freies* und *gebundenes Vorkommen* von Variablen. Die Aussagenlogik – also die Logik der Symbole $\wedge$, $\vee$, $\Rightarrow$ und $\neg$ – ist *entscheidbar*, aber für die Prädikatenlogik gilt dies nicht mehr.

Bei der Simulation der logischen Gesetze für Quantoren spielt die Ersetzung von Variablen einer Formel durch Terme – die sogenannte *Substitution* – eine Schlüsselrolle: eine Formel $\forall x{:}T\,.\,A$ gilt als bewiesen, wenn man in A die Variable x durch einen beliebigen Term ersetzen und dann beweisen kann. Das Gleiche gilt für $\exists x{:}T\,.\,A$, wenn man *einen* solchen Term findet.[34]

Um präzise zu definieren, was Substitution genau bedeutet, führen wir zunächst ein paar Begriffe ein, welche klarstellen, ob eine Variable in einer Formel substituiert werden darf oder nicht. Wir nennen ein Vorkommen einer Variablen x innerhalb einer Formel der Gestalt $\forall x{:}T\,.\,A$ bzw. $\exists x{:}T\,.\,A$ *gebunden*, denn in diesen Formeln ist der Name x nur ein Platzhalter. Die Formel würde sich nicht ändern, wenn man jedes Vorkommen von x durch eine andere Variable y ersetzt. In der Teilformel A dagegen tritt x *frei* auf, denn hier würde eine Umbenennung die Bedeutung der gesamten Formel ändern.

Definition 2.2.18 (Freies und gebundenes Vorkommen von Variablen)

Es sei $x, y \in \mathcal{V}$ Variablen, $f \in \mathcal{F}$ ein Funktionssymbol, $P \in \mathcal{P}$ ein Prädikatssymbol, $t_1, \ldots, t_n$ Terme und A, B Formeln. Das freie und gebundene Vorkommen der Variablen x in einem Ausdruck ist induktiv durch die folgenden Bedingungen definiert.

1. *In dem Term x kommt x frei vor. Die Variable y kommt überhaupt nicht vor, wenn x und y verschieden sind.*
2. *In $f(t_1, ..., t_n)$ bleibt jedes freie Vorkommen von x in einem Term t_i frei und jedes gebundene Vorkommen von x in einem Term t_i gebunden.*
3. *In Λ kommt x nicht vor.*
4. *In $t_1{=}t_2$ bleibt jedes freie Vorkommen von x in einem Term t_i frei und jedes gebundene Vorkommen von x in einem Term t_i gebunden.*
5. *In $P(t_1, ..., t_n)$ bleibt jedes freie Vorkommen von x in einem Term t_i frei und jedes gebundene Vorkommen von x in einem Term t_i gebunden.*
6. *In $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$ und (A) bleibt jedes freie Vorkommen von x in den Formeln A und B frei und jedes gebundene Vorkommen von x in den Formeln A und B gebunden.*
7. *In $\forall x{:}T\,.\,A$ und $\exists x{:}T\,.\,A$ wird jedes freie Vorkommen von x in A gebunden. Gebundene Vorkommen von x in A bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn x und y verschieden sind.*

Das freie Vorkommen der Variablen $x_1, \ldots, x_n$ in einem Term t bzw. einer Formel A wird mit $t[x_1, \ldots, x_n]$ bzw. $A[x_1, \ldots, x_n]$ gekennzeichnet.

Eine Term bzw. eine Formel ohne freie Variablen heißt geschlossen.

[34] Man beachte jedoch, daß sich die Semantik des Allquantors in einem formalen System nur zum Teil wiederspiegeln läßt. In der Semantik ist die Formel $\forall x{:}T\,.\,A$ wahr, wenn die durch A dargestellte Aussage für alle möglichen Interpretationen der Variablen x wahr ist. In einem formalen Kalkül kann man dagegen nur prüfen, ob für jeden *Term* t gilt, daß die Formel A *beweisbar* ist, wenn man jedes Vorkommen der Variablen x durch t ersetzt.

Das ist nicht dasselbe, da ein Modell Werte enthalten mag, die sich nicht durch einen Term ausdrücken lassen. Wenn das Bereichssymbol T zum Beispiel als Menge der reellen Zahlen interpretiert wird, so gibt es überabzählbar viele Werte, die man als Interpretation für x einsetzen kann, aber nur abzählbar viele Terme.

Dieses Problem gilt allerdings nur in der klassischen Mathematik, welche die Existenz reeller Zahlen durch das Axiom der kleinsten oberen Schranke postuliert. In der intuitionistischen Mathematik befaßt man sich nur mit Objekten, die – zumindest gedanklich – konstruierbar sind, also mit endlichen Mitteln beschrieben werden können.

Wir haben die Definition des freien und gebundenen Vorkommens von Variablen bewußt ausführlicher gestaltet, als dies für die reine Prädikatenlogik notwendig ist. Vorerst kann eine Variable in einem Term oder einer atomaren Formel nicht gebunden vorkommen. Dies wird sich aber schon im Zusammenhang mit dem λ-Kalkül in Abschnitt 2.3 ändern, wo zusätzlich zu den Quantoren weitere Möglichkeiten entstehen, frei vorkommende Variablen zu *binden*. Wichtig ist auch, daß Variablen in einem Ausdruck sowohl frei als auch gebunden vorkommen können, wie das folgende Beispiel illustriert.

Beispiel 2.2.19

Wir wollen das Vorkommen der Variablen `x` im Term `(∀x:T. P(x) ∧ Q(x)) ∧ R(x)` analysieren

- Die Variable `x` tritt frei auf im Term `x`.
- Nach (5.) ändert sich daran nichts, wenn die Terme `P(x), Q(x)` und `R(x)` gebildet werden.
- Nach (6.) ist `x` auch frei in `P(x) ∧ Q(x)`.
- In `∀x:T. P(x) ∧ Q(x)` ist `x` gemäß (7.) – von außen betrachtet – gebunden.
- Dies bleibt so auch nach der Klammerung in `(∀x:T. P(x) ∧ Q(x))`.
- In der Konjunktion `(∀x:T. P(x) ∧ Q(x)) ∧ R(x)` tritt `x` gemäß (6.) nun sowohl frei als auch gebunden auf.

Insgesamt haben wir folgende Vorkommen von `x` innerhalb des Terms `(∀x:T. P(x) ∧ Q(x)) ∧ R(x)` und seiner Teilterme.

$$\overbrace{\overbrace{(\forall \mathtt{x}\mathtt{:T.}\ \underbrace{\mathtt{P(x)}\ \wedge\ \mathtt{Q(x)}}_{\mathtt{x}\ \textit{frei}})}^{\mathtt{x}\ \textit{gebunden}}\ \wedge\ \underbrace{\mathtt{R(x)}}_{\mathtt{x}\ \textit{frei}}}^{\mathtt{x}\ \textit{frei und gebunden}}$$

Mit Hilfe der eben definierten Konzepte "freies" bzw. "gebundenes Vorkommen" können wir nun die Substitution einer Variablen x durch einen Term t innerhalb einer Formel A genau definieren. Dabei soll die Substitution das syntaktische Gegenstück zum Einsetzen von Werten in Teilformeln quantifizierter Formeln darstellen. Es ist leicht einzusehen, daß hierfür nur die *freien* Vorkommen von x in A zur Verfügung stehen. Die gebundenen Vorkommen von x sind nämlich, wie oben erwähnt, austauschbare Platzhalter, bei denen der tatsächliche Name keine Rolle spielt. Deshalb sollte eine Substitution von x durch t bei den beiden Formeln $\exists x{:}T.\ P(x)$ und $\exists y{:}T.\ P(y)$ denselben Effekt haben, d.h. gar keinen. Die folgende Definition der Substitution trägt diesem Gedanken Rechnung.

Definition 2.2.20 (Substitution)

- *Eine Substitution ist eine endliche Abbildung σ von der Menge $\mathcal{V}$ der Variablen in die Menge der Terme (d.h. $\sigma(x)\neq x$ gilt nur für endlich viele Variablen $x\in\mathcal{V}$).*
- *Gilt $\sigma(x_1)=t_1,\ \ldots,\ \sigma(x_n)=t_n$, so schreiben wir kurz $\sigma = [t_1,\ldots,t_n/x_1,\ldots,x_n]$.*[35]
- *Die Anwendung einer Substitution $\sigma=[t/x]$ auf einen prädikatenlogischen Ausdruck A wird mit $A[t/x]$ bezeichnet und ist induktiv wie folgt definiert.*

  1. $x[t/x] \;=\; t$;
     $x[t/y] \;=\; x$, *wenn x und y verschieden sind.*
  2. $f(t_1,...,t_n)[t/x] \;=\; f(t_1[t/x],...,t_n[t/x])$.
  3. $\Lambda[t/x] \;=\; \Lambda$.
  4. $(t_1{=}t_2)[t/x] \;=\; t_1[t/x]{=}t_2[t/x]$.
  5. $P(t_1,...,t_n)[t/x] \;=\; P(t_1[t/x],...,t_n[t/x])$.

6. $$\begin{array}{lcl}(\neg A)[t/x] & = & \neg A[t/x],\\ (A \wedge B)[t/x] & = & A[t/x] \wedge B[t/x],\\ (A \vee B)[t/x] & = & A[t/x] \vee B[t/x],\\ (A \Rightarrow B)[t/x] & = & A[t/x] \Rightarrow B[t/x],\\ (A)[t/x] & = & (A[t/x]).\end{array}$$

7. $$\begin{array}{lcllcl}(\forall x{:}T\,.\,A)[t/x] & = & \forall x{:}T\,.\,A & (\exists x{:}T\,.\,A)[t/x] & = & \exists x{:}T\,.\,A\\ (\forall x{:}T\,.\,A)[t/y] & = & (\forall z{:}T\,.\,A[z/x])[t/y] & (\exists x{:}T\,.\,A)[t/y] & = & (\exists z{:}T\,.\,A[z/x])[t/y],\end{array}$$
wenn x und y verschieden sind, y frei in A vorkommt und x frei in t vorkommt. z ist eine neue Variable, die von x und y verschieden ist und weder in A noch in t vorkommt.

$$\begin{array}{lcllcl}(\forall x{:}T\,.\,A)[t/y] & = & \forall x{:}T\,.\,A[t/y] & (\exists x{:}T\,.\,A)[t/y] & = & \exists x{:}T\,.\,A[t/y],\end{array}$$
wenn x und y verschieden sind und es der Fall ist, daß y nicht frei in A ist oder daß x nicht frei in t vorkommt.

Dabei sind $x, y \in \mathcal{V}$ Variablen, $f \in \mathcal{F}$ ein Funktionssymbol, $P \in \mathcal{P}$ ein Prädikatssymbol, $t, t_1, \ldots, t_n$ Terme und A, B Formeln.

- *Die Anwendung komplexerer Substitutionen auf einen Ausdruck, $PA[t_1, \ldots, t_n/x_1, \ldots, x_n]$, wird entsprechend definiert.*

In Definition 2.2.20(7.) mußten wir für die Anwendung von Subtitutionen auf quantifizierte Formeln einige Sonderfälle betrachten. Die Formel ändert sich gar nicht, wenn die zu ersetzende Variable gebunden ist. Andernfalls müssen wir vermeiden, daß eine freie Variable in den Bindungsbereich eines Quantors gerät ("*capture*"), daß also zum Beispiel der Allquantor nach dem Ersetzen eine Variable bindet, die in dem eingesetzten Term t frei war. Die folgende Substitution entspräche nämlich nicht dem Gedanken einer Ersetzung:

$$(\forall \mathtt{y}{:}\mathbb{N}\,.\,\mathtt{x<y})[\mathtt{y/x}] = \forall \mathtt{y}{:}\mathbb{N}\,.\,\mathtt{y<y}$$

Deshalb wird zunächst eine Umbenennung der gebundenen Variablen vorgenommen zu einer Variablen, die bisher überhaupt nicht vorkam, und die Ersetzung geht in zwei Phasen vor sich

$$(\forall \mathtt{y}{:}\mathbb{N}\,.\,\mathtt{x<y})[\mathtt{y/x}] = (\forall \mathtt{z}{:}\mathbb{N}\,.\,\mathtt{x<z})[\mathtt{y/x}] = \forall \mathtt{z}{:}\mathbb{N}\,.\,\mathtt{y<z}$$

Der letzte Fall der Definition 2.2.20 beschreibt die normale Situation: ersetzt wird in den Unterformeln. Wir wollen nun die Substitution an einem Beispiel erklären.

Beispiel 2.2.21

$$\begin{array}{lcl}
 & & ((\forall \mathtt{y}{:}\mathbb{N}.\ \mathtt{R(+(x,y))} \wedge \exists \mathtt{x}{:}\mathbb{N}.\mathtt{x=y} \wedge \mathtt{P(x)})[\mathtt{-(y,4)/x}]\\
= & & (\forall \mathtt{y}{:}\mathbb{N}.\ \mathtt{R(+(x,y))} \wedge \exists \mathtt{x}{:}\mathbb{N}.\mathtt{x=y})[\mathtt{-(y,4)/x}] \wedge \mathtt{P(x)}[\mathtt{-(y,4)/x}]\\
= & & (\forall \mathtt{z}{:}\mathbb{N}.\ \mathtt{R(+(x,z))} \wedge \exists \mathtt{x}{:}\mathbb{N}.\mathtt{x=z})[\mathtt{-(y,4)/x}] \wedge \mathtt{P(-(y,4))}\\
= & & (\forall \mathtt{z}{:}\mathbb{N}.\ \mathtt{R(+(x,z))}[\mathtt{-(y,4)/x}] \wedge (\exists \mathtt{x}{:}\mathbb{N}.\mathtt{x=z})[\mathtt{-(y,4)/x}]) \wedge \mathtt{P(-(y,4))}\\
= & & (\forall \mathtt{z}{:}\mathbb{N}.\ \mathtt{R(+(-(y,4),z))} \wedge \exists \mathtt{x}{:}\mathbb{N}.\mathtt{x=z} \wedge \mathtt{P(-(y,4))}
\end{array}$$

Mit diesen Begriffen sind wir nun in der Lage, die noch ausstehenden Regeln für die Quantoren aufzustellen und zu begründen.

[35] Leider gibt es eine Fülle von Notationen für die Substitution. Anstelle von $\sigma(x){=}t$ schreiben manche $\sigma{=}[x\backslash t]$ andere $\sigma{=}\{x\backslash t\}$ oder $\sigma{=}\{t/x\}$. Die in diesem Skriptum verwendete Konvention $\sigma{=}[t/x]$ entspricht derjenigen, die in der Literatur zu konstruktiven Kalkülen und zum λ-Kalkül üblich ist. All diesen Schreibweisen ist aber gemein, daß die zu ersetzende Variable "unterhalb" des Terms steht, der sie ersetzt. Auch hat es sich eingebürgert, die Anwendung einer Substitution auf einen Ausdruck dadurch zu kennzeichnen, daß man die Substitution *hinter* den Ausdruck schreibt.

Universelle Quantifizierung

Entsprechend unserer Semantikdefinition 2.2.8 auf Seite 26 ist eine Formel $\forall x{:}T\,.\,A$ unter einer Interpretation genau dann wahr, wenn A für jeden möglichen Wert wahr ist, den wir für x einsetzen können. Da eine allgemeingültige Formel für jede beliebige Interpretation ι wahr sein muß, folgt hieraus, daß $\forall x{:}T\,.\,A$ genau dann allgemeingültig ist, wenn die Teilformel A allgemeingültig ist – unabhängig davon, ob x in A überhaupt vorkommt oder nicht. Damit genügt es, A zu zeigen, wenn man $\forall x{:}T\,.\,A$ beweisen möchte.

Für Sequenzen, welche ja auch die Abhängigkeit der Konklusion von den Annahmen beschreiben, muß man allerdings berücksichtigen, daß x in den bisherigen Annahmen bereits frei vorkommen könnte. Eine Regel

aus $\Gamma \vdash A$ *folgt* $\Gamma \vdash \forall x{:}T\,.\,A$

würde also eine *unzulässige Verallgemeinerung* darstellen, da in Γ Annahmen über das spezielle x stehen könnten, das durch den Quantor gebunden ist. Ein Schluß von $\texttt{x=4} \vdash \texttt{x=4}$ auf $\texttt{x=4} \vdash \forall \texttt{x:}\mathbb{N}\,.\,\texttt{x=4}$ ist sicherlich nicht akzeptabel. Die Grundlage für eine korrekte Einführungsregel für den Allquantor liefert das folgende Lemma.

Lemma 2.2.22

Wenn die Variable x in der Hypothesenliste Γ nicht frei vorkommt, dann ist $\Gamma \vdash \forall x{:}T\,.\,A$ genau dann gültig, wenn $\Gamma \vdash A$ gültig ist.

In Gentzens Originalarbeit [Gentzen, 1935] hat diese Feststellung zu einer sogenannten *Eigenvariablenbedingung* bei der Einführungsregel für den Allquantor geführt. Die Regel darf nur angewandt werden, wenn die quantifizierte Variable nicht in der Hypothesenliste vorkommt. Andernfalls ist ihre Anwendung verboten. Diese Einschränkung ist allerdings etwas zu restriktiv. Es genügt zu fordern, daß die quantifizierte Variable umbenannt werden muß, wenn sie bereits in der Hypothesenliste erscheint. Dies führt zu der folgenden Einführungsregel (*Verallgemeinerung*) für den Allquantor

$\Gamma \vdash \forall x{:}T\,.\,A$ **by** `all_i` *Umbenennung* $[x'/x]$*, wenn x in Γ frei vorkommt*
$\Gamma, x'{:}T \vdash A[x'/x]$

Aus Gründen, die im Zusammenhang mit der Typentheorie deutlicher werden, fügt die Einführungsregel eine Deklaration einer Variablen zur Hypothesenliste hinzu. Dabei wird der Name x durch eine *neue* Variable x' ersetzt, die bisher nirgends vorkam, wenn x in Γ frei vorkommt (dies ersetzt die Eigenvariablenbedingung).[36]

Die Eliminationsregel (*Spezialisierung*) besagt, daß jeder Spezialfall $A[t/x]$ gültig ist, wenn $\forall x{:}T\,.\,A$ gültig ist. Um also eine Konklusion C unter der Annahme $\forall x{:}T\,.\,A$ zu zeigen, reicht es zu zeigen, daß C unter den bisherigen Hypothesen und der zusätzlichen Annahme $A[t/x]$ gilt. Der Term t für diese *Instantiierung* der universell quantifizierten Formel muß hierbei als Parameter der Regel angegeben werden.

$\Gamma, \forall x{:}T\,.\,A, \Delta \vdash C$ **by** `all_e` i t
$\Gamma, \forall x{:}T\,.\,A, \Delta, A[t/x] \vdash C$

Die Annahme $\forall x{:}T\,.\,A$ bleibt hierbei erhalten, da sie im Beweis noch einmal benötigt werden könnte wie z.B. in einem Beweis für $\forall \texttt{x:}\mathbb{N}\,.\,\texttt{>(x,0)} \vdash \texttt{>(1,0)} \wedge \texttt{>(2,0)}$.

Beispiel 2.2.23

Abbildung 2.5 zeigt einen Beweis für das Distributivgesetz zwischen Allquantor und Konjunktion.

$$(\forall x{:}T\,.\,P(x) \wedge Q(x)) \quad \Rightarrow \quad (\forall x{:}T\,.\,P(x)) \wedge (\forall x{:}T\,.\,Q(x))$$

In diesem Beweis ist die Reihenfolge der Regeln für den Allquantor sehr wichtig. Der Term x kann erst in der Regel `all_e 1` x benutzt werden, nachdem x durch `all_i` freigegeben wurde.

[36] Üblicherweise wird der Kalkül benutzt, um geschlossene Theoreme ohne freie Variablen zu beweisen. Das bedeutet, daß die Ausgangssequenz die Gestalt '$\vdash C$' hat, wobei C keine freie Variable besitzt. Im Verlauf des Beweises können freie Variablen dann nur in die Hypothesenliste geraten, wenn sie zuvor deklariert wurden. Für die Regel `all_i` reicht es daher, das Vorkommen einer Deklaration von x zu überprüfen.

```
∀x:T.P(x)∧Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
by and_i
|\
| ∀x:T.P(x)∧Q(x)  ⊢ ∀x:T.P(x)
| by all_i
|  \
|   ∀x:T.P(x)∧Q(x), x:T  ⊢  P(x)
|   by all_e 1 x
|    \
|     ∀x:T.P(x)∧Q(x), x:T, P(x)∧Q(x)  ⊢  P(x)
|     by and_e 3
|      \
|       ∀x:T.P(x)∧Q(x), x:T, P(x), Q(x)  ⊢  P(x)
|       by hypothesis 3
 \
  ∀x:T.P(x)∧Q(x)  ⊢ ∀x:T.Q(x)
  by all_i
   \
    ∀x:T.P(x)∧Q(x), x:T  ⊢  Q(x)
    by all_e 1 x
     \
      ∀x:T.P(x)∧Q(x), x:T, P(x)∧Q(x)  ⊢  Q(x)
      by and_e 3
       \
        ∀x:T.P(x)∧Q(x), x:T, P(x), Q(x)  ⊢  Q(x)
        by hypothesis 4
```

Abbildung 2.5: Sequenzenbeweis für das Distributivgesetz zwischen Allquantor und Konjunktion

Würde man versuchen, den Beweis dadurch zu verkürzen, daß man die Annahmen $P(x)$, $Q(x)$ erzeugt, *bevor* man die Einführungsregel `all_i` verwendet, so würde folgende Situation entstehen:

```
∀x:T.P(x)∧Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
by all_e 1 x
 \
  ∀x:T.P(x)∧Q(x), P(x)∧Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
  by and_e 2
   \
    ∀x:T.P(x)∧Q(x), P(x), Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
    by and_i
    |\
    | ∀x:T.P(x)∧Q(x), P(x), Q(x)  ⊢  ∀x:T.P(x)
    | by all_i
    |  \
    |   ∀x:T.P(x)∧Q(x), P(x), Q(x), x':T  ⊢  P(x')
    |   by ?????
     \
      ∀x:T.P(x)∧Q(x), P(x), Q(x)  ⊢  ∀x:T.Q(x)
      by all_i
       \
        ∀x:T.P(x)∧Q(x), P(x), Q(x), x':T  ⊢  Q(x')
        by ?????
```

Da bei der Anwendung von `all_i` die Variable x in den Annahmen frei vorkommt, muß eine Umbenennung stattfinden. Damit sind jedoch die Hypothesen $P(x)$ bzw. $Q(x)$ für den Beweis unbrauchbar.[37]

Existentielle Quantifizierung

Eine Formel $\exists x{:}T\,.\,A$ ist genau dann wahr, wenn wir für x mindestens einen Wert einsetzen können, der A wahr macht. Um also $\exists x{:}T\,.\,A$ zeigen zu können, reicht es zu zeigen, daß $A[t/x]$ für einen Term t gilt. Dieser

[37] Prinzipiell wäre es möglich, den Beweis dennoch zu Ende zu führen, da der Weg des Beweises aus Abbildung 2.5 immer noch verfolgt werden kann. Der Beweis würde nur einige unnötige Schritte enthalten. Sowohl in Gentzens Originalkalkül [Gentzen, 1935] als auch im NuPRL System ist dieser Beweisversuch jedoch völlig undurchführbar. Die Eigenvariablenbedingung von Gentzen würde die Anwendung von `all_i` verbieten. In NuPRL wäre bereits die Regel `all_e 1` x nicht erlaubt, da der Term x Variablen enthält, die noch nicht deklariert waren.

Term muß als Parameter der Einführungsregel für den Existenzquantor angegeben werden.

$$\frac{\Gamma \vdash \exists x{:}T\,.\,A \quad \textbf{by ex\_i } t}{\Gamma \vdash A[t/x]}$$

Die Eliminationsregel ist etwas komplizierter, weil hier ähnliche Probleme auftreten wie bei der Einführungsregel für den Allquantor. Da eine allgemeingültige Formel für jede beliebige Interpretation ι wahr sein muß, folgt eine Konklusion C aus $\exists x{:}T\,.\,A$ genau dann, wenn sie aus A folgt. Allerdings ist hierbei wieder zu beachten, daß die freie Variable x nicht auch in C oder Γ frei ist. Ansonsten würde $A \vdash C$ leichter zu beweisen sein als $\exists x{:}T\,.\,A \vdash C$. Das folgende Lemma ist das Analogon zu Lemma 2.2.22 auf Seite 39.

Lemma 2.2.24

Wenn die Variable x in der Konklusion C und der Hypothesenliste Γ nicht frei vorkommt, dann ist $\Gamma, \exists x{:}T\,.\,A \vdash C$ genau dann gültig, wenn $\Gamma, A \vdash C$ gültig ist.

In Gentzens Sequenzenkalkül gilt deshalb auch bei der Eliminationsregel für den Existenzquantor eine Eigenvariablenbedingung. Wir ersetzen diese wiederum durch die Forderung nach einer Variablenumbenennung.

$$\frac{\Gamma,\exists x{:}T\,.\,A,\Delta \vdash C \quad \textbf{by ex\_e } i}{\Gamma,x'{:}T,A[x'/x],\Delta \vdash C} \qquad \textit{Umbenennung } [x'/x]\textit{, wenn } x \textit{ in } C \textit{ oder } \Gamma \textit{ frei vorkommt}$$

Beispiel 2.2.25

Hier ist ein Beweis für die Vertauschbarkeit von Existenzquantoren.

$\exists x{:}T\,.\exists y{:}S\,.P(x,y) \vdash \exists y{:}S\,.\exists x{:}T\,.P(x,y)$
by ex_e 1

$x{:}T,\ \exists y{:}S\,.P(x,y) \vdash \exists y{:}S\,.\exists x{:}T\,.P(x,y)$
by ex_e 2

$x{:}T,\ y{:}S,\ P(x,y) \vdash \exists y{:}S\,.\exists x{:}T\,.P(x,y)$
by ex_i y

$x{:}T,\ y{:}S,\ P(x,y) \vdash \exists x{:}T\,.P(x,y)$
by ex_i x

$x{:}T,\ y{:}S,\ P(x,y) \vdash P(x,y)$
by hypothesis 3

2.2.7 Gleichheit

Mit den Regel für die Quantoren und Junktoren ist der Kalkül für die Prädikatenlogik abgeschlossen. Die Behandlung von Gleichheit gehört eigentlich nicht zur Logik dazu. Dennoch ist dies eine derart fundamentale mathematische Beweistechnik, daß wir bereits jetzt auf ihre Repräsentation in formalen Kalkülen eingehen wollen.

Eine Formel $t_1{=}t_2$ ist genau dann wahr, wenn t_1 und t_2 die gleichen Objekte beschreiben. Um dies zu beweisen, müßte man eigentlich auf die konkreten Eigenschaften der Terme eingehen und erklären, wie man den "Wert" eines Terms bestimmt – also, daß zum Beispiel `+(4,5)` und `9` das gleiche Objekt kennzeichnen. Innerhalb der Prädikatenlogik ist das nicht möglich, da wir die Bedeutung von Funktions- und Prädikatssymbolen ja gerade *nicht* berücksichtigen wollen.[38] Nichtsdestotrotz gibt es einige logische Grundregeln zur Verarbeitung von Gleichheit, die von der konkreten Bedeutung eines Terms unabhängig sind.

[38] Die *Auswertung* von Termen innerhalb eines formalen Kalküls ist genau das zentrale Thema des Abschnitts 2.3. Der λ-Kalkül beschreibt, wie sogenannte *Normalformen* eines Terms – das syntaktische Gegenstück zu einem *Wert* – zu *berechnen* sind. Die Frage nach der *semantischen Gleichheit* wird damit prinzipiell durch Berechnung der Normalform und einen Test auf syntaktische Gleichheit (die Reflexivitätsregel) gelöst. Die hierbei entstehenden Probleme werden in Abschnitt 2.3 ausführlich besprochen.

Zwei Terme t_1 und t_2 sind mit Sicherheit gleich, wenn sie syntaktisch identisch sind. Dies ist das Gesetz der *Reflexivität*.

$\Gamma \vdash t{=}t$ **by** `reflexivity`

Offensichtlich spielt bei der Gleichheit die Reihenfolge keine Rolle. $t_1{=}t_2$ ist wahr, wenn dies für $t_2{=}t_1$ gilt. Gleichheit ist also eine *symmetrische* Relation.

$\Gamma \vdash t_1{=}t_2$ **by** `symmetry`
$\Gamma \vdash t_2{=}t_1$

Das letzte fundamentale Gesetz der Gleichheit ist *Transitivität*. Zwei Terme t_1 und t_2 sind gleich, wenn beide gleich sind mit einem dritten Term u. In der entsprechenden Inferenzregel ist u als Parameter anzugeben.

$\Gamma \vdash t_1{=}t_2$ **by** `transitivity` u
$\Gamma \vdash t_1{=}u$
$\Gamma \vdash u{=}t_2$

Diese drei Regeln besagen, auf welche Arten man – neben dem Ausrechnen – die Gleichheit zweier Terme beweisen kann. Sie entsprechen also den Einführungsregeln für die Gleichheit. Wie aber kann man Gleichheiten verwenden? Dem intuitiven Verständnis nach kann man zwei gleiche Objekte durch nichts unterscheiden. Das bedeutet, daß jede Eigenschaft, die für das eine Objekt gilt, auch für das andere gilt.[39] Dieses *Kongruenzgesetz* wird formal durch eine Substitutionsregel ausgedrückt. Um eine Formel A zu beweisen, in der ein Term t_1 vorkommt, genügt es t_1 durch einen gleichwertigen Term t_2 zu ersetzen und die entsprechend modifizierte Formel zu beweisen. Die Gleichheit $t_1{=}t_2$ muß hierzu angegeben werden.

$\Gamma \vdash A[t_1/x]$ **by** `substitution` $t_1{=}t_2$
$\Gamma \vdash t_1{=}t_2$
$\Gamma \vdash A[t_2/x]$

Man hätte diese Regel auch als echte Eliminationsregel formulieren können, bei der die Gleichheit $t_1{=}t_2$ bereits in den Annahmen steckt. Die oben angegebene Form ist aber in praktischen Anwendungen handlicher.

Beispiel 2.2.26

Die Substitutionsregel macht die Transitivitätsregel eigentlich überflüssig. Hier ist ein Beweis für eine Simulation der Transitivitätsregel.

$t_1{=}u \wedge u{=}t_2 \vdash t_1{=}t_2$
by `and_e 1`
\
$t_1{=}u,\ u{=}t_2 \vdash t_1{=}t_2$
by `substitution` $t_1{=}u$
|\
| $t_1{=}u,\ u{=}t_2 \vdash t_1{=}u$
| **by** `hypothesis 1`
\
$t_1{=}u,\ u{=}t_2 \vdash u{=}t_2$
by `hypothesis 2`

Die Symmetrieregel läßt sich in ähnlicher Weise simulieren.

Abbildung 2.6 faßt alle Regeln des analytischen Sequenzenkalküls für die Prädikatenlogik zusammen.

2.2.8 Eigenschaften des Kalküls

Die wichtigsten Eigenschaften, die man von einem Kalkül fordern sollte, sind Korrektheit und Vollständigkeit (vgl. Definition 2.1.6 auf Seite 16), also die Eigenschaft, daß alle wahren Aussagen beweisbar sind und keine falschen. Mit Hilfe der in Definition 2.2.10 und 2.2.13 (siehe Seite 29 und 30) geprägten Begriffe lassen sich diese Forderungen nun wie folgt für die Prädikatenlogik erster Stufe präzisieren.

[39] In der Prädikatenlogik *höherer* Stufe drückt man dies so aus, daß man über alle Prädikate quantifiziert:
$t_1{=}t_2 \equiv \forall \mathtt{P}.\ \mathtt{P}(t_1) \Leftrightarrow \mathtt{P}(t_2)$.

$\Gamma,C,\Delta \vdash C$ **by** `hypothesis` i	
$\Gamma,\Delta \vdash C$ **by** `cut` i A $\Gamma,\Delta \vdash A$ $\Gamma,A,\Delta \vdash C$	$\Gamma,A,\Delta \vdash C$ **by** `thin` i $\Gamma,\Delta \vdash C$
$\Gamma \vdash A \wedge B$ **by** `and_i` $\Gamma \vdash A$ $\Gamma \vdash B$	$\Gamma,A \wedge B,\Delta \vdash C$ **by** `and_e` i $\Gamma,A,B,\Delta \vdash C$
$\Gamma \vdash A \vee B$ **by** `or_i1` $\Gamma \vdash A$	
$\Gamma \vdash A \vee B$ **by** `or_i2` $\Gamma \vdash B$	$\Gamma,A \vee B,\Delta \vdash C$ **by** `or_e` i $\Gamma,A,\Delta \vdash C$ $\Gamma,B,\Delta \vdash C$
$\Gamma \vdash A \Rightarrow B$ **by** `imp_i` $\Gamma,A \vdash B$	$\Gamma,A \Rightarrow B,\Delta \vdash C$ **by** `imp_e` i $\Gamma,A \Rightarrow B,\Delta \vdash A$ $\Gamma,\Delta,B \vdash C$
$\Gamma \vdash \neg A$ **by** `not_i` $\Gamma,A \vdash \Lambda$	$\Gamma,\neg A,\Delta \vdash C$ **by** `not_e` i $\Gamma,\neg A,\Delta \vdash A$ $\Gamma,\Delta,\Lambda \vdash C$ $\Gamma,\Lambda,\Delta \vdash C$ **by** `false_e` i
$\Gamma \vdash \forall x{:}T\,.\,A$ **by** `all_i` * $\Gamma,x'{:}T \vdash A[x'/x]$	$\Gamma,\forall x{:}T\,.\,A,\Delta \vdash C$ **by** `all_e` i t $\Gamma,\forall x{:}T\,.\,A,\Delta,A[t/x] \vdash C$
$\Gamma \vdash \exists x{:}T\,.\,A$ **by** `ex_i` t $\Gamma \vdash A[t/x]$	$\Gamma,\exists x{:}T\,.\,A,\Delta \vdash C$ **by** `ex_e` i ** $\Gamma,x'{:}T,A[x'/x],\Delta \vdash C$
$\Gamma \vdash t{=}t$ **by** `reflexivity`	$\Gamma \vdash t_1{=}t_2$ **by** `symmetry` $\Gamma \vdash t_2{=}t_1$
$\Gamma \vdash t_1{=}t_2$ **by** `transitivity` u $\Gamma \vdash t_1{=}u$ $\Gamma \vdash u{=}t_2$	$\Gamma \vdash A[t_1/x]$ **by** `substitution` $t_1{=}t_2$ $\Gamma \vdash t_1{=}t_2$ $\Gamma \vdash A[t_2/x]$
$\Gamma \vdash C$ **by** `classical_contradiction` $\Gamma,\ \neg C \vdash \Lambda$	

**: Die Umbenennung* $[x'/x]$ *erfolgt, wenn* x *in* Γ *frei vorkommt*

***: Die Umbenennung* $[x'/x]$ *erfolgt, wenn* x *in* C *oder* Γ *frei vorkommt.*

Abbildung 2.6: Der analytische Sequenzenkalkül für die Prädikatenlogik erster Stufe

Definition 2.2.27

Ein Kalkül für die Prädikatenlogik heißt

- *korrekt, wenn mit seinen Inferenzregeln nur gültige Formeln ableitbar sind,*
- *vollständig, wenn jede gültige Formel in dem Kalkül ableitbar ist.*

Auf die einzelnen Regeln des analytischen Sequenzenkalküls bezogen bedeutet Korrektheit also folgendes.

Eine Regel des Sequenzenkalküls ist korrekt, wenn die Allgemeingültigkeit eines Beweisziels aus der Gültigkeit der erzeugten Teilziele folgt.

Diese Eigenschaften läßt sich für jede einzelne Regel relativ leicht zeigen. Wir wollen dies am Beispiel der Einführungsregel für die Implikation zeigen.

Lemma 2.2.28

Die Einführungsregel `imp_i` *für die Implikation ist korrekt*

Beweis: Die Regel `imp_i` hat die Gestalt $\dfrac{\Gamma \vdash A \Rightarrow B}{\Gamma, A \vdash B}$ **by** `imp_i`

Zu zeigen ist, daß aus der Gültigkeit des von `imp_i` erzeugten Teilziels $\Gamma, A \vdash B$ die Allgemeingültigkeit des Beweisziels $\Gamma \vdash A \Rightarrow B$ folgt.

Es sei $\Gamma = A_1, \ldots, A_n$ eine beliebige Folge von Formeln, A, B Formeln und $\Gamma, A \vdash B$ eine allgemeingültige Sequenz. Dann ist $\iota(\Gamma, A \vdash B) = \mathsf{wahr}$ für jede Interpretationsfunktion ι.

Es sei also ι eine beliebige Interpretationsfunktion. Nach Definition 2.2.13 auf Seite 30 folgt dann, daß $\iota(B)$=wahr ist, wann immer $\iota(A_1)$=wahr, $\ldots$ $\iota(A_n)$=wahr und $\iota(A)$=wahr ist.

Es sei also $\iota(A_1)$=wahr, $\ldots$ und $\iota(A_n)$=wahr. Dann ist $\iota(B)$=wahr, wann immer $\iota(A)$=wahr ist. Nach Definition 2.2.8 ist somit $\iota(A \Rightarrow B)$=wahr. Damit gilt, daß $\iota(A \Rightarrow B)$=wahr ist, wann immer $\iota(A_1)$=wahr, $\ldots$ und $\iota(A_n)$=wahr ist. Somit ist $\iota(\Gamma \vdash A \Rightarrow B) = \mathsf{wahr}$. Da ι beliebig gewählt war, haben wir gezeigt, daß $\Gamma \vdash A \Rightarrow B$ eine allgemeingültige Sequenz ist. Damit ist die Behauptung bewiesen. □

Die Korrektheitsbeweise für die anderen Regeln des Sequenzenkalküls verlaufen ähnlich und folgen dabei den bei der Präsentation gegebenen Begründungen.

Die Vollständigkeit des Kalküls ist nicht so leicht zu beweisen, da hierzu komplexe Induktionen über den Wahrheitswert von Formeln geführt werden müssen. Wir wollen an dieser Stelle auf den Beweis verzichten und verweisen interessierte Leser auf [Richter, 1978, Seite 132].

Insgesamt ist also der in Abbildung 2.6 zusammmengestellte Sequenzenkalkül korrekt und vollständig für die intuitionistische Prädikatenlogik erster Stufe, wenn man die Regel `classical_contradiction` wegläßt, und korrekt und vollständig für die klassische Prädikatenlogik erster Stufe, wenn man sie hinzunimmt.

2.2.9 Beweismethodik

Der Sequenzenkalkül gibt uns Regeln an die Hand, mit denen wir die Korrektheit jeder wahren prädikatenlogischen Aussage auf rein formale Weise beweisen können. Er garantiert die *Korrektheit* von Beweisen, die nach seinen Regeln aufgebaut sind. Diese sind analytisch formuliert, unterstützen also eine zielorientierte Vorgehensweise bei der Beweisführung. Beweiseditoren wie das NuPRL System ermöglichen es, Beweise schrittweise nur durch die Angabe von Regelnamen und Kontrollparametern zu konstruieren.

Der Sequenzenkalkül beinhaltet jedoch keine Methode, *wie* man Beweise für eine vorgegebene Formel führen kann. So ist man hierbei im wesentlichen immer noch auf den eigenen Einfallsreichtum angeweisen. Nichtsdestotrotz lassen sich einige Leitlinien angeben, welche das Finden eines formalen Beweises erleichtern.[40]

[40] Diese Leitlinien sind auch der Kern von Taktiken (vgl. Kapitel 4.2), welche einen Großteil der Beweise automatisch finden.

- Wende die Hypothesenregel `hypothesis` und die Widerspruchsregel `false_e` an, wann immer dies möglich ist, da sie Teilbeweise (Zweige im Beweisbaum) abschließen.

- Verwende Einführungsregeln, um die Konklusion in kleinere Teile aufzubrechen, und Eliminationsregeln, um Annahmen aufzubrechen (*Dekomposition*).

- Wenn mehr als eine aussagenlogische Regel anwendbar ist, spielt bei klassischer Logik die Reihenfolge keine Rolle. Bei intuitionistischem Schließen sollte die Eliminationsregel für die Disjunktion `or_e` *vor* der Einführungsregel `or_i` (in der man sich für einen Fall entscheidet) angewandt werden. Andernfalls mag es sein, daß unbeweisbare Teilziele erzeugt werden.

- Die Regel `ex_i` und `all_e` sollten niemals vor den Regeln `ex_e` und `all_i` angewandt werden. Wie das Beispiel 2.2.23 auf Seite 39 zeigt, könnten die ersten Regeln freie Variablen in der Konklusion bzw. den Annahmen erzeugen, welche bei der Anwendung der zweiten zu Umbenennungen führen.[41]

- Wenn die obigen Richtlinien immer noch Wahlmöglichkeiten lassen, sollte die Regel benutzt werden, welche die wenigsten Teilziele erzeugt.

Der kritische Punkt bei der Erstellung von Sequenzenbeweisen ist meist die Auswahl eines Terms t, der bei der Auflösung eines Quantors für eine gebundene Variable x substituiert werden soll. Dieser Term kann eigentlich erst dann bestimmt werden, wenn man den Beweis zu Ende geführt hat. Auf der anderen Seite kann der Beweis nicht fortgesetzt werden, solange der Term t nicht angegeben wurde. Diese Problematik macht eine automatische Konstruktion von Sequenzenbeweisen sehr schwer und führt dazu, daß beim Schließen mit Quantoren eine Interaktion zwischen Mensch und Computer notwendig ist.

In den letzten Jahrzehnten wurden jedoch eine Reihe von Beweissuchverfahren zu maschinennäheren Kalkülen wie der *Resolution* oder den *Matrixkalkülen* (*Konnektionsmethode*) entwickelt. Sie erlauben es, zunächst die Anforderungen an Terme, welche für quantifizierte Variablen eingesetzt werden müssen, zu sammeln und dann den Term durch *Unifikation* zu bestimmen. Für die klassische Logik sind auf dieser Basis eine Fülle von Theorembeweisern implementiert worden. Für die intuitionistische Logik gibt es Ansätze, diese Methoden entsprechend zu übertragen [Wallen, 1990], was sich jedoch als erheblich komplizierter erweist.

Der Nachteil dieser Techniken ist, daß die erzeugten Beweise schwer zu verstehen sind und wieder in lesbare Beweise in natürlicher Deduktion oder im Sequenzenkalkül übersetzt werden müssen. Auch hier sind bereits erste Ansätze untersucht worden, die wir im zweiten Teil dieser Veranstaltung näher ansehen wollen.

2.3 Der λ-Kalkül

Wir haben im letzten Abschnitt bereits darauf hingewiesen, daß eine der großen Schwächen der Prädikatenlogik erster Stufe darin besteht, daß es keine (direkten) Möglichkeiten gibt, Schlüsse über den Wert von Termen zu führen. Die Tatsache, daß die Terme `+(4,7)`, `+(5,6)` und `11` gleich sind, läßt sich nicht vernünftig ausdrücken, da es an Mechanismen fehlt, den Wert dieser Terme auf rein syntaktischem Wege auszurechnen.

Um derartige Berechnungen durchführen zu können, wurde seit Beginn dieses Jahrhunderts eine Vielfalt von mathematischen Modellen entwickelt. Manche davon, wie die Turingmaschinen oder die Registermaschinen, orientieren sich im wesentlichen an dem Gedanken einer maschinellen Durchführung von Berechnungen. Andere, wie die μ-rekursiven Funktionen, basieren auf einem Verständnis davon, was intuitiv berechenbar ist und wie sich berechenbare Funktionen zu neuen zusammensetzen lassen.

Unter all diesen Kalkülen ist der λ-Kalkül der einfachste, da er nur drei Mechanismen kennt: die Definition einer Funktion (*Abstraktion*), die Anwendung einer Funktion auf ein Argument (*Applikation*) und die Auswertung einer Anwendung, bei der die Definition bekannt ist (*Reduktion*). Dabei werden bei der Auswertung

[41] NuPRL würde eine Anwendung von `ex_i` bzw. `all_e` mit freien Variablen als Kontrollparameter ohnehin verweigern.

ausschließlich Terme manipuliert, ohne daß hierbei deren Bedeutung in Betracht gezogen wird (was einem Rechner ohnehin unmöglich wäre). Die Syntax und der Berechnungsmechanismus sind also extrem einfach. Dennoch läßt sich zeigen, daß man mit diesen Mechanismen alle berechenbaren Funktionen simulieren kann.

Wir wollen die Grundgedanken des λ-Kalküls an einem einfachen Beispiel erläutern.

Beispiel 2.3.1

Es ist allgemein üblich, Funktionen dadurch zu charakterisieren, daß man ihr Verhalten auf einem beliebigen Argument x beschreibt. Um also zum Beispiel eine Funktion zu definieren, welche ihr Argument zunächst verdoppelt und anschließend die Zahl *drei* hinzuaddiert, schreibt man kurz:

```
f(x) = 2*x+3
```

Diese Notation besagt, daß das Symbol `f` als Funktion zu interpretieren ist, welche jedem Argument `x` den Wert der Berechnung zuordnet, die durch den Ausdruck `2*x+3` beschrieben ist. Dabei kann für `x` jede beliebige Zahl eingesetzt werden. Um nun einen konkreten Funktionswert wie zum Beispiel `f(4)` auszurechnen, geht man so vor, daß zunächst jedes Vorkommen des Symbols `x` durch `4` ersetzt wird, wodurch sich der Ausdruck `2*4+3` ergibt. Dieser wird anschließend ausgewertet und man erhält `11` als Resultat.

Bei diesem Prozeß spielt das Symbol `f` eigentlich keine Rolle. Es dient nur als Abkürzung für eine Funktionsbeschreibung, welche besagt, daß jedem Argument `x` der Wert `2*x+3` zuzuordnen ist. Im Prinzip müßte es also auch möglich sein, ohne dieses Symbol auszukommen und die Funktion durch einen Term zu beschreiben, der genau die Abbildung `x` $\mapsto$ `2*x+3` wiederspiegelt.

Genau dies ist die Grundidee des λ-Kalküls. Funktionen lassen sich eindeutig durch einen Term beschreiben, welcher angibt, wie sich die Funktion auf einem beliebigen Argument verhält. Der Name des Arguments ist dabei nur ein Platzhalter, der – so die mathematische Sicht – zur *Abstraktion* der Funktionsbeschreibung benötigt wird. Um diese abstrakte Platzhalterrolle zu kennzeichnen, hat man ursprünglich das Symbol '\' benutzt und geschrieben

```
\x. 2*x+3
```

Dies drückt aus, daß eine Abbildung mit formalem Argument `x` definiert wird, welche als Ergebnis den Wert des Ausdrucks rechts vom Punkt liefert. Später wurde – der leichteren Bezeichnung wegen – das Symbol '\' durch den griechischen Buchstaben λ (*lambda*) ersetzt. In heutiger Notation schreibt man

```
λx. 2*x+3
```

Diese Abstraktion von Ausdrücken über formale Parameter ist eines der beiden fundamentalen Grundkonzepte des λ-Kalküls. Es wird benutzt, um Funktionen zu *definieren*. Natürlich aber will man definierte Funktionen auch auf bestimmte Eingabewerte *anwenden*. Die Notation hierfür ist denkbar einfach: man schreibt einfach das Argument hinter die Funktion und benutzt Klammern, wenn der Wunsch nach Eindeutigkeit dies erforderlich macht. Um also obige Funktion auf den Wert `4` anzuwenden schreibt man einfach:

```
(λx. 2*x+3)(4)
```

Dabei hätte die Klammerung um die `4` auch durchaus entfallen können. Diese Notation – man nennt sie *Applikation* – besagt, daß die Funktion `λx. 2*x+3` auf das Argument `4` angewandt wird. Sie sagt aber nichts darüber aus, welcher Wert bei dieser Anwendung herauskommt. Abstraktion und Applikation sind daher nichts anderes als syntaktische *Beschreibungsformen* für Operationen, die auszuführen sind.

Es hat sich gezeigt, daß durch Abstraktion und Applikation alle Terme gebildet werden können, die man zur Charakterisierung von Funktionen und ihrem Verhalten benötigt. Was bisher aber fehlt, ist die Möglichkeit, Funktionsanwendungen auch *auszuwerten*. Auch dieser Mechanismus ist naheliegend: um eine Funktionsanwendung auszuwerten, muß man einfach die Argumente anstelle der Platzhalter einsetzen. Man berechnet den Wert einer Funktionsanwendung also durch *Reduktion* und erhält

```
2*4+3
```

Bei der Reduktion verschwindet also die Abstraktion `λx` und die Anwendung `(4)` und stattdessen wird im inneren Ausdruck der Platzhalter `4` durch das Argument `4` ersetzt. Diese Operation ist nichts anderes als eine simple Ersetzung von Symbolen durch Terme, also eine *Substitution* im Sinne von Definition 2.2.20 (siehe Seite 37). Damit ist die Auswertung von Termen ein ganz schematischer Vorgang, der sich durch eine einfache Symbolmanipulation beschreiben läßt.

$$\texttt{(}\lambda\texttt{x. 2*x+3)(4)} \longrightarrow \texttt{(2*x+3)}\ [\texttt{x/4}] = \texttt{2*4+3}.$$

Anders als Abstraktion und Applikation ist diese durch das Symbol $\longrightarrow$ gekennzeichnete Reduktion kein Mechanismus um Terme zu bilden, sondern einer um Terme in andere Terme umzuwandeln, welche im Sinne der vorgesehenen Bedeutung den gleichen Wert haben. Reduktion ist also eine Konversionsregel im Sinne von Abschnitt 2.1.6 (Seite 15).

Durch Abstraktion, Applikation und Reduktion ist der λ-Kalkül vollständig charakterisiert. Weitere Operationen sind nicht erforderlich. So können wir uns darauf konzentrieren, präzise Definitionen für diese Konzepte zu liefern und einen Kalkül zu erstellen, der es ermöglicht, Aussagen über den *Wert* eines λ-Terms zu beweisen. Der λ-Kalkül erlaubt symbolisches Rechnen auf Termen und geht somit weit über die Möglichkeiten der Prädikatenlogik hinaus.[42]

Anders als diese gibt der λ-Kalkül jedoch eine *intensionale* Sicht auf Funktionen. λ-Terme beschreiben uns die *innere* Struktur von Funktionen, nicht aber ihr äußeres (*extensionales*) Verhalten. Im λ-Kalkül werden Funktionen als eine *Berechnungsvorschrift* angesehen. Diese erklärt, wie der Zusammenhang zwischen dem Argument einer Funktion und ihrem Resultat zu bestimmen ist, nicht aber, welche mengentheoretischen *Objekte* hinter einem Term stehen. Der λ-Kalkül ist also eine Art *Logik der Berechnung*.

Eine weitere Änderung gegenüber der Prädikatenlogik erster Stufe ist, daß im λ-Kalkül Funktionen selbst wieder Argumente von anderen Funktionen sein dürfen. Ausdrücke wie `(λf.λx. f(x))(λx. 2*x)` werden im λ-Kalkül durchaus sehr häufig benutzt (man könnte fast nichts beschreiben ohne sie), während sie in der Prädikatenlogik erster Stufe verboten sind. In diesem Sinne ist der λ-Kalkül eine *Logik höherer Ordnung*.

Aus der Berechnungsvorschrift des λ-Kalküls ergibt sich unmmittelbar auch ein logischer Kalkül zum Schließen über den Wert eines λ-Ausdrucks. Logisch betrachtet ist damit der λ-Kalkül ein *Kalkül der Gleichheit* und er bietet ein klar definiertes und einfaches Konzept, um logische Schlußfolgerungen über das Verhalten von Programmen zu ziehen. Es muß nur sichergestellt werden, daß sich die Gleichheit zweier Terme genau dann beweisen läßt, wenn die Berechnungsvorschrift bei beiden zum gleichen Ergebnis führt.

Die Semantik von λ-Ausdrücken mengentheoretisch zu beschreiben ist dagegen verhältnismäßig schwierig, da – anders als bei der Prädikatenlogik – eine konkrete mengentheoretische Semantik bei der Entwicklung des λ-Kalküls keine Rolle spielte. Zwar ist es klar, daß es sich bei λ-Ausdrücken im wesentlichen um Funktionen handeln soll, aber die zentrale Absicht war die Beschreibung der *berechenbaren* Funktionen. Eine *operationale* Semantik, also eine Vorschrift, wie man den Wert eines Ausdrucks ausrechnet, läßt sich daher leicht angeben. Was aber die berechenbaren Funktionen im Sinne der Mengentheorie genau sind, das kann man ohne eine komplexe mathematische Theorie kaum angeben. Es ist daher kein Wunder, daß die (extensionale) Semantik erst lange nach der Entwicklung des Kalküls gegeben werden konnte.

Wir werden im folgenden zuerst die Syntax von λ-Ausdrücken sowie ihre operationale Semantik beschreiben und die mengentheoretische Semantik nur am Schluß kurz skizzieren. Wir werden zeigen, daß der λ-Kalkül

[42] Aus der Sicht der Logiker kann man den λ-Kalkül als einen Mechanismus zur Erzeugung von Funktionszeichen ansehen. Der Term `λx. 2*x` ist *einer* der vielen Namen, die man für die Verdoppelungsfunktion nehmen kann. Gegenüber anderen hat er den Vorteil, daß man ihm die intendierte Bedeutung besser ansehen kann. Aus diesem Gedankengang der *Namens*abstraktion ergab sich übrigens auch das Symbol \ als Kennzeichnung, daß jetzt ein neuer Name generiert wird. Durch die Verwendung von λ-Termen kann in der Logik also das Alphabet $\mathcal{F}$ entfallen.

Der λ-Kalkül hat ansonsten sehr viel mit der Prädikatenlogik gemeinsam. Die Alphabete $\mathcal{F}$, $\mathcal{P}$ und $\mathcal{T}$ entfallen und übrig bleiben nur die Mechanismen zur Verarbeitung von Variablen. Für diese verhält sich der λ-Operator wie ein neuer Quantor, der Variablen bindet. In der Semantik kann man ihn als Mengenabstraktion interpretieren: der Term `λx.x+2` steht für die Menge $\{(x,x+2) \mid x \in \mathcal{U}\}$. Aus dieser Betrachtungsweise sind viele Definitionen wie Syntax, Substitution etc. relativ naheliegend, da sie sehr ähnlich zu denen der Prädikatenlogik sind.

tatsächlich genauso ausdruckstark ist wie jedes andere Berechenbarkeitsmodell und hierfür eine Reihe von Standardoperationen durch λ-Ausdrücke beschreiben. Die Turing-Mächtigkeit des λ-Kalküls hat natürlich auch Auswirkungen auf die Möglichkeiten einer automatischen Unterstützung des Kalkül zum Schließen über die Werte von λ-Ausdrücken. Dies werden wir am Ende dieses Abschnitts besprechen.

2.3.1 Syntax

Die Syntax von λ-Ausdrücken ist sehr leicht zu beschreiben, da es nur Variablen, Abstraktion und Applikation gibt. Stilistisch ist sie so ähnlich wie die Definition der Terme der Prädikatenlogik (Definition 2.2.2, Seite 24)

Definition 2.3.2 (λ-Terme)

Es sei $\mathcal{V}$ ein Alphabet von Variablen(-symbolen)

Die Terme der Sprache des λ-Kalküls – kurz λ-Terme sind induktiv wie folgt definiert.

- *Jede Variable $x \in \mathcal{V}$ ist ein λ-Term.*
- *Ist $x \in \mathcal{V}$ eine Variable und t ein beliebiger λ-Term, dann ist die λ-Abstraktion $\lambda x.t$ ein λ-Term.*
- *Sind t und f beliebige λ-Terme, dann ist die Applikation $f\ t$ ein λ-Term.*
- *Ist t ein beliebiger λ-Term, dann ist (t) ein λ-Term.*

Wie bei der Prädikatenlogik gibt es für die Wahl der Variablennamen im Prinzip keinerlei Einschränkungen bis auf die Tatsache, daß sie sich im Computer darstellen lassen sollten und keine reservierten Symbole wie λ enthalten. Namenskonventionen lassen sich nicht mehr so gut einhalten wie früher, da bei einem Term wie `λx.x` noch nicht feststeht, ob die Variable `x` ein einfaches Objekt oder eine Funktion beschreiben soll. Für die Verarbeitung ist dies ohnehin unerheblich. Wir wollen die Bildung von Termen gemäß der obigen Regeln nun anhand einiger Beispiele veranschaulichen.

Beispiel 2.3.3

Die folgenden Ausdrücke sind korrekte λ-Terme im Sinne von Definition 2.3.2.

`x`, `pair`, `x(x)`, `λf.λg.λx. f g (g x)`, `λf.λx.f(x)`, `(λx.x(x))(λx.x(x))`

Die Beispiele zeigen insbesondere, daß es erlaubt ist, Terme zu bilden, bei denen man Funktionen *höherer Ordnung* – also Funktionen, deren Argumente wiederum Funktionen sind, wie z.B. `f` in `f g (g x)` – assoziiert, und Terme, die *Selbstanwendung* wie in `x(x)` beschreiben. Dies macht die Beschreibung einer mengentheoretischen Semantik (siehe Abschnitt 2.3.6) natürlich sehr viel schwieriger als bei der Prädikatenlogik.

Die Definition läßt es zu, λ-Terme zu bilden, ohne daß hierzu Klammern verwendet werden müssen. Wie bei der Prädikatenlogik erhebt sich dadurch jedoch die Frage nach einer eindeutigen Decodierbarkeit ungeklammerter λ-Terme. Deshalb müssen wir wiederum Prioritäten einführen und vereinbaren, daß der "λ-Quantor" schwächer bindet als die Applikation.

Definition 2.3.4 (Konventionen zur Eindeutigkeit von λ-Termen)

Die Applikation bindet stärker als die λ-Abstraktion und ist linksassoziativ.[43] *In einem λ-Term braucht ein durch einen stärker bindenden Operator gebildeter Teilterm nicht geklammert zu werden.*

Gemäß dieser Konvention ist also der Term `f a b` gleichbedeutend mit `(f(a))(b)` und nicht etwa mit `f(a(b))`. Die Konvention, die Applikation linksassoziativ zu interpretieren, liegt darin begründet, daß hierdurch der Term `f a b` etwa dasselbe Verhalten zeigt wie die Anwendung einer Funktion `f` auf das Paar `(a,b)`. Im zweiten Fall werden die Argumente auf einmal verarbeitet, während sie im λ-Kalkül der Reihe nach abgearbeitet werden. Die Anwendung von `f` auf `a` liefert eine neue Funktion, die wiederum auf `b` angewandt wird. Die Restriktion der λ-Abstraktionen auf *einstellige* Funktionen ist also keine wirkliche Einschränkung.[44]

[43] Eine Assoziativitätsvereinbarung für die Abstraktion braucht – wie bei den logischen Quantoren – nicht getroffen zu werden. `λx.λy.t` ist gleichwertig mit `λx.(λy.t)`, da die alternative Klammersetzung `(λx.λy).t` kein syntaktisch korrekter λ-Term ist.

[44] Die Umwandlung einer Funktion f, die auf Paaren von Eingaben operiert, in eine einstellige Funktion höherer Ordnung F mit der Eigenschaft $F(x)(y) = f(x, y)$ nennt man – in Anlehnung an den Mathematiker Haskell B. Curry – *currying*. Es sei allerdings angemerkt, daß diese Technik auf den Mathematiker Schönfinkel und nicht etwa auf Curry zurückgeht.

2.3.2 Operationale Semantik: Auswertung von Termen

In Beispiel 2.3.1 haben wir bereits angedeutet, daß wir mit jedem λ-Term natürlich eine gewisse Bedeutung assoziieren. Ein Term `λx.2*x` soll eine Funktion beschreiben, die bei Eingabe eines beliebigen Argumentes dieses verdoppelt. Diese Bedeutung wird im λ-Kalkül durch eine *Berechnungsvorschrift* ausgedrückt, welche aussagt, auf welche Art der Wert eines λ-Terms zu bestimmen ist. Diese Vorschrift verwandelt den bisher bedeutungslosen λ-Kalkül in einen Berechnungsformalismus.

Es ist offensichtlich, daß eine Berechnungsregel rein syntaktischer Natur sein muß, denn Rechenmaschinen können ja nur Symbole in andere Symbole umwandeln. Im Falle des λ-Kalküls ist diese Regel sehr einfach: wird die Apllikation einer Funktion $\lambda x\,.\,t$ auf ein Argument s ausgeweetet, so wird der formale Parameter x der Funktion – also die Variablen der λ-Abstraktion – im Funktionskörper t durch das Argument s ersetzt. Der Ausdruck $(\lambda x\,.\,t)(s)$ wird also zu $t[s/x]$ *reduziert*. Um dies präzise genug zu definieren müssen wir die Definitionen der freien und gebundenen Vorkommen von Variablen (vergleiche Definition 2.2.18 auf Seite 36) und der Substitution (Definition 2.2.20 auf Seite 37) entsprechend auf den λ-Kalkül anpassen.

Definition 2.3.5 (Freies und gebundenes Vorkommen von Variablen)

Es seien $x, y \in \mathcal{V}$ Variablen sowie f und t λ-Terme. Das freie und gebundene Vorkommen der Variablen x in einem λ-Term ist induktiv durch die folgenden Bedingungen definiert.

1. *Im λ-Term x kommt x frei vor. Die Variable y kommt nicht vor, wenn x und y verschieden sind.*
2. *In $\lambda x\,.\,t$ wird jedes freie Vorkommen von x in t gebunden. Gebundene Vorkommen von x in t bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn x und y verschieden sind.*
3. *In $f\,t$ bleibt jedes freie Vorkommen von x in f oder t frei und jedes gebundene Vorkommen von x in f oder t gebunden.*
4. *In (t) bleibt jedes freie Vorkommen von x in t frei und jedes gebundene Vorkommen gebunden.*

Das freie Vorkommen der Variablen $x_1, \ldots, x_n$ in einem λ-Term t wird mit $t[x_1, \ldots, x_n]$ gekennzeichnet.

Eine λ-Term ohne freie Variablen heißt geschlossen.

Man beachte, daß im Unterschied zur Prädikatenlogik Variablen jetzt auch innerhalb des "Funktionsnamens" einer Applikation $f\,t$ frei oder gebunden vorkommen sind, da f seinerseits ein komplexerer λ-Term sein kann. Das folgende Beispiel illustriert die Möglichkeiten des freien bzw. gebundenen Vorkommens von Variablen.

Beispiel 2.3.6

Wir wollen das Vorkommen der Variablen `x` im λ-Term `λf.λx.(λz. f x z) x` analysieren

- Die Variable `x` tritt frei auf im λ-Term `x`.
- Nach (3.) ändert sich daran nichts, wenn die λ-Terme `f x` und `f x z` gebildet werden.
- Nach (2.) bleibt `x` frei im λ-Term `λz. f x z` .
- Nach (3.) bleibt `x` frei im λ-Term `(λz. f x z) x` .
- In `λx.(λz. f x z) x` ist `x` gemäß (2.) – von außen betrachtet – gebunden.
- Nach (2.) bleibt `x` gebunden im gesamten Term `λf.λx.(λz. f x z) x`.

Insgesamt haben wir folgende Vorkommen von `x`: $\overbrace{\lambda \texttt{f}.\lambda \texttt{x}.\underbrace{(\lambda \texttt{z. f x z}) \texttt{ x}}_{\texttt{x } \textit{frei}}}^{\texttt{x } \textit{gebunden}}$

Auch das Konzept der Substitution muß geringfügig geändert werden. Während in Definition 2.2.20 nur innerhalb der Argumente einer Funktionsanwendung ersetzt werden konnte, können im λ-Kalkül auch innerhalb der Funktion einer Applikation (die ebenfalls ein Term ist) Ersetzungen vorgenommen werden. Die λ-Abstraktion dagegen verhält sich wie ein Quantor. Entsprechend müssen wir wieder drei Fälle betrachten.

Definition 2.3.7 (Substitution)

Die Anwendung einer Substitution $\sigma=[t/x]$ auf einen λ-Term u – bezeichnet durch $\underline{u[t/x]}$ – ist induktiv wie folgt definiert.

$x[t/x] = t$

$x[t/y] = x$, *wenn x und y verschieden sind.*

$(\lambda x\,.\,u)[t/x] = \lambda x\,.\,u$

$(\lambda x\,.\,u)[t/y] = (\lambda z\,.\,u[z/x])[t/y]$

wenn x und y verschieden sind, y frei in u vorkommt und x frei in t vorkommt. z ist eine neue Variable, die von x und y verschieden ist und weder in u noch in t vorkommt.

$(\lambda x\,.\,u)[t/y] = \lambda x\,.\,u[t/y]$

wenn x und y verschieden sind und es der Fall ist, daß y nicht frei in u ist oder daß x nicht frei in t vorkommt.

$(f\,u)[t/x] = f[t/x]\,u[t/x]$

$(u)[t/x] = (u[t/x])$

Dabei sind $x, y \in \mathcal{V}$ Variablen sowie f,u und t λ-Terme.

Die Anwendung komplexerer Substitutionen auf einen λ-Term, $u[t_1, \ldots, t_n/x_1, \ldots, x_n]$, wird entsprechend definiert. Wir wollen die Substitution an einem einfachen Beispiel erklären.

Beispiel 2.3.8

$$
\begin{aligned}
& (\lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\,[\lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= \ & \lambda \mathtt{f}.\ (\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\,[\lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= \ & \lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\ (\mathtt{n\ f\ (f\ x)})\,[\lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= \ & \lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\ (\lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)}
\end{aligned}
$$

Bei der Anwendung einer Substitution auf eine λ-Abstraktion müssen wir also in besonderen Fällen wiederum auf eine Umbenennung gebundener Variablen zurückgreifen, um zu vermeiden, daß eine freie Variable ungewollt in den Bindungsbereich der λ-Abstraktion gerät. Diese Umbenennung gebundener Variablen ändert die Bedeutung eines Terms überhaupt nicht. Terme, die sich nur in den Namen ihrer gebundenen Variablen unterscheiden, müssen also im Sinne der Semantik als gleich angesehen werden, auch wenn sie textlich verschieden sind. Zur besseren Unterscheidung nennt man solche Termpaare daher *kongruent*.

Definition 2.3.9 (α-Konversion)

Eine $\underline{\text{Umbenennung gebundener Variablen}}$ (oder $\underline{\alpha\text{-Konversion}}$) in einem λ-term t ist die Ersetzung eines Teilterms von t mit der Gestalt $\lambda x\,.\,u$ durch den Term $\lambda z\,.\,u[z/x]$, wobei z eine Variable ist, die in u bisher nicht vorkam.

Zwei λ-Terme t und u heißen $\underline{\text{kongruent}}$ oder $\underline{\alpha\text{-konvertibel}}$, wenn u sich aus t durch endlich viele Umbenennungen gebundener Variablen ergibt.

Umbennenung kann also als eine erste einfache Rechenvorschrift angesehen werden, welche Terme in andere Terme umwandelt (konvertiert), die syntaktisch verschieden aber gleichwertig sind. Eine wirkliche Auswertung eines Termes findet jedoch nur statt, wenn reduzierbare Teilterme durch ihre kontrahierte Form ersetzt werden.

Definition 2.3.10 (Reduktion)

Die $\underline{\text{Reduktion}}$ eines λ-terms t ist die Ersetzung eines Teiltermes von t mit der Gestalt $(\lambda x\,.\,u)(s)$ durch den Term $u[s/x]$. Der Term $(\lambda x\,.\,u)(s)$ wird dabei als $\underline{\text{Redex}}$ bezeichnet und $u[s/x]$ als sein $\underline{\text{Kontraktum}}$.[45]

Ein λ-Terme t heißt $\underline{\text{reduzierbar}}$ auf einen λ-Term u – im Zeichen $\underline{t \xrightarrow{*} u}$ –, wenn u sich aus t durch endlich viele Reduktionen und α-Konversionen ergibt.

[45] Das Wort *Redex* steht für einen reduzierbaren Ausdruck (reducible expression) und *Kontraktum* für die zusammengezogene Form dieses Ausdrucks.

Für die Auswertung von λ-Termen durch einen Menschen würden diese beiden Definitionen vollkommen ausreichen. Da wir den Prozeß der Reduktion jedoch als Berechnungsmechanismus verstehen wollen, den eine Maschine ausführen soll, geben wir zusätzlich eine detaillierte formale Definition.

Definition 2.3.11 (Formale Definition der Reduzierbarkeit)

- *Konversion von λ-Termen ist eine binäre Relation $\cong$, die induktiv wie folgt definiert ist.*
  1. $\lambda x.u \cong \lambda z.u[z/x]$, *falls* z *eine Variable ist, die in* u *nicht vorkommt.* — *α-Konversion*
  2. $f\ t \cong f\ u$, *falls* $t \cong u$ — *μ-Konversion*
  3. $f\ t \cong g\ t$, *falls* $f \cong g$ — *ν-Konversion*
  4. $\lambda x.t \cong \lambda x.u$, *falls* $t \cong u$ — *ξ-Konversion*
  5. $t \cong t$ — *ρ-Konversion*
  6. $t \cong u$, *falls* $u \cong t$ — *σ-Konversion*
  7. $t \cong u$, *falls es einen λ-Term* s *gibt mit* $t \cong s$ *und* $s \cong u$ — *τ-Konversion*
- *Reduktion von λ-Termen ist eine binäre Relation $\longrightarrow$, die induktiv wie folgt definiert ist.*
  1. $(\lambda x.u)\ s \longrightarrow u[s/x]$ — *β-Reduktion*
  2. $f\ t \longrightarrow f\ u$, *falls* $t \longrightarrow u$
  3. $f\ t \longrightarrow g\ t$, *falls* $f \longrightarrow g$
  4. $\lambda x.t \longrightarrow \lambda x.u$, *falls* $t \longrightarrow u$
  5. $t \longrightarrow u$, *falls es einen λ-Term* s *gibt mit* $t \longrightarrow s$ *und* $s \cong u$ *oder* $t \cong s$ *und* $s \longrightarrow u$
- *Reduzierbarkeit von λ-Termen ist eine binäre Relation $\xrightarrow{*}$, die wie folgt definiert ist*

  $t \xrightarrow{*} u$, *falls* $t \xrightarrow{n} u$ *für ein* $n \in \mathbb{N}$.

  Dabei ist die Relation $\xrightarrow{n}$ induktiv wie folgt definiert
  - $t \xrightarrow{0} u$, *falls* $t \cong u$
  - $t \xrightarrow{1} u$, *falls* $t \longrightarrow u$
  - $t \xrightarrow{n+1} u$, *falls es einen λ-Term* s *gibt mit* $t \longrightarrow s$ *und* $s \xrightarrow{n} u$

Die Regeln, die zusätzlich zur zur β-Reduktion (bzw. zur α-Konversion) genannt werden, drücken aus, daß Reduktion auf jedes Redex innerhalb eines Termes angewandt werden darf, um diesen zu verändern. Unter all diesen Regeln ist die β-Reduktion die einzige 'echte' Reduktion. Aus diesem Grunde schreibt man oft auch $t \xrightarrow{\beta} u$ anstelle von $t \longrightarrow u$. Wir wollen nun die Reduktion an einigen Beispielen erklären.

Beispiel 2.3.12

1. `(λn.λf.λx. n f (f x)) (λf.λx.x)` $\longrightarrow$ `(λf.λx. n f (f x))[λf.λx.x / n]`

   `=` `λf.λx. (λf.λx.x) f (f x)` *(vergleiche Beispiel 2.3.8 auf Seite 50)*

   `(λf.λx.x) f` $\longrightarrow$ `λx.x`

   also `(λf.λx.x) f (f x)` $\longrightarrow$ `(λx.x) (f x)`

   also `λx.(λf.λx.x) f (f x)` $\longrightarrow$ `λx.(λx.x) (f x)`

   also `λf.λx.(λf.λx.x) f (f x)` $\longrightarrow$ `λf.λx.(λx.x) (f x)`

   `(λx.x) (f x)` $\longrightarrow$ `f x`

   also `λx.(λx.x) (f x)` $\longrightarrow$ `λx. f x`

   also `λf.λx.(λx.x) (f x)` $\longrightarrow$ `λf.λx. f x`

   Diese ausführliche Beschreibung zeigt alle Details einer formalen Reduktion einschließlich des Hinabsteigens in Teilterme, in denen sich die zu kontrahierenden Redizes befinden. Diese Form ist für eine

Reduktion "von Hand" jedoch zu ausführlich, da es einem Menschen nicht schwerfällt, einen zu reduzierenden Teilterm zu identifizieren und die Reduktion durchzuführen. Wir schreiben daher kurz

```
      (λn.λf.λx. n f (f x)) (λf.λx.x)
⟶     λf.λx. (λf.λx.x) f (f x)
⟶     λf.λx. (λx.x) (f x)
⟶     λf.λx. f x
```

oder noch kürzer: `(λn.λf.λx. n f (f x)) (λf.λx.x)` $\xrightarrow{3}$ `λf.λx. f x`

2.
```
      (λn.λf.λx. n f (f x)) (λf.λx. f x)
⟶     λf.λx. (λf.λx. f x) f (f x)
⟶     λf.λx. (λx. f x) (f x)
⟶     λf.λx. f (f x)
```

3. Bei der Reduktion des Terms `(λf.λx. f x (f f)) (λx.λy. y)` gibt es mehrere Möglichkeiten vorzugehen. Die vielleicht naheliegenste ist die folgende:

```
      (λf.λx. f x (f f)) (λx.λy. y)
⟶     λx. (λx.λy. y) x ((λx.λy. y) (λx.λy. y))
⟶     λx. (λy. y) ((λx.λy. y) (λx.λy. y))
⟶     λx. ((λx.λy. y) (λx.λy. y))
⟶     λx. λy. y)
```

Wir hätten ab dem zweiten Schritt aber auch zunächst das rechte Redex reduzieren können und folgende Reduktionskette erhalten:

```
      (λf.λx. f x (f f)) (λx.λy. y)
⟶     λx. (λx.λy. y) x ((λx.λy. y) (λx.λy. y))
⟶     λx. (λx.λy. y) x (λy. y))
⟶     λx. (λy. y) (λy. y))
⟶     λx. λy. y)
```

Das dritte Beispiel zeigt deutlich, daß Reduktion keineswegs ein deterministischer Vorgang ist. Unter Umständen gibt es mehrere Redizes, auf die eine β-Reduktion angewandt werden kann, und damit auch mehrere Arten, den Wert eines Terms durch Reduktion auszurechnen. Um also den λ-Kalkül als eine Art Programmiersprache verwendbar zu machen, ist es zusätzlich nötig, eine *Reduktionsstrategie* zu fixieren. Diese Strategie muß beim Vorkommen mehrerer Redizes in einem Term entscheiden, welches von diesen zuerst durch ein Kontraktum ersetzt wird. Die Auswirkungen einer solchen Festlegung und weitere Reduktionseigenschaften des λ-Kalküls werden wir im Abschnitt 2.3.7 diskutieren.

2.3.3 Standard-Erweiterungen

Bisher haben wir den λ-Kalkül im wesentlichen als ein formales System betrachtet, welches geeignet ist, Terme zu manipulieren. Wir wollen nun zeigen, daß dieser einfache Formalismus tatsächlich geeignet ist, bekannte berechenbare Funktionen auszudrücken. Dabei müssen wir jedoch beachten, daß die einzige Berechnungsform die Reduktion von Funktionsanwendungen ist. Zahlen, boolesche Werte und ähnliche aus den meisten Programmiersprachen vertraute Operatoren gehören nicht zu den Grundkonstrukten des λ-Kalküls. Wir müssen sie daher im λ-Kalkül *simulieren*.[46]

Aus theoretischer Sicht bedeutet dies, den λ-Kalkül durch Einführung abkürzender Definitionen im Sinne von Abschnitt 2.1.5 *konservativ* zu erweitern. Auf diese Art behalten wir die einfache Grundstruktur des

[46] In *realen* Computersystemen ist dies nicht anders. Auch ein Computer operiert nicht auf Zahlen sondern nur auf Bitmustern, welche Zahlen *darstellen*. Alle arithmetischen Operationen, die ein Computer ausführt, sind nichts anderes als eine Simulation dieser Operationen durch entsprechende Manipulationen der Bitmuster.

λ-Kalküls, wenn es darum geht, grundsätzliche Eigenschaften dieses Kalküls zu untersuchen, erweitern aber gleichzeitig die Ausdruckskraft der vordefinierten formalen Sprache und gewinnen somit an Flexibilität.

Höhere funktionale Programmiersprachen wie LISP oder ML können in diesem Sinne als konservative Erweiterungen des λ-Kalküls betrachtet werden.[47] Sie unterscheiden sich von diesem nur durch eine große Menge vordefinierter Konstrukte, die ein einfacheres Programmieren in diesen Sprachen erst möglich machen.

Boolesche Operatoren

Im reinen λ-Kalkül ist die Funktionsanwendung (Applikation) die einzige Möglichkeit, "Programme" und "Daten" in Verbindung zu bringen. In praktischen Anwendungen besteht jedoch oft die Notwendigkeit, bestimmte Programmteile nur auszuführen, wenn eine Bedingung erfüllt ist. Um dies zu simulieren, benötigt man boolesche Werte und ein Konstrukt zur Erzeugung *bedingter Funktionsaufrufe*.

Definition 2.3.13 (Boolesche Operatoren)

$$\begin{aligned} \mathsf{T} &\equiv \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x} \\ \mathsf{F} &\equiv \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y} \\ \mathsf{cond}(b;s;t) &\equiv b\,s\,t \end{aligned}$$

Die Terme **T** und **F** sollen die Wahrheitswerte *wahr* und *falsch* simulieren. Der Term **cond**$(b;s;t)$ beschreibt ein sogenanntes *Conditional*. Es nimmt einen booleschen Ausdruck b als erstes Argument, wertet diesen aus und berechnet dann – je nachdem ob diese Auswertung **T** oder **F** ergab – den Wert von s oder den von t. Es ist nicht schwer zu beweisen, daß **T**, **F** und cond sich tatsächlich wie die booleschen Operatoren verhalten, die man hinter dem Namen vermutet.

Beispiel 2.3.14

Wir zeigen, daß **cond**$(\mathsf{T};s;t)$ zu s reduziert und **cond**$(\mathsf{F};s;t)$ zu t

$$\begin{aligned} \mathsf{cond}(\mathsf{T};s;t) &\equiv \mathsf{T}\,s\,t \equiv (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x})\,s\,t \longrightarrow (\lambda \mathtt{y}.s)\,t \longrightarrow s \\ \mathsf{cond}(\mathsf{F};s;t) &\equiv \mathsf{F}\,s\,t \equiv (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y})\,s\,t \longrightarrow (\lambda \mathtt{y}.\mathtt{y})\,t \longrightarrow t \end{aligned}$$

Damit ist das Conditional tatsächlich invers zu den booleschen Werten **T** und **F**.[48] Im Beispiel 2.3.25 (Seite 59) werden wir darüber hinaus auch noch zeigen, daß **T** und **F** auch fundamental unterschiedliche Objekte beschreiben und somit tatsächlich dem Gedanken entsprechen, daß **T** und **F** einander widersprechen.

Die bisherige Präsentationsform des Conditionals als Operator cond, der drei Eingabeparameter erwartet, entspricht immer noch der Denkweise von Funktionsdefinition und -anwendung. Sie hält sich daher an die syntaktischen Einschränkungen an den Aufbau von Termen, die uns aus der Prädikatenlogik (vergleiche Definition 2.2.2 auf Seite 24) bekannt und für einen Parser leicht zu verarbeiten ist. Für die meisten Programmierer ist diese Präsentationsform jedoch sehr schwer zu lesen, da sie mit der Schreibweise "if b then s else t" vertrauter sind. Wir ergänzen daher die Definitionen boolescher Operatoren um eine verständlichere Notation für das Conditional.

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \equiv \mathsf{cond}(b;s;t)$$

Im folgenden werden wir für die meisten Operatoren immer zwei Formen definieren: eine mathematische 'Termform' und eine leichter zu lesende "Display Form" dieses Terms.[49]

[47] Man beachte, daß für die die textliche Form der abkürzender Definitionen keinerlei Einschränkungen gelten – bis auf die Forderung, daß sie in einem festen Zeichensatz beschreibbar sind und alle syntaktischen Metavariablen ihrer rechten Seiten auch links vorkommen müssen.

[48] In der Denkweise des Sequenzenkalküls können wir **T** und **F** als Operatoren zur *Einführung* boolescher Werte betrachten und cond als Operator zur *Elimination* boolescher Werte. Diese Klassifizierung von Operatoren – man sagt dazu auch *kanonische* und *nichtkanonische* Operatoren – werden wir im Abschnitt 2.4 weiter aufgreifen.

[49] Im NuPRL System und der Typentheorie, die wir im Kapitel 3 vorstellen, wird diese Idee konsequent zu Ende geführt. Wir unterscheiden dort die sogenannte *Abstraktion* (nicht zu verwechseln mit der λ-Abstraktion), welche einen neuen Operatornamen wie cond einführt, von der *Display Form*, die beschreibt, wie ein Term textlich darzustellen ist, der diesen Operatornamen als Funktionszeichen benutzt. Auf diese Art erhalten wir eine einheitliche Syntaxbeschreibung für eine Vielfalt von Operatoren und dennoch Flexibilität in der Notation.

Paare und Projektionen

Boolesche Operationen wie das Conditional können dazu benutzt werden, 'Programme' besser zu strukturieren. Aber auch bei den 'Daten' ist es wünschenswert, Strukturierungsmöglichkeiten anzubieten. Die wichtigste dieser Strukturierungsmöglichkeiten ist die Bildung von Tupeln, die es uns erlauben, `f(a,b,c)` anstelle von `f a b c` zu schreiben. Auf diese Art wird deutlicher, daß die Werte `a, b` und `c` zusammengehören und nicht etwa einzeln abgearbeitet werden sollen.

Die einfachste Form von Tupeln sind Paare, die wir mit $\langle \mathtt{a},\mathtt{b} \rangle$ bezeichnen. Komplexere Tupel können durch das Zusammensetzen von Paaren gebildet werden. `(a,b,c)` läßt sich zum Beispiel als $\langle \mathtt{a},\langle \mathtt{b},\mathtt{c} \rangle\rangle$ schreiben. Neben der *Bildung* von Paaren aus einzelnen Komponenten benötigen wir natürlich auch Operatoren, die ein Paar p *analysieren*. Üblicherweise verwendet man hierzu *Projektionen*, die wir mit $p.\mathsf{1}$ und $p.\mathsf{2}$ bezeichnen.

Definition 2.3.15 (Operatoren auf Paaren)

$$
\begin{array}{ll}
\mathsf{pair}(s,t) & \equiv \lambda \mathtt{p}.\ \mathtt{p}\, s\, t \\
\mathsf{pi1}(pair) & \equiv pair\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}) \\
\mathsf{pi2}(pair) & \equiv pair\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}) \\
\mathsf{spread}(pair; x,y.t) & \equiv pair\ (\lambda x.\lambda y.t) \\
\langle s,t \rangle & \equiv \mathsf{pair}(s,t) \\
pair.\mathsf{1} & \equiv \mathsf{pi1}(pair) \\
pair.\mathsf{2} & \equiv \mathsf{pi2}(pair) \\
\mathsf{let}\ \langle x,y \rangle = pair\ \mathsf{in}\ t & \equiv \mathsf{spread}(pair; x,y.t)
\end{array}
$$

Der spread-Operator beschreibt eine *einheitliche* Möglichkeit, Paare zu analysieren: ein Paar p wird aufgespalten in zwei Komponenten, die wir mit x und y bezeichnen und in einem Term t weiter verarbeiten. [50] Die Projektionen können als Spezialfall des spread-Operators betrachtet werden, in denen der Term t entweder x oder y ist.

In seiner ausführlicheren Notation $\mathsf{let}\ \langle x,y \rangle = p\ \mathsf{in}\ t$ wird dieses Konstrukt zum Beispiel in der Sprache ML benutzt. Bei seiner Ausführung wird der Term p ausgewertet, bis seine beiden Komponenten feststehen, und diese dann anstelle der Variablen x und y im Term t eingesetzt. Das folgende Beispiel zeigt, daß genau dieser Effekt durch die obige Definition erreicht wird.

Beispiel 2.3.16

$$
\begin{array}{lcll}
\mathsf{let}\ \langle x,y \rangle = \langle u,v \rangle\ \mathsf{in}\ t & \equiv & \langle u,v \rangle(\lambda x.\lambda y.t) & \equiv\ (\lambda \mathtt{p}.\ \mathtt{p}\, u\, v)(\lambda x.\lambda y.t) \\
 & \longrightarrow & (\lambda x.\lambda y.t)\, u\, v & \\
 & \longrightarrow & (\lambda y.t[u/x])\, u\, v & \\
 & \longrightarrow & t[u,v/x,y]) &
\end{array}
$$

Auf ähnliche Weise kann man zeigen $\langle u,v \rangle.\mathsf{1} \longrightarrow u$ und $\langle u,v \rangle.\mathsf{2} \longrightarrow v$.

Damit ist der spread-Operator also tatsächlich invers zur Paarbildung.

Natürliche Zahlen

Es gibt viele Möglichkeiten, natürliche Zahlen und Operationen auf Zahlen im λ-Kalkül darzustellen. Die bekannteste dieser Repräsentationen wurde vom Mathematiker Alonzo Church entwickelt. Man nennt die entsprechenden λ-Terme daher auch *Church Numerals*. In dieser Repräsentation codiert man eine natürliche Zahl n durch einen λ-Term, der zwei Argumente `f` und `x` benötigt und das erste insgesamt n-mal auf das zweite anwendet. Wir bezeichnen diese Darstellung einer Zahl n durch einen λ-Term mit $\overline{n}$. Auf diese Art wird eine Verwechselung der *Zahl* n mit ihrer Darstellung als *Term* vermieden.

[50] Aus logischer Sicht sind x und y zwei Variablen, deren Vorkommen in t durch den spread-Operator gebunden wird. Zusätzlich wird festgelegt, daß x an die erste Komponente des Paares p gebunden wird und y an die zweite.

Definition 2.3.17 (Darstellung von Zahlen durch Church Numerals)

$$
\begin{array}{lll}
f^n\ t & \equiv & \underbrace{f\ (f\ \ldots(f\,t)\ \ldots)}_{n\text{-}mal} \\
\overline{n} & \equiv & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x} \\
\mathsf{s} & \equiv & \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)} \\
\mathsf{add} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ f\ (n\ f\ x)} \\
\mathsf{mul} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ (n\ f)\ x} \\
\mathsf{exp} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ m\ f\ x} \\
\mathsf{zero} & \equiv & \lambda \mathtt{n}.\ \mathtt{n}\ (\lambda \mathtt{n}.\mathsf{F})\ \mathsf{T} \\
\mathsf{p} & \equiv & \lambda \mathtt{n}.\ (\mathtt{n}\ (\lambda \mathtt{fx}.\ \langle \mathsf{s}, \mathsf{let}\ \langle \mathtt{f},\mathtt{x}\rangle = \mathtt{fx}\ \mathsf{in}\ \mathtt{f\ x}\rangle)\ \langle \lambda \mathtt{z}.\overline{0}, \overline{0}\rangle).2 \\
\mathsf{PRs}[\mathit{base},h] & \equiv & \lambda \mathtt{n}.\ \mathtt{n}\ h\ \mathit{base}
\end{array}
$$

Die Terme **s**, **add**, **mul**, **exp**, **p** und **zero** repräsentieren die Nachfolgerfunktion, Addition, Multiplikation, Exponentiation, Vorgängerfunktion und einen Test auf Null. **PRs** ist eine einfache Form der Primitiven Rekursion. Diese Repräsentationen hängen natürlich von der Zahlendarstellung durch Church Numerals ab.

So muß die *Nachfolgerfunktion* **s** dafür sorgen, daß bei Eingabe eines Terms $\overline{n}$=λf.λx. $\mathtt{f}^n\,\mathtt{x}$ das erste Argument f einmal öfter als bisher auf das zweite Argument x angewandt wird. Dies wird dadurch erreicht, daß durch geschickte Manipulationen das zweite Argument durch (f x) ausgetauscht wird. Bei der Simulation der *Addition* benutzt man die Erkenntnis, daß $\mathtt{f}^{m+n}\,\mathtt{x}$ identisch ist mit $\mathtt{f}^m\ (\mathtt{f}^n\,\mathtt{x})$ und ersetzt entsprechend das zweite Argument von $\overline{m}$=λf.λx. $\mathtt{f}^m\,\mathtt{x}$ durch $\mathtt{f}^n\,\mathtt{x}$. Bei der *Multiplikation* muß man – gemäß der Erkenntnis $\mathtt{f}^{m*n}\,\mathtt{x} = (\mathtt{f}^m)^n\,\mathtt{x}$ – das erste Argument modifizieren und bei der *Exponentiation* muß man noch einen Schritt weitergehen. Der *Test auf Null* ist einfach, da $\overline{0}$=λf.λx. x ist. Angewandt auf (λn.**F**) und **T** liefert dies **T** während jedes andere Church Numeral die Funktion (λn.**F**) auswertet, also **F** liefert.

Die *Vorgängerfunktion* ist verhältnismäßig kompliziert zu simulieren, da wir bei Eingabe des Terms der Form $\overline{n}$=λf.λx. $\mathtt{f}^n\,\mathtt{x}$ eine Anwendung von f *entfernen* müssen. Dies geht nur dadurch, daß wir den Term komplett neu aufbauen. Wir tun dies, indem wir den Term λf.λx. $\mathtt{f}^n\,\mathtt{x}$ schrittweise abarbeiten und dabei die Nachfolgerfunktion **s** mit jeweils einem Schritt Verzögerung auf $\overline{0}$ anwenden. Bei der Programmierung müssen wir hierzu ein Paar ⟨f,x⟩ verwalten, wobei x der aktuelle Ausgabewert ist und f die *im nächsten Schritt* anzuwendende Funktion. Startwert ist also $\overline{0}$ für x und λz.$\overline{0}$ für f, da im ersten Schritt ja ebenfalls $\overline{0}$ als Ergebnis benötigt wird. Ab dann wird für x immer der bisherige Wert benutzt und **s** für f.

Wir wollen am Beispiel der Nachfolgerfunktion und der Addition zeigen, daß die hier definierten Terme tatsächlich das Gewünschte leisten.

Beispiel 2.3.18

Es sei $n \in \mathbb{N}$ eine beliebige natürliche Zahl. Wir zeigen, daß **s** $\overline{n}$ tatsächlich reduzierbar auf $\overline{n+1}$ ist.

$$
\begin{array}{lll}
\mathsf{s}\ \overline{n} & \equiv & (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f}\ (\mathtt{f\ x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ (\mathtt{f\ x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\ (\mathtt{f\ x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{n+1}\ \mathtt{x} \qquad \equiv \quad \overline{n+1}
\end{array}
$$

Es seien $m,n \in \mathbb{N}$ beliebige natürliche Zahlen. Wir zeigen, daß **add** $\overline{m}$ $\overline{n}$ reduzierbar auf $\overline{m+n}$ ist.

$$
\begin{array}{lll}
\mathsf{add}\ \overline{m}\ \overline{n} & \equiv & (\lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ f\ (n\ f\ x)})\ \overline{m}\ \overline{n} \\
& \longrightarrow & (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \overline{m}\ \mathtt{f\ (n\ f\ x)})\ \overline{n} \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \overline{m}\ \mathtt{f\ (}\overline{n}\ \mathtt{f\ x)} \qquad \equiv \quad \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ \mathtt{f}\ (\overline{n}\ \mathtt{f\ x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ (\overline{n}\ \mathtt{f\ x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\ (\overline{n}\ \mathtt{f\ x}) \qquad \equiv \quad \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\ ((\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f\ x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\ ((\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\ (\mathtt{f}^n\,\mathtt{x}) \\
& \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{m+n}\,\mathtt{x} \qquad \equiv \quad \overline{m+n}
\end{array}
$$

Ähnlich leicht kann man zeigen, daß **mul** die Multiplikation, **exp** die Exponentiation und **zero** einen Test auf Null repräsentiert. Etwas schwieriger ist die Rechtfertigung der Vorgängerfunktion **p**. Die Rechtfertigung von **PRs** läßt sich durch Abwandlung der Listeninduktion aus Beispiel 2.3.20 erreichen.

Listen

Endliche Listen von Termen lassen sich mathematisch als eine Erweiterung des Konzepts natürlicher Zahlen ansehen. Bei Zahlen startet man mit der Null und kann jede weitere Zahl dadurch bilden, daß man schrittweise die Nachfolgerfunktion **s** anwendet. Bei endlichen Listen startet man enstprechend mit einer leeren (null-elementige) Liste – bezeichnet durch `[]` – und bildet weitere Listen dadurch, daß man schrittweise Elemente $\mathtt{a}_i$ vor die bestehende Liste `L` anhängt. Die entsprechende Operation – bezeichnet durch $\mathtt{a}_i\mathtt{.L}$ – wird durch eine Funktion **cons** – das Gegenstück zur Nachfolgerfunktion **s** ausgeführt.

Definition 2.3.19 (Operatoren auf Listen)

$$\begin{array}{ll}
\mathtt{[]} & \equiv \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x} \\
\mathbf{cons}(t,list) & \equiv \lambda\mathtt{f}.\lambda\mathtt{x}.\,\mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}) \\
t.list & \equiv \mathbf{cons}(t,list) \\
\mathbf{list_ind}[base,h] & \equiv \lambda\mathtt{list}.\ \mathtt{list}\ h\ base
\end{array}$$

Die leere Liste ist also genauso definiert wie die Repräsentation der Zahl 0 während der Operator cons nun die Elemente der Liste – jeweils getrennt durch die Variable `f` – nebeneinanderstellt. Die Liste $\mathtt{a}_1\mathtt{.a}_2\ \ldots\mathtt{a}_n$ wird also dargestellt durch den Term

$$\lambda\mathtt{f}.\lambda\mathtt{x}.\,\mathtt{f}\ \mathtt{a}_1\ (\mathtt{f}\ \mathtt{a}_2\ \ldots(\mathtt{f}\ \mathtt{a}_n\ \mathtt{x})\ \ldots)$$

Die Induktion auf Listen $\mathbf{list\_ind}[base,h]$ ist die entsprechende Erweiterung der einfachen primitiven Rekursion. Ihr Verhalten beschreibt das folgende Beispiel.

Beispiel 2.3.20

Es sei f definiert durch $f{\equiv}\mathbf{list\_ind}[base,h]$. Wir zeigen, daß f die Rekursionsgleichungen

$$f(\mathtt{[]}) = base \quad \text{und} \quad f(t.list) = h\ (t)\ (f(list))$$

erfüllt. Im Basisfall ist dies relativ einfach

$$\begin{array}{lcll}
f(\mathtt{[]}) & \equiv & \mathbf{list_ind}[base,h]\ \mathtt{[]} & \equiv\ (\lambda\mathtt{list}.\ \mathtt{list}\ h\ base)\ \mathtt{[]} \\
& \longrightarrow & \mathtt{[]}\ h\ base & \equiv\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x})\ h\ base \\
& \longrightarrow & (\lambda\mathtt{x}.\mathtt{x})\ base & \\
& \longrightarrow & base &
\end{array}$$

Schwieriger wird es im Rekursionsfall:

$$\begin{array}{lcll}
f(t.list) & \equiv & \mathbf{list_ind}[base,h]\ t.list & \equiv\ (\lambda\mathtt{list}.\ \mathtt{list}\ h\ base)\ t.list \\
& \longrightarrow & t.list\ h\ base & \equiv\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\,\mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}))\ h\ base \\
& \longrightarrow & (\lambda\mathtt{x}.\,h\ t\ (list\ h\ \mathtt{x}))\ base & \\
& \longrightarrow & h\ t\ (list\ h\ base) &
\end{array}$$

Dies ist offensichtlich nicht der gewünschte Term. Wir können jedoch zeigen, daß sich $h\ (t)\ (f(list))$ auf denselben Term reduzieren läßt.

$$\begin{array}{lcll}
h\ (t)\ (f(list)) & \equiv & h\ t\ (\mathbf{list_ind}[base,h]\ list) & \equiv\ h\ t\ ((\lambda\mathtt{list}.\ \mathtt{list}\ h\ base)\ list) \\
& \longrightarrow & h\ t\ (list\ h\ base) &
\end{array}$$

Die Rekursionsgleichungen von f beziehen sich also auf *semantische Gleichheit* und nicht etwa darauf, daß die linke Seite genau auf die rechte reduziert werden kann.

Rekursion

Die bisherigen Konstrukte erlauben uns, bei der Programmierung im λ-Kalkül Standardkonstrukte wie Zahlen, Tupel und Listen sowie eine bedingte Funktionsaufrufe zu verwenden. Uns fehlt nur noch eine Möglichkeit *rekursive Funktionsaufrufe* – das Gegenstück zur Schleife in imperative Programmiersprachen – auf einfache Weise zu beschreiben. Wir benötigen also einen Operator, der es uns erlaubt, eine Funktion f durch eine rekursive Gleichung der Form

$$f(x) = t[f,x]^{51}$$

zu definieren, also durch eine Gleichung, in der f auf beiden Seiten vorkommt. Diese Gleichung an sich beschreibt aber noch keinen Term, sondern nur eine *Bedingung*, die ein Term zu erfüllen hat, und ist somit nicht unmittelbar für die Programmierung zu verwenden. Glücklicherweise gibt es jedoch ein allgemeines Verfahren, einen solchen Term direkt aus einer rekursiven Gleichung zu erzeugen. Wenn wir nämlich in der obigen Gleichung den Term t durch die Funktion $T \equiv \lambda f.\lambda x.t[f,x]$ ersetzen, so können wir die Rekursionsgleichung umschreiben als $f(x) = T\ f\ x$ bzw. als

$$f = T\ f$$

Eine solche Gleichung zu lösen, bedeutet, einen *Fixpunkt* der Funktion T zu bestimmen, also ein Argument f, welches die Funktion T in sich selbst abbildet. Einen Operator R, welcher für beliebige Terme (d.h. also Funktionsgleichungen) deren Fixpunkt bestimmt, nennt man *Fixpunktkombinator* (oder *Rekursor*).

Definition 2.3.21 (Fixpunktkombinator)

Ein Fixpunktkombinator ist ein λ-Term R mit der Eigenschaft, daß für jeden beliebigen λ-Term T die folgende Gleichung erfüllt ist:

$$R\ T = T(R\ T))$$

Ein Fixpunktkombinator R liefert bei Eingabe eines beliebigen Terms T also eine Funktion $f = R(T)$, für welche die rekursive Funktionsgleichung $f = T\,f$ erfüllt ist.

Natürlich entsteht die Frage, ob solche Fixpunktkombinatoren überhaupt existieren. Für den λ-Kalkül kann diese Frage durch die Angabe konkreter Fixpunktkombinatoren positiv beantwortet werden. Der bekannteste unter diesen ist der sogenannte **Y**-Kombinator.

Definition 2.3.22 (Der Fixpunktkombinator Y)

$$\mathbf{Y} \equiv \lambda \texttt{f.}\,(\lambda \texttt{x.}\ \texttt{f}\ (\texttt{x}\ \texttt{x}))\,(\lambda \texttt{x.}\ \texttt{f}\ (\texttt{x}\ \texttt{x}))$$
$$\mathsf{letrec}\ f(x) = t \equiv \mathbf{Y}(\lambda f.\lambda x.t)$$

Lemma 2.3.23

Y *ist ein Fixpunktkombinator*

Beweis: Wie im Falle der Listeninduktion (Beispiel 2.3.20) ergibt sich die Gleichung $\mathbf{Y}(t) = t(\mathbf{Y}(t))$ nur dadurch, daß $\mathbf{Y}(t)$ und $t(\mathbf{Y}(t))$ auf denselben Term reduziert werden können.[52]

$$\begin{aligned}
\mathbf{Y}\ t &\equiv \lambda \texttt{f.}\,(\lambda \texttt{x.}\ \texttt{f}\ (\texttt{x}\ \texttt{x}))\,(\lambda \texttt{x.}\ \texttt{f}\ (\texttt{x}\ \texttt{x}))\ t \\
&\longrightarrow (\lambda \texttt{x.}\ t\ (\texttt{x}\ \texttt{x}))\ (\lambda \texttt{x.}\ t\ (\texttt{x}\ \texttt{x})) \\
&\longrightarrow t\ (\ (\lambda \texttt{x.}\ t\ (\texttt{x}\ \texttt{x}))\ (\lambda \texttt{x.}\ t\ (\texttt{x}\ \texttt{x}))\ ) \\
t\ (\mathbf{Y}\ t) &\equiv t\ (\lambda \texttt{f.}\,(\lambda \texttt{x.}\ \texttt{f}\ (\texttt{x}\ \texttt{x}))\,(\lambda \texttt{x.}\ \texttt{f}\ (\texttt{x}\ \texttt{x}))\ t) \\
&\longrightarrow t\ (\ (\lambda \texttt{x.}\ t\ (\texttt{x}\ \texttt{x}))\ (\lambda \texttt{x.}\ t\ (\texttt{x}\ \texttt{x}))\ )
\end{aligned}$$

□

Fixpunktkombinatoren wie **Y** erzeugen also aus jeder beliebigen rekursiven Funktionsgleichung der Form $f(x)$ = t einen λ-Term, welcher – wenn eingesetzt für f – diese Gleichung erfüllt. Fixpunktkombinatoren können also zur "Implementierung" rekursiver Funktionen eingesetzt werden. Man beachte aber, daß der Ausdruck $\mathsf{letrec}\ f(x) = t$ einen Term beschreibt und nicht etwa eine Definition ist, die den *Namen* f mit diesem Term verbindet.

[51] $t[f,x]$ ist ein Term, in dem die Variablen f und x frei vorkommen (vergleiche Definition 2.2.18 auf Seite 36).

[52] Ein Fixpunktkombinator, der sich auf seinen Fixpunkt reduzieren läßt, ist $(\lambda \texttt{x.}\lambda \texttt{y.}\ \texttt{y}\ (\texttt{x}\ \texttt{x}\ \texttt{y}))\ (\lambda \texttt{x.}\lambda \texttt{y.}\ \texttt{y}\ (\texttt{x}\ \texttt{x}\ \texttt{y}))$

$\Gamma \vdash f\,t = g\,u$ **by** `apply_eq` $\Gamma \vdash f = g$ $\Gamma \vdash t = u$	$\Gamma \vdash \lambda x.t = \lambda y.u$ **by** `lambda_eq`* $\Gamma,\ x'{:}\mathtt{U} \vdash t[x'/x] = u[x'/y]$
$\Gamma \vdash (\lambda x.u)\,s = t$ **by** `reduction` $u[s/x] = t$	
$\Gamma \vdash t{=}t$ **by** `reflexivity`	$\Gamma \vdash t_1{=}t_2$ **by** `symmetry` $\Gamma \vdash t_2{=}t_1$
$\Gamma \vdash t_1{=}t_2$ **by** `transitivity` u $\Gamma \vdash t_1{=}u$ $\Gamma \vdash u{=}t_2$	

*: *Die Umbenennung $[x'/x]$ erfolgt, wenn x in Γ frei vorkommt.*

Abbildung 2.7: Sequenzenkalkül für die Gleichheit von λ-Termen

2.3.4 Ein Kalkül zum Schließen über Berechnungen

Bisher haben wir uns im wesentlichen mit mit der Auswertung von λ-Termen beschäftigt. Die Reduktion von λ-Termen dient dazu, den Wert eines gegebenen Termes zu bestimmen. Zum Schließen über Programme und ihr Verhalten reicht dies jedoch nicht ganz aus, da wir nicht nur Terme untersuchen wollen, die – wie 4+5 und 9 – aufeinander reduzierbar sind, sondern auch Terme, die – wie 4+5 und 2+7 – den gleichen Wert haben. Mit den bisher eingeführten Konzepten läßt sich Werte-Gleichheit relativ leicht definieren.

Definition 2.3.24 (Gleichheit von λ-Termen)

Zwei λ-Terme heißen (semantisch) gleich (konvertierbar), wenn sie auf denselben λ-Term reduziert werden können:

$t{=}u$ gilt genau dann, wenn es einen Term v gibt mit $t \xrightarrow{*} v$ und $u \xrightarrow{*} v$.

Man beachte, daß Werte-Gleichheit weit mehr ist als nur syntaktische Gleichheit und daß sich das Gleichheitssymbol = auf diese Werte-Gleichheit bezieht.

Da wir für die Reduktion von λ-Termen in Definition 2.3.11 bereits eine sehr präzise operationalisierbare Charakterisierung angegeben haben, ist es nunmehr nicht sehr schwer, einen Kalkül aufzustellen, mit dem wir formale Schlüsse über die Resultate von Berechnungen – also die Gleichheit zweier λ-Terme – ziehen können. Wir formulieren hierzu die "Regeln" der Definition 2.3.11 im Stil des Sequenzenkalküls und ergänzen eine Regel für Symmetrie.[53]

Abbildung 2.7 faßt die Regeln des Sequenzenkalküls für die Gleichheit von λ-Termen zusammen. Sie ergänzen die bekannten Gleichheitsregeln der Prädikatenlogik (siehe Abschnitt 2.2.7) um die β-Reduktion und zwei Regeln zur Dekomposition von λ-Termen. Letztere machen die Substutionsregel innerhalb des reinen λ-Kalküls überflüssig.

Die Regel `lambda_eq` verdient besondere Beachtung, da ihre Formulierung auch die α-Konversion beinhaltet. Zwei λ-Abstraktionen $\lambda x.t$ und $\lambda y.u$ sind gleich, die Terme t und u nach einer Umbennenung der gebundenen Variablen x und y zu einer gemeinsamen Abstraktionsvariablen gleich sind. Auch hier gilt eine Art Eigenvariablenbedingung: eine Umbenennung ist in jedem Fall erforderlich, wenn die Abstraktionsvariable (des ersten Terms) bereits in der Hypothesenliste vorkommt.

Mit dem folgenden Beispiel wollen wir nun zeigen, daß die booleschen Werte **T** und **F** tatsächlich 'gegenteilige' Objekte beschreiben. Wären sie nämlich gleich, dann würde folgen, daß *alle* λ-Terme gleich sind.

[53] Der Beweisbegriff ergibt sich unmittelbar aus dem der Prädikatenlogik (siehe Definition 2.2.12 auf Seite 30). Die einzige Änderung besteht darin, daß die Konklusion nun immer eine Gleichheitsformel ist. Eine vollständige Definition werden wir erst im Kapitel 3 für die Typentheorie aufstellen, welche alle Kalküle dieses Kapitels umfaßt.

Beispiel 2.3.25

Es seien s und t beliebige λ-Terme. Ein Beweis für $s{=}t$ könnte wie folgt verlaufen:

```
⊢  s=t
by transitivity  if F then s else t
|\
| ⊢ s = if F then s else t
| by symmetry
|   \
|    ⊢ if F then s else t = s
|    by transitivity  if T then s else t
|    |\
|    | ⊢ if F then s else t =  if T then s else t
|    | by apply_eq        (das schließt eine Auflösung der Definitionen mit ein)
|    | |\
|    | | ⊢ F s  = T s
|    | | by apply_eq
|    | | |\
|    | | | ⊢ [F = T]
|    | | |  \
|    | | |   ⊢ s = s
|    | | |    by reflexivity
|    | |  \
|    | |   ⊢ t = t
|    | |   by reflexivity
|    |  \
|    |   ⊢ if T then s else t  = s
|    |   by reduction
|    |    \
|    |     ⊢ (λy.s) t = s
|    |     by reduction
|    |      \
|    |       ⊢ s = s
|    |       by reflexivity
|  \
|   if F then s else t = t
|   by reduction
|    \
|     ⊢ (λy.y) t = t
|     by reduction
|      \
|       ⊢ t = t
|       by reflexivity
```

Dieses Beispiel gibt uns auch eine Handhabe, wie wir die Ungleichheit zweier Terme s und t beweisen können. Wenn wir mit den Regeln aus Abbildung 2.7 zeigen können, daß aus $s{=}t$ die Gültigkeit von **F = T** folgt, dann wissen wir, daß s und t nicht gleich sein können. Diese Erkenntnis ist allerdings nicht direkt im Kalkül enthalten.

2.3.5 Die Ausdruckskraft des λ-Kalküls

Zu Beginn dieses Abschnitts haben wir die Einführung des λ-Kalküls als Formalismus zum Schließen über Berechnungen damit begründet, daß der λ-Kalkül das einfachste aller Modelle zur Erklärung von Berechenbarkeit ist. Wir wollen nun zeigen, daß der λ-Kalkül *Turing-mächtig* ist, also genau die Klasse der rekursiven Funktionen beschreiben kann. Gemäß der Church'schen These ist er damit in der Lage, jede berechenbare Funktion auszudrücken.

Wir machen uns bei diesem Beweis zunutze, daß Turingmaschinen, imperative Programmiersprachen, μ-rekursive Funktionen etc. bekanntermaßen äquivalent sind.[54] Der Begriff der Berechenbarkeit wird dabei in allen Fällen auf den natürlichen Zahlen abgestützt. Um also einen Vergleich durchführen zu können, definieren wir zunächst die λ-Berechenbarkeit von Funktionen auf natürlichen Zahlen.

[54] Diese Tatsache sollte aus einer Grundvorlesung über theoretische Informatik bekannt sein.

Definition 2.3.26 (λ-Berechenbarkeit)

Eine (möglicherweise partielle) Funktion $f{:}\mathbb{N}^n \to \mathbb{N}$ heißt λ-berechenbar, wenn es einen λ-Term t gibt mit der Eigenschaft, daß für alle $x_1, ..., x_n, m \in \mathbb{N}$ gilt:

$$f(x_1, ..., x_n) = m \quad \text{genau dann, wenn} \quad t\ \overline{x_1}\ \ldots \overline{x_n}\ = \overline{m}$$

Es ist leicht einzusehen, daß man λ-berechenbare Funktionen programmieren kann. Um die Umkehrung zu beweisen – also die Behauptung, daß jede rekursive Funktion auch λ-berechenbar ist – zeigen wir, daß jede μ-rekursive Funktion im λ-Kalkül simuliert werden kann. Wir fassen zu diesem Zweck die Definition der μ-rekursiven Funktionen kurz zusammen.

Definition 2.3.27 (μ-rekursive Funktionen)

Die Klasse der μ-rekursiven Funktionen ist induktiv durch die folgenden Bedingungen definiert.

1. *Alle Konstanten $0, 1, 2, \ldots \in \mathbb{N}$ sind (nullstellige) μ-rekursive Funktionen.*
2. *Die Nullfunktion $z : \mathbb{N} \to \mathbb{N}$ – definiert durch $z(n) = 0$ für alle $n \in \mathbb{N}$ – ist μ-rekursiv.*
3. *Die Nachfolgerfunktion $s : \mathbb{N} \to \mathbb{N}$ – definiert durch $s(n) = n + 1$ für alle $n \in \mathbb{N}$ – ist μ-rekursiv.*
4. *Die Projektionsfunktionen $pr^n_m : \mathbb{N}^n \to \mathbb{N}$ $(m \leq n)$ – definiert durch $pr^n_m(x_1, ..., x_n) = x_m$ für alle $x_1, ..., x_n \in \mathbb{N}$ – sind μ-rekursiv.*
5. *Die Komposition $Cn[f, g_1...g_n]$ der Funktionen $f, g_1...g_n$ ist μ-rekursiv, wenn $f, g_1...g_n$ μ-rekursive Funktionen sind. Dabei ist $Cn[f, g_1...g_n]$ die eindeutig bestimmte Funktion h, für die gilt*
   $$h(\vec{x}) = f(\, g_1(\vec{x}), ..., g_n(\vec{x})\,)^{55}$$
6. *Die primitive Rekursion $Pr[f, g]$ zweier Funktionen f und g ist μ-rekursiv, wenn f und g μ-rekursiv sind. Dabei ist $Pr[f, g]$ die eindeutig bestimmte Funktion h, für die gilt*
   $$h(\vec{x}, 0) = f(\vec{x}) \quad \text{und} \quad h(\vec{x}, y + 1) = g(\vec{x}, y, h(\vec{x}, y))$$
7. *Die Minimierung $Mn[f]$ einer Funktion f ist μ-rekursiv, wenn f μ-rekursiv ist. Dabei ist $Mn[f]$ die eindeutig bestimmte Funktion h, für die gilt*
   $$h(\vec{x}) = \begin{cases} min\{y \mid f(\vec{x}, y) = 0\} & \text{falls dies existiert und alle } f(\vec{x}, i), i < y \text{ definiert sind} \\ undefiniert & \text{sonst} \end{cases}$$

Aufgrund der konservativen Erweiterungen des λ-Kalküls, die wir im Abschnitt 2.3.3 gegeben haben, ist es nun nicht mehr schwer, den Äquivalenzbeweis zu führen.

Satz 2.3.28

Die Klasse der λ-berechenbaren Funktionen ist identisch mit der Klasse der μ-rekursiven Funktionen.

Beweis: Die λ-berechenbaren Funktionen lassen sich offensichtlich durch Programme einer der gängigen imperativen Programmiersprachen simulieren. Da diese sich wiederum durch Turingmaschinen beschreiben lassen und die Klasse der μ-rekursiven Funktionen identisch sind mit der Klasse der Turing-berechenbaren Funktionen, folgt hieraus, daß alle λ-berechenbaren Funktionen auch μ-rekursiv sind.

Wir können uns daher auf den interessanten Teil des Beweises konzentrieren, nämlich dem Nachweis, daß man mit einem so einfachen Berechnungsmechanismus wie dem λ-Kalkül tatsächlich alle berechenbaren Funktionen repräsentieren können. Wir weisen dazu nach, daß alle sieben Bedingungen der μ-rekursiven Funktionen aus Definition 2.3.27 durch λ-Terme erfüllt werden können.

1. Gemäß Definition 2.3.26 werden Konstanten $0, 1, 2, \ldots \in \mathbb{N}$ genau durch die Church-Numerals $\overline{0}$,$\overline{1}$,$\overline{2}$,... aus Definition 2.3.17 repräsentiert. Damit sind alle Konstanten auch λ-berechenbar.
2. Die Nullfunktion z läßt sich darstellen durch den Term $\lambda \mathtt{n}.\,\overline{0}$ und ist somit λ-berechenbar.
3. Die Nachfolgerfunktion s haben wir in Beispiel 2.3.18 auf Seite 55 untersucht. Sie wird dargestellt durch den Term $\mathbf{s}$ und ist somit auch λ-berechenbar.

[55] $\vec{x}$ ist abkürzend für ein Tupel $(x_1, ..., x_m)$

4. Die Projektionsfunktionen pr^n_m sind leicht zu repräsentieren. Man wähle für pr^n_m den Term $\lambda \mathtt{x}_1 . \ldots \lambda \mathtt{x}_n . \mathtt{x}_m$.

5. Die Komposition läßt sich genauso leicht direkt nachbilden. Definieren wir

   $$\mathbf{Cn} \equiv \lambda \mathtt{f} . \lambda \mathtt{g}_1 . \ldots . \lambda \mathtt{g}_n . \ \lambda \vec{\mathtt{x}} . \ \mathtt{f}\ (\mathtt{g}_1\ \vec{\mathtt{x}})\ \ldots (\mathtt{g}_n\ \vec{\mathtt{x}})$$

   so simuliert **Cn** den Kompositionsoperator Cn.[56]

   Sind also $f, g_1 ... g_n$ λ-berechenbare Funktionen und $F, G_1 ... G_n$ die zugehörigen λ-Terme, so repräsentiert **Cn** $F\ G_1\ \ldots G_n$ – wie man durch Einsetzen leicht zeigen kann – die Komposition $Cn[f,\ g_1 ... g_n]$. Damit ist gezeigt, daß die Komposition λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.

6. Mithilfe der Fixpunktkombinatoren läßt sich die primitive Rekursion zweier Funktionen auf einfache und natürliche Weise nachbilden. Wir müssen hierzu nur die Rekursionsgleichungen von $Pr[f, g]$ in eine einzige Gleichung umwandeln. Für $h = Pr[f, g]$ gilt

   $$h(\vec{\mathrm{x}}, y) = \begin{cases} f(\vec{\mathrm{x}}) & \text{falls } y = 0 \\ g(\vec{\mathrm{x}}, y{-}1,\ h(\vec{\mathrm{x}}, y{-}1)) & \text{sonst} \end{cases}$$

   Wenn wir die rechte Seite dieser Gleichung durch Conditional und Vorgängerfunktion beschreiben und hierauf anschließend den **Y**-Kombinator anwenden, so haben wir eine Beschreibung fur die Funktion h durch einen λ-Term. Definieren wir also

   $$\mathbf{Pr} \equiv \lambda \mathtt{f} . \lambda \mathtt{g} . \ \mathbf{Y}\ (\lambda \mathtt{h} . \lambda \vec{\mathtt{x}} . \lambda \mathtt{y} . \ \mathsf{if}\ \mathbf{zero}\mathtt{y}\ \mathsf{then}\ \mathtt{f}\ \vec{\mathtt{x}}\ \mathsf{else}\ \mathtt{g}\ \vec{\mathtt{x}}\ (\mathbf{p}\ \mathtt{y})\ (\mathtt{h}\ \vec{\mathtt{x}}\ (\mathbf{p}\ \mathtt{y}))\ )$$

   so simuliert **Pr** den Operator der primitiven Rekursion Pr. Damit ist gezeigt, daß die primitive Rekursion λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.

7. Auch die Minimierung kann mithilfe der Fixpunktkombinatoren als λ-berechenbar nachgeweisen werden. Minimierung $Mn[f](\vec{\mathrm{x}})$ ist im Endeffekt eine unbegrenzte Suche nach einem Wert y, für den $f(\vec{\mathrm{x}}, y) = 0$ ist. Startwert dieser Suche ist die Zahl 0.

   Die bei einem gegebenen Startwert y beginnende Suche nach einer Nullstelle von f läßt sich wie folgt durch durch eine rekursive Gleichung ausdrücken.

   $$\mathsf{min}_f(\vec{\mathrm{x}}, y) = \begin{cases} y & \text{falls } f(\vec{\mathrm{x}}, y) = 0 \\ \mathsf{min}_f(\vec{\mathrm{x}}, y+1) & \text{sonst} \end{cases}$$

   Da f und $\vec{\mathrm{x}}$ für diese Gleichung als Konstante aufgefaßt werden können, läßt sich diese Gleichung etwas vereinfachen, bevor wir auf sie den **Y**-Kombinator anwenden. Definieren wir also

   $$\mathbf{Mn} \equiv \lambda \mathtt{f} . \lambda \vec{\mathtt{x}} . \ (\ \mathbf{Y}\ (\lambda \mathtt{min} . \lambda \mathtt{y} . \ \mathsf{if}\ \mathbf{zero}(\mathtt{f}\ \vec{\mathtt{x}}\ \mathtt{y})\ \mathsf{then}\ \mathtt{y}\ \mathsf{else}\ \mathtt{min}\ (\mathbf{s}\ \mathtt{y})\ )\ )\ \overline{0}$$

   so simuliert **Mn** den Minimierungsoperator Mn. Damit ist gezeigt, daß die Minimierung λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.

Wir haben somit bewiesen, daß alle Grundfunktionen λ-berechenbar sind und die Klasse der λ-berechenbaren Funktionen abgeschlossen ist unter den Operationen Komposition, primitive Rekursion und Minimierung. Damit sind alle μ-rekursiven Funktionen auch λ-berechenbar. □

2.3.6 Semantische Fragen

Wir haben in den bisherigen Abschnitten die Syntax und die Auswertung von λ-Termen besprochen und gezeigt, daß man mit λ-Termen alle berechenbaren Funktionen ausdrücken kann. Offensichtlich ist auch, daß jeder λ-Term der Gestalt $\lambda x . t$ eine Funktion beschreiben muß. Jedoch ist unklar, mit welchem mathematischen Modell man eine solche Funktion als mengentheoretisches Objekt beschreiben kann.

Es ist nicht sehr schwer, ein sehr abstraktes und an Termen orientiertes Modell für den λ-Kalkül anzugeben (siehe [Church & Rosser, 1936]). Jeder Term t beschreibt die Menge M_t der Terme, die im Sinne von Definition 2.3.24 gleich sind zu t. Jede Abstraktion $\lambda x . t$ repräsentiert eine Funktion $f_{\lambda x.t}$, welche eine Menge M_u in die Menge $M_{t[u/x]}$ abbildet. Diese Charakterisierung bringt uns jedoch nicht weiter, da sie keine Hinweise auf den Zusammenhang zwischen Funktionen des λ-Kalküls und 'gewöhnlichen' mathematischen Funktionen liefert.

[56] Genau besehen haben wir hier beschrieben, wie wir für jede feste Anzahl n von Funktionen $\mathtt{g}_1 \ldots \mathtt{g}_n$ den Kompositionsoperator simulieren.

Einfache mathematische Modelle, in denen λ-Terme als Funktionen eines festen Funktionenraumes interpretiert werden können, sind jedoch ebenfalls auszuschließen. Dies liegt an der Tatsache, daß λ-Terme eine Doppelrolle spielen: sie können als Funktion und als Argument einer anderen Funktion auftreten. Daher ist es möglich, λ-Funktionen zu konstruieren, die in sinnvoller Weise auf sich selbst angewandt werden können.

Beispiel 2.3.29

Wir betrachten den Term

$$\texttt{twice} \equiv \lambda\texttt{f}.\lambda\texttt{x}.\texttt{f (f x)}.$$

Angewandt auf zwei Terme f und u produziert dieser Term die zweifache Anwendung von f auf u

$$\texttt{twice}\ f\ u \xrightarrow{*} f\ (f\ u)$$

Damit ist `twice` als eine Funktion zu verstehen. Andererseits darf `twice` aber auf sich selbst angewandt werden, wodurch eine Funktion entsteht, die ihr erstes Argument viermal auf das zweite anwendet:

$$\begin{aligned}
\texttt{twice twice} &\equiv \texttt{(}\lambda\texttt{f.}\lambda\texttt{x.f (f x)) twice} \\
&\xrightarrow{*} \lambda\texttt{x. twice (twice x)} \\
&\equiv \lambda\texttt{x.(}\lambda\texttt{f.}\lambda\texttt{x.f (f x)) (twice x)} \\
&\xrightarrow{*} \lambda\texttt{x.}\lambda\texttt{x'. (twice x) ((twice x) x')} \\
&\cong \lambda\texttt{f.}\lambda\texttt{x. (twice f) ((twice f) x)} \\
&\xrightarrow{*} \lambda\texttt{f.}\lambda\texttt{x. (twice f) (f (f x))} \\
&\xrightarrow{*} \lambda\texttt{f.}\lambda\texttt{x. f (f (f (f x)))}
\end{aligned}$$

Die übliche mengentheoretische Sichtweise von Funktionen muß die Selbstanwendung von Funktionen jedoch verbieten. Sie identifiziert nämlich eine Funktion f mit der Menge $\{(x, y) \mid y = f(x)\}$, also dem Graphen der Funktion. Wenn eine selbstanwendbare Funktion wie `twice` mengentheoretisch interpretierbar wäre, dann müsste `twice` als eine solche Menge aufgefaßt werden und würde gelten, daß $(\texttt{twice}, y)$ für irgendein y ein Element dieser Menge ist, weil nun einmal `twice` ein legitimes Argument von `twice` ist. Dies aber verletzt ein fundamentales Axiom der Mengentheorie, welches besagt, daß eine Menge sich selbst nicht enthalten darf.[57] Wir können daher nicht erwarten, 'natürliche' Modelle für den λ-Kalkül angeben zu können. Dies wird erst möglich sein, wenn wir die zulässigen Terme auf syntaktischem Wege geeignet einschränken.[58] Diese Problematik ist jedoch nicht spezifisch für den λ-Kalkül, sondern betrifft *alle* Berechenbarkeitsmodelle, da diese – wie wir im vorigen Abschnitt gezeigt hatten – äquivalent zum λ-Kalkül sind.

Das erste mathematische Modell für berechenbare Funktionen ist die *Domain-Theorie*, die Anfang der siebziger Jahre von Dana Scott [Scott, 1972, Scott, 1976] entwickelt wurde. Diese Theorie basiert im wesentlichen auf topologischen Begriffen wie Stetigkeit und Verbänden. Sie benötigt jedoch ein tiefes Verständnis komplexer mathematischer Theorien. Aus diesem Grunde werden wir auf die *denotationelle* Semantik des λ-Kalküls nicht weiter eingehen.

2.3.7 Eigenschaften des λ-Kalküls

Bei den bisherigen Betrachtungen sind wir stillschweigend davon ausgegangen, daß jeder λ-Term einen Wert besitzt, den wir durch Reduktion bestimmen können. Wie weit aber müssen wir gehen, bevor wir den Prozeß der Reduktion als beendet erklären können? Die Antwort ist eigentlich naheliegend: wir hören erst dann auf, wenn nichts mehr zu reduzieren ist, also der entstandene Term kein Redex im Sinne der Definition 2.3.10 mehr

[57] Ein Verzicht auf dieses Axiom würde zu dem *Russelschen Paradox* führen:

Man betrachte die Mengen $M = \{X \mid X \notin X\}$ und untersuche, ob $M \in M$ ist oder nicht. Wenn wir $M \in M$ annehmen, so folgt nach Definition von M, daß M – wie jedes andere Element von M – nicht in sich selbst enthalten ist, also $M \notin M$. Da M aber die Menge *aller* Mengen ist, die sich selbst nicht enthalten, muß M ein Element von M sein: $M \in M$.

[58] Ein einfaches aber doch sehr wirksames Mittel ist hierbei die Forderung nach Typisierbarkeit, die wir im nächsten Abschnitt diskutieren werden. Die Zuordnung von Typen zu Termen beschreibt bereits auf syntaktischem Wege, zu welcher Art von Funktionenraum eine Funktion gehören soll.

enthält. Diese Überlegung führt dazu, eine Teilklasse von λ-Termen besonders hervorzuheben, nämlich solche, die nicht mehr reduzierbar sind. Diese Terme repräsentieren die Werte, die als Resultat von Berechnungen entstehen können.

Definition 2.3.30 (Normalform)

Es seien s und t beliebige λ-Terme.

1. *t ist in Normalform, wenn t keine Redizes enthält.*
2. *t ist normalisierbar, wenn es einen λ-Term in Normalform gibt, auf den t reduziert werden kann.*
3. *t heißt Normalform von s, wenn t in Normalform ist und* $s \xrightarrow{*} t$ *gilt.*

Diese Definition wirft natürlich eine Reihe von Fragen auf, die wir im folgenden diskutieren wollen.

Hat jeder λ-Term eine Normalform?

In Anbetracht der Tatsache, daß der λ-Kalkül Turing-mächtig ist, muß diese Frage natürlich mit *nein* beantwortet werden. Bekanntermaßen enthalten die rekursiven Funktionen auch die partiellen Funktionen – also Funktionen, die nicht auf allen Eingaben definiert sind – und es ist nicht möglich, einen Formalismus so einzuschränken, daß nur totale Funktionen betrachtet werden, ohne daß dabei gleichzeitig manche berechenbare totale Funktion nicht mehr beschreibbar ist.[59] Diese Tatsache muß sich natürlich auch auf die Reduzierbarkeit von λ-Termen auswirken: es gibt Reduktionsketten, die nicht terminieren. Wir wollen es jedoch nicht bei dieser allgemeinen Antwort belassen, sondern einen konkreten Term angeben, der nicht normalisierbar ist.

Lemma 2.3.31

Der Term `(λx. x x) (λx. x x)` *besitzt keine Normalform.*

Beweis: Bei der Reduktion von `(λx. x x) (λx. x x)` gibt es genau eine Möglichkeit. Wenn wir diese ausführen, erhalten wir denselben Term wie zuvor und können den Term somit unendlich lange weiterreduzieren. □

Führt jede Reduktionsfolge zu einer Normalform, wenn ein λ-Term normalisierbar ist?

Im Beispiel 2.3.12 (Seite 51) hatten wir bereits festgestellt, daß es unter Umständen mehrere Möglichkeiten gibt, einen gegebenen λ-Term zu reduzieren. Da wir bereits wissen, daß nicht jede Reduktionskette terminieren muß, erhebt sich natürlich die Frage, ob es etwa Terme gibt, bei denen eine Strategie, den zu reduzierenden Teilterm auszuwählen, zu einer Normalform führt, während eine andere zu einer nichtterminierenden Reduktionsfolge führt. Dies ist in der Tat der Fall.

Lemma 2.3.32

Es gibt normalisierbare λ-Terme, bei denen nicht jede Reduktionsfolge zu einer Normalform führt.

Beweis: Es sei W≡`λx. x x x` und I≡`λx. x`. Wir betrachten den Term

`(λx.λy.y) (WW) I`.

Es gibt zwei Möglichkeiten, diesen Term zu reduzieren. Wählen wir die am meisten links-stehende, so ergibt sich als Reduktionsfolge:

$$\texttt{(}\lambda\texttt{x.}\lambda\texttt{y.y) (WW) I} \xrightarrow{*} \texttt{(}\lambda\texttt{y.y) I} \xrightarrow{*} \texttt{I}$$

und wir hätten eine Normalform erreicht. Wenn wir dagegen den Teilterm `(WW)` zuerst reduzieren, dann erhalten wir `(WWW)`. Reduzieren wir dann wieder im gleichen Bereich, so erhalten wir folgende Reduktionskette

$$\texttt{(}\lambda\texttt{x.}\lambda\texttt{y.y) (WW) I} \xrightarrow{*} \texttt{(}\lambda\texttt{x.}\lambda\texttt{y.y) (WWW) I} \xrightarrow{*} \texttt{(}\lambda\texttt{x.}\lambda\texttt{y.y) (WWWW) I} \xrightarrow{*} \ldots$$

Diese Kette erreicht niemals eine Normalform. □

[59] In der Sprache der theoretischen Informatik heißt dies: "die Menge der total-rekursiven Funktionen ist nicht rekursiv-aufzählbar"

Wie kann man eine Normalform finden, wenn es eine gibt?

Wir wissen nun, daß die Bestimmung einer Normalform von der Reduktionsstrategie abhängen kann. Die Frage, die sich daraus unmittelbar ergibt, ist, ob es denn wenigstens eine einheitliche Strategie gibt, mit der man eine Normalform finden kann, wenn es sie gibt? Die Beantwortung dieser Frage ist von fundamentaler Bedeutung für die praktische Verwendbarkeit des λ-Kalküls als Programmiersprache. Ohne eine Reduktionsstrategie, mit der man garantiert eine Normalform auch finden kann, wäre der λ-Kalkül als Grundlage der Programmierung unbrauchbar. Glücklicherweise kann man diese Frage positiv beantworten

Lemma 2.3.33 (Leftmost-Reduktion)

Reduziert man in einem λ-Term immer das jeweils am meisten links stehende (äußerste) Redex, so wird man eine Normalform finden, wenn der Term normalisierbar ist.

Intuitiv läßt sich der Erfolg dieser Strategie wie folgt begründen. Der Beweis von Lemma 2.3.32 hat gezeigt, daß normalisierbare Terme durchaus Teilterme enthalten können, die nicht normalisierbar sind. Diese Teilterme können nun zur Bestimmung der Normalform nichts beitragen, da ihr Wert ja nicht festgestellt werden kann. Wenn wir daher die äußerste Funktionsanwendung zuerst reduzieren, werden wir feststellen, ob ein Teilterm überhaupt benötigt wird, bevor wir ihn unnötigerweise reduzieren.

Die Strategie, zuerst die Funktionsargumente einzusetzen, bevor sie deren Wert bestimmt, entspricht der *call-by-name* Auswertung in Programmiersprachen. Sie ist die sicherste Reduktionsstrategie, aber die Sicherheit wird oft auf Kosten der Effizienz erkauft. Es mag nämlich sein, daß durch die Reduktion ein Argument verdoppelt wird und daß wir es somit zweimal reduzieren müssen. Eine *call-by-value* Strategie hätte uns diese Doppelarbeit erspart, aber diese können wir nur anwenden, wenn wir wissen, daß sie garantiert terminiert. Da das Halteproblem jedoch unentscheidbar ist, gibt es leider keine Möglichkeit, dies für beliebige λ-Terme im Voraus zu entscheiden.[60] Ein präziser Beweis für diese Aussage ist verhältnismäßig aufwendig. Wir verweisen daher auf Lehrbücher über den λ-Kalkül wie [Barendregt, 1981, Stenlund, 1972] für Details.

Ist die Normalform eines λ-Terms eindeutig?

Auch hierauf benötigen wir eine positive Antwort, wenn wir den λ-Kalkül als Programmiersprache verwenden wollen. Wenn nämlich bei der Auswertung eines λ-Terms verschiedene Reduktionsstrategien zu verschiedenen Ergebnissen (Normalformen) führen würden, dann könnte man den Kalkül zum Rechnen und zum Schließen über Programme nicht gebrauchen, da er nicht eindeutig genug festlegt, was der Wert eines Ausdrucks ist.[61]

Rein syntaktisch betrachtet ist der Reduktionsmechanismus des λ-Kalküls ein *Termersetzungssystem*, welches Vorschriften angibt, wie Terme in andere Terme umgeschrieben werden dürfen (engl. *rewriting*). In der Denkweise der Termersetzung ist die Frage nach der Eindeutigkeit der Normalform ein Spezialfall der Frage nach der *Konfluenz* eines Regelsystems, also der Frage, ob zwei Termersetzungsketten, die im gleichen Term begonnen haben, wieder zusammengeführt werden können.

Die genauen Definitionen der Konfluenz werden wir im Abschnitt 2.4.5.3 geben, wo wir für typisierbare λ-Terme ein Konfluenztheorem beweisen werden. Auch für den uneingeschränkten λ-Kalkül kann man Konfluenz beweisen. Der Beweis dieses Theorems, das nach den Mathematikern A. Church und B. Rosser benannt wurde, ist allerdings relativ aufwendig. Für Details verweisen wir daher wiederum auf Lehrbücher wie [Barendregt, 1981, Hindley & Seldin, 1986, Stenlund, 1972].

Satz 2.3.34 (Church-Rosser Theorem)

Es seien t, u und v beliebige λ-Terme und es gelte $t \xrightarrow{*} u$ und $t \xrightarrow{*} v$.
Dann gibt es einen λ-Term s mit der Eigenschaft $u \xrightarrow{*} s$ und $v \xrightarrow{*} s$.

[60] Diese Möglichkeit wird uns nur bei typisierbaren λ-Termen geboten, die wir im folgenden Abschnitt besprechen.

[61] Rein theoretisch gäbe es aus einem solchen Dilemma immer noch einen Ausweg. Man könnte den Kalkül von vorneherein mit einer Reduktionsstrategie koppeln und hätte dann eine eindeutige Berechnungsvorschrift. Auf den Kalkül zum Schließen über Programme hätte dies aber negative Auswirkunge, da wir auch hier die Reduktionsstrategie mit einbauen müßten.

Eine unmittelbare Konsequenz dieses Theorems ist, daß die Normalformen eines gegebenen λ-Terms bis auf α-Konversionen eindeutig bestimmt sind und daß der in Abbildung 2.7 auf Seite 58 angegebenen Kalkül zum Schließen über die Gleichheit von λ-Termen korrekt ist im Sinne von Definition 2.3.24.

Korollar 2.3.35

Es seien t, u und v beliebige λ-Terme.

1. *Wenn u und v Normalformen von t sind, dann sind sie kongruent im Sinne von Definition 2.3.9.*
2. *Der Kalkül zum Schließen über die Gleichheit ist korrekt: Wenn $u{=}v$ bewiesen werden kann, dann gibt es einen λ-Term s mit der Eigenschaft $u \xrightarrow{*} s$ und $v \xrightarrow{*} s$.*
3. *Gilt $u{=}v$ und v ist in Normalform, so folgt $u \xrightarrow{*} v$.*
4. *Gilt $u{=}v$, so haben u und v dieselben Normalformen oder überhaupt keine.*
5. *Wenn u und v in Normalform sind, dann sind sie entweder kongruent oder nicht gleich.*

Es ist also legitim, λ-Terme als Funktionen anzusehen und mit dem in Abbildung 2.7 angegebenen Kalkül Schlüsse über die Resultate von Berechnungen zu ziehen.

Welche extensionalen Eigenschaften von λ-Termen kann man automatisch beweisen?

Die bisherigen Fragestellungen richteten sich im wesentlichen auf den Reduktionsmechanismus des λ-Kalküls. Wir wollen mit dem λ-Kalkül jedoch nicht nur rechnen, sondern auch extensionalen Eigenschaften von Programmen – also das nach außen sichtbare Verhalten – mit formalen Beweismethoden untersuchen. Die Frage, welche dieser Eigenschaften mithilfe fester Verfahren überprüft werden können, liegt also nahe.

Leider gibt uns die Rekursionstheorie (siehe z.B. [Rogers, 1967]) auf diese Frage eine sehr negative Antwort. Der Preis, den wir dafür bezahlen, daß der λ-Kalkül genauso mächtig ist wie die rekursiven Funktionen, ist – neben der Notwendigkeit, auch partielle Funktionen zu betrachten – die Unentscheidbarkeit *aller* extensionaler Eigenschaften von Programmen.

Satz 2.3.36 (Satz von Rice)

Keine nichttriviale extensionale Eigenschaft von λ-Termen kann mithilfe eines allgemeinen Verfahrens entschieden werden.

Die Bedeutung dieser Aussage wird klar, wenn wir uns ein paar typische Fragen ansehen, die man gerne mit einem festen Verfahren entscheiden können würde:

- *Terminiert die Berechnung einer Funktion f bei Eingabe eines Argumentes x?* (Halteproblem)
- *Ist die durch einen λ-Term f beschriebene Funktion total?*
- *Gehört ein Wert y zum Wertebereich einer durch einen λ-Term f beschriebenen Funktion?*
- *Berechnet die durch einen λ-Term f beschriebene Funktion bei Eingabe von x den Wert y?*
- *Sind zwei λ-berechenbare Funktionen f und g gleich?* (Dies ist nicht einmal beweisbar)

Die Liste könnte beliebig weiter fortgesetzt werden, da der Satz von Rice tatsächlich besagt, daß nicht eine einzige interessante Fragestellung mit einem universellen Verfahren entschieden werden kann. Aus diesem Grunde ist der allgemeine λ-Kalkül zum Schließen über Programme und ihre Eigenschaften ungeeignet. Essentiell für eine derart negative Antwort ist dabei jedoch die Forderung nach einem *universellen* Verfahren, das für jede beliebige Eingabe die Frage beantwortet. Verzichtet man jedoch auf die Universalität und schränkt die zu betrachtenden λ-Terme durch syntaktische Mittel ein, so kann es durchaus möglich sein, für diese kleinere Klasse von Programmen Entscheidungsverfahren für die wichtigsten Fragen zu konstruieren.

2.4 Die einfache Typentheorie

Im vorhergehenden Abschnitt haben wir gesehen, daß der λ-Kalkül einen einfachen und zugleich sehr mächtigen Formalismus zum Schließen über Programme darstellt. Die große Ausdruckskraft bringt jedoch eine Reihe von Problemen mit sich, welche eine Automatisierung des Schlußfolgerungsprozesses erheblich erschweren. Es gibt keine natürlichen mengentheoretischen Interpretationen für λ-Terme; keine extensionale Eigenschaft von λ-Termen kann vollautomatisch entschieden werden; in formalen Beweisen müssen wir mit partiellen Funktionen rechnen und bei totalen Funktionen hängt die Terminierung von Reduktionen immer noch davon ab, an welcher Stelle wir reduzieren.

Viele dieser Probleme haben ihre Ursache in der Universalität des λ-Kalküls, die in der Programmierpraxis gar nicht voll ausgeschöpft wird. Operationen wie ein universeller Fixpunktkombinator tauchen in realistischen Programmen niemals auf und werden nur benötigt, um komplexere Berechnungs*mechanismen* innerhalb des λ-Kalküls zu *erklären*. Es ist also durchaus möglich, auf Universalität zu verzichten, ohne daß der Kalkül – im Hinblick auf praktische Verwendbarkeit – an Ausdruckskraft verliert. Unser Ziel ist daher, durch syntaktische Einschränkungen des ansonsten sehr gut geeigneten λ-Kalküls ein schwächeres Berechenbarkeitsmodell zu entwickeln, welches ermöglicht, wichtige Eigenschaften der betrachteten Programme automatisch zu beweisen, und dennoch ausdrucksstark genug ist, alle in der Praxis auftretenden Probleme zu handhaben.

Der zentrale Grund für die Unentscheidbarkeit von Programmeigenschaften im λ-Kalkül ist die Tatsache, daß λ-Terme auf sich selbst angewandt werden dürfen.[62] Es ist möglich, einen λ-Term ganz anders zu verwenden als für den Zweck, für den er ursprünglich geschaffen wurde. So ist es zum Beispiel legitim, das λ-Programm $\overline{2}\,\overline{2}$ aufzustellen und zu $\overline{4}$ auszuwerten. Auch sind die λ-Terme für $\overline{0}$, **T** und `[]` (vgl. Abschnitt 2.3.3) identisch und somit könnte man auch Ausdrücke wie **add T** `[]` zu $\overline{0}$ auswerten, was semantisch ziemlich unsinnig ist. Die uneingeschränkte Anwendbarkeit von λ-Termen auf Argumente ist – wie in Abschnitt 2.3.6 angedeutet – auch der Grund dafür, daß es keine einfachen mengentheoretischen Modelle für den λ-Kalkül geben kann. Um diese Probleme zu lösen, liegt es nahe, einen Formalismus zu entwickeln, durch den die Anwendbarkeit von λ-Termen auf sinnvolle Weise beschränkt werden kann.

Mathematisch besehen ist die Unentscheidbarkeit von Programmeigenschaften verwandt mit dem Russelschen Paradox der frühen Mengentheorie (siehe Fußnote 57 auf Seite 62). So wie die Möglichkeit einer *impredikativen* Beschreibung von Mengen[63] in der Mengentheorie zu paradoxen Situationen führt, so ermöglicht die Selbstanwendbarkeit in der Programmierung die Konstruktion von Diagonalisierungsbeweisen, welche die Entscheidbarkeit von Programmeigenschaften ad absurdum führen. In seiner *Theorie der Typen* [Russel, 1908] hat Russel bereits im Jahre 1908 als Lösung vorgeschlagen, durch eine *Typdisziplin* jedem mathematischen Symbol einen Bereich zuzuordnen, zu dem es gehören soll. Diese Typdisziplin verbietet die Bildung von Mengen der Art $\{X \mid X \notin X\}$, da das Symbol '$\in$' nunmehr Objekte verschiedener Typen in Beziehung setzt.

In gleicher Weise kann man den λ-Kalkül durch eine Typdisziplin ergänzen, welche eine stärkere Strukturierung von λ-Programmen bereits auf der syntaktischen Ebene ermöglicht. Im *getypten λ-Kalkül* [Church, 1940] werden Typen dazu benutzt, um die Menge der zulässigen λ-Terme syntaktisch stärker einzuschränken als dies im allgemeinen Kalkül gemäß Definition 2.3.2 der Fall ist. Darüber hinaus dienen sie aber auch als Bezeichner für die Funktionenräume, zu denen ein getypter λ-Term gehören soll. Die Typdisziplin verbietet viele der seltsam anmutenden Konstruktionen[64] des allgemeinen λ-Kalküls und sorgt somit dafür, daß viele Eigenschaften getypter λ-Terme algorithmisch entscheidbar werden.

[62] Im Konzept der Turingmaschinen und imperativen Programmiersprachen entspricht die Selbstanwendbarkeit der Existenz einer universellen Maschine, auf die man ungern verzichten möchte.

[63] Unter einer *impredikativen* Beschreibung versteht man die uneingeschränkte abstrakte Definition einer Menge M durch $M = \{x \mid x \text{ hat Eigenschaft } P\}$.

[64] Die Tendenz in modernen Programmiersprachen geht – wenn auch verspätet – in die gleiche Richtung. Während es in vielen älteren Programmiersprachen wie Fortran, C, Lisp, etc. erlaubt ist, Variablen völlig anders zu verwenden als ursprünglich vorgesehen (man darf z.B. einen Integer-Wert als boolesche Variable oder als Buchstaben weiterverwenden, wenn es der Effizienz dient), haben fast alle neueren Programmiersprachen wie Pascal, Eiffel, ML, etc. ein strenges Typsystem, welches derartige Mißbräuche verbietet. Diese Typdisziplin bringt zwar einen gewissen Mehraufwand bei der Programmierung mit sich, sorgt aber auch für einen strukturierteren und übersichtlicheren Programmierstil.

Für einen formalen Kalkül zum Schließen über Programmeigenschaften bietet eine Typdisziplin aber noch mehr als nur eine Einschränkung von Termen auf solche, die natürliche Modelle und ein automatisiertes Schließen zulassen. Der Typ eines Terms kann darüber hinaus auch als eine Beschreibung der *Eigenschaften* des Terms angesehen werden und in diese Beschreibung könnte man weit mehr aufnehmen als nur die Charakterisierung der Mengen, auf denen der Term operiert. Eine Typisierung der Art

$$\texttt{sqrt x} \in \{\texttt{y:}\mathbb{N} \mid \texttt{y*y} \leq \texttt{x} < \texttt{(y+1)*(y+1)}\}$$

würde zum Beispiel festlegen, daß die Funktion `sqrt` für jedes `x` eine (eindeutig bestimmte) Integerquadratwurzel von `x` bestimmt. Die Typdisziplin bietet also eine einfache Möglichkeit an, syntaktisch über den Bezug zwischen einem Programm und seiner *Spezifikation* zu schließen.

Es gibt zwei Wege, eine Typdisziplin über λ-Termen aufzubauen. Die ursprüngliche Vorgehensweise von Church's *getyptem λ-Kalkül* ('typed λ-calculus', siehe [Church, 1940]) nimmt die Typen direkt in die Terme mit auf – wie zum Beispiel in $\lambda \mathtt{f}^{\mathtt{S}\to\mathtt{T}}.\lambda \mathtt{x}^{\mathtt{S}}.\, \mathtt{f}^{\mathtt{S}\to\mathtt{T}}\,\mathtt{x}^{\mathtt{S}}$. Im wesentlichen muß man hierzu die Definition 2.3.2 der λ-Terme auf Seite 48 in eine für *getypte λ-Terme abwandeln*.[65] Ein zweiter Ansatz betrachtet λ-Terme und Typen als zwei unabhängige Konzepte. Dies bedeutet, daß der λ-Kalkül unverändert übernommen wird und eine (einfache) *Typentheorie* als separater Kalkül hinzukommt, mit dem nachgewiesen werden kann, daß ein λ-Term *typisierbar* ist, also zu einem bestimmten Typ gehört. In diesem Ansatz würde man zum Beispiel $\lambda \mathtt{f}.\lambda \mathtt{x}.\, \mathtt{f\,x} \in (\mathtt{S}\to\mathtt{T}) \to \mathtt{S} \to \mathtt{T}$ nachweisen.

Beide Vorgehensweisen führen im Endeffekt zu den gleichen Resultaten, obwohl sie von ihrem Ansatz her recht verschieden sind. So werden im getypten λ-Kalkül die einzelnen Variablen getypt, während die Typentheorie jeweils nur einem gesamten Ausdruck einen Typ zuweist. Wir werden daher im die Begriffe *getypter λ-Kalkül* und *Typentheorie* als synonym ansehen und nicht weiter unterscheiden..

Aus praktischer Hinsicht ist der Weg der Typentheorie sinnvoller, da er bei der Formulierung und Berechnung von Programmen die volle Freiheit des λ-Kalküls behält und die Typisierung als ein statisches Konzept hinzunimmt, welches nur bei der Beweisführung hinzugenommen wird. Damit bleibt die Syntax von Programmen unkompliziert, da sie nicht immer wieder durch Typinformationen überfrachtet wird.[66] Erkauft wird diese Freiheit durch einen gewissen Mehraufwand bei der Beweisführung, da für einzelne Teilausdrücke eine Typisierung rekonstruiert werden muß, die man im getypten λ-Kalkül direkt aus dem Programm ablesen könnte (weil man sie dort bereits angeben *mußte*).

Wir werden im folgenden nun die sogenannte *Theorie der einfachen Typen* (*simply typed λ-calculus*) vorstellen, welche nur diejenigen Typisierungskonzepte enthält, die für eine natürliche Interpretation von λ-Termen unumgänglich sind. Wir werden am Ende dieses Abschnitts sehen, daß wir zugunsten einer hinreichenden Ausdruckskraft noch weitere Typkonstrukte hinzunehmen müssen, aber die hier entwickelten Konzepte fortführen können. Bevor wir die formalen Details besprechen, wollen wir zwei grundsätzliche Fragen klären.

Was sind die Charakteristika eines Typs?

Typen können in erster Näherung als das syntaktische Gegenstück zu Mengen angesehen werden. In der Mathematik ist eine Menge eindeutig dadurch charakterisiert, daß man angibt, welche *Elemente* zu dieser Menge gehören. So besteht zum Beispiel die Menge $\mathbb{N}$ der natürlichen Zahlen aus den Elementen `0,1,2,3,4,...`, die Menge $\mathbb{Z}$ der ganzen Zahlen aus den Elementen `0,1,-1,2,-2,...`, die Menge $\mathbb{Q}$ der Rationalzahlen aus $\texttt{0,1,-1,}\frac{1}{2}\texttt{,}-\frac{1}{2}\texttt{,2,-2,...}$ und die Menge $\mathbb{B}$ der booleschen Werte aus wahr und falsch.

Etwas komplizierter wird es, wenn man Mengen*konstruktoren* wie das Produkt $M \times M'$ zweier Mengen M und M', den Funktionenraum $M \to M'$ oder den Raum M `list` aller Listen von Elementen aus M charakterisieren möchte. In diesem Falle ist es nicht möglich, die konkreten Elemente anzugeben. Man muß sich daher

[65] Aus diesem Ansatz stammt der Begriff der *getypten Variablen*, den wir benutzen werden, um auszudrücken, daß eine Variable ein Platzhalter für Terme eines bestimmten Typs ist. In $\lambda \mathtt{f}^{\mathtt{S}\to\mathtt{T}}.\lambda \mathtt{x}^{\mathtt{S}}.\mathtt{f}^{\mathtt{S}\to\mathtt{T}}\,\mathtt{x}^{\mathtt{S}}$ ist z.B. die Variable `f` eine Variable des Typs $\mathtt{S}\to\mathtt{T}$.

[66] Moderne Programmiersprachen gehen den gleichen Weg. Das Typkonzept steht außerhalb des eigentlichen auszuführenden Programms und wird statisch vom Compiler überprüft. Hierdurch entfällt die Notwendigkeit von Kontrollen zur Ausführungszeit, wodurch ein Programm deutlich effizienter wird.

damit behelfen, zu beschreiben, wie die Elemente dieser Mengen zu *bilden* sind. So ist zum Beispiel $M \times M'$ genau die Menge aller Paare (a,b), wobei a ein Element von M und b eines von M' ist.

Genau besehen haben wir auch bei der Charakterisierung der Mengen $\mathbb{N}$, $\mathbb{Z}$ und $\mathbb{Q}$ nur textliche Beschreibungen der Elemente verwendet und nicht etwa die Elemente selbst. So benutzen wir bei den größeren natürlichen Zahlen wie 12 eine Aneinanderreihung von Symbolen. Bei den ganzen Zahlen erscheint das Symbol '-', um negative Zahlen zu beschreiben. Bei den Rationalzahlen behelfen wir uns mit der Bruchschreibweise, wobei es sogar vorkommen kann, daß wir dasselbe Element auf verschiedene Arten beschreiben, wie etwa durch $\frac{129}{21}$ und $\frac{86}{14}$. Bei den Rationalzahlen hat man sich darauf geeinigt, unter allen Darstellungsmöglichkeiten einer Zahl diejenige besonders herauszuheben, die sich nicht weiter (durch Kürzen) vereinfachen läßt. Eine solche standardisierte Darstellung nennt man auch *kanonisch* und man spricht der Einfachheit halber von den kanonischen Elementen der Menge $\mathbb{Q}$.

Bei den bisherigen Betrachtungen haben wir uns hauptsächlich mit den mathematischen Aspekten von Typen beschäftigt. Typen spielen jedoch auch in der Programmierung eine wesentliche Rolle. Mithilfe einer Typdisziplin kann man statisch überprüfen, ob die Anwendung von Operationen wie $+, *, -, /, \wedge, \vee$ auf bestimmte Objekte überhaupt sinnvoll ist und was sie im Zusammenhang mit diesem Objekt überhaupt bedeutet. So wird zum Beispiel ein Compiler Ausdrücke wie $1 \vee 5$ oder a/b als unzulässig erklären, wenn a und b als boolesche Werte deklariert sind. Ausdrücke wie $1+2$ und $\frac{1}{2}+\frac{2}{3}$ sind dagegen zulässig, werden jedoch trotz des gleichen Operationssymbols zur Ausführung verschiedener Operationen führen. Die zulässigen Operationen auf den Elementen eines Datentyps müssen daher als untrennbarer Bestandteil dieses Typs angesehen werden. Wir fassen diese Erkenntnis als einen ersten wichtigen Grundsatz zusammen:

Ein Typ besteht aus kanonischen Elementen und zulässigen Operationen auf seinen Elementen.

Die kanonischen Elemente eines Typs geben an, wie Elemente zu *bilden* sind. Die zulässigen Operationen dagegen beschreiben, auf welche Arten Elemente für eine weitere Verarbeitung *verwendet* werden dürfen.[67]

Diese Charakterisierung von Typen ist zum Teil noch semantischer Natur. Da wir Typen jedoch als Bestandteil von Kalkülen verwenden wollen, die sich beim logischen Schließen ausschließlich auf die Syntax eines Ausdrucks stützen können, müssen wir den obigen Grundsatz mithilfe von syntaktischen Konzepten neu formulieren. Dabei stützen wir uns auf den Gedanken, daß jede Operation – ob sie kanonische Elemente erzeugt oder Elemente verwendet – durch einen Term beschrieben wird. Alles, was wir zu tun haben, ist, diese Terme im Zusammenhang mit dem Typkonzept zu klassifizieren in *kanonische Terme des Typs* – also Terme, die als Standardbeschreibung von Elementen gelten *sollen* – und in *nicht-kanonische Terme des Typs* – also Terme, die Elemente eines Typs als Bestandteile verwenden.[68] Für die Beschreibung eines Typs ist es natürlich auch notwendig, ihn zu benennen, wobei wir ebenfalls einen syntaktischen (Typ-)Ausdruck verwenden müssen. Insgesamt erhalten wir also:

Ein Typ wird definiert durch einen Typ-Ausdruck und seine kanonischen und nichtkanonischen Terme.

Welche Arten von Typen sind zu betrachten?

Wir wollen die Typdisziplin in erster Linie dazu verwenden, um eine natürliche Semantik für λ-Terme zu ermöglichen. λ-Terme sollen als Funktionen eines geeigneten Funktionenraumes interpretiert werden können. Wir benötigen also ein Konstrukt der Form $S \to T$, welches den Typ aller Funktionen vom Typ S in den Typ T bezeichnet. Da λ-Terme nur auf zwei Arten gebildet werden können, ist auch die Aufteilung in kanonische und nichtkanonische Terme einfach. Alle λ-Abstraktionen der Form $\lambda x.t$ gelten als kanonisch[69] während alle Applikationen der Form $f\,t$ als nichtkanonische Terme eingestuft werden.

[67] In modernen Programmiersprachen, in denen sich die objektorientierte Denkweise immer weiter durchsetzt, wird dieser Gedanke durch das Konzept der Klasse (Modul Unit o.ä.) realisiert. Hier wird angegeben, welche Operationen zulässig sind, um Elemente der Klasse zu erzeugen bzw. zu verändern, und mit welchen Operationen man auf sie zugreifen kann.

[68] Diese Trennung in kanonische und nichtkanonische Terme entspricht, wie wir später deutlicher sehen werden, der Auftrennung eines Kalküls in Einführungs- und Eliminationsregeln. Einführungsregeln sagen, wie kanonische Elemente eines Typs (Beweise einer Formel) aufzubauen sind, während Eliminationsregel sagen, wie man sie (nichtkanonisch) verwendet.

Außer dem Funktionenraumkonstruktor werden für den λ-Kalkül keine weiteren Typkonstrukte benötigt. Wir erhalten daher zunächst eine sehr einfache Typentheorie. Wir werden natürlich versuchen, für die im Abschnitt 2.3.3 angegebenen Simulationen gängiger Programmierkonstrukte auch entsprechende Typkonstrukte als definitorische Erweiterungen der einfachen Typentheorie zu geben. Dies ist aber, wie wir im Abschnitt 2.4.8 zeigen werden, nur bis zu einer gewissen Grenze möglich.

Wir werden im folgenden nun präzisieren, wie Typkonstrukte zu bilden sind, wie einem λ-Term ein geeigneter Typ zugeordnet werden kann, wie man formal beweisen kann, daß eine solche Zuweisung korrekt ist, und welche Eigenschaften typisierbare λ-Terme besitzen.

2.4.1 Syntax

Die Syntax der einfachen Typentheorie ist denkbar einfach zu definieren, da es nur einen einzigen Konstruktor gibt. In ihrem Aufbau folgt sie dem aus der Prädikatenlogik und dem λ-Kalkül bekannten Schema.

Definition 2.4.1 (Typ-Ausdrücke)

Es sei $\mathcal{V}$ ein Alphabet von Variablen(-symbolen) und $\mathcal{T}$ ein Alphabet von Bereichssymbolen (Typen).

Typ-Ausdrücke – kurz Typen – sind induktiv wie folgt definiert.

- *Jede Variable $T \in \mathcal{T}$ ist ein (atomarer) Typ.*
- *Sind S und T beliebige Typen, so ist auch $S \to T$ ein Typ.*
- *Ist T ein beliebiger Typ, dann ist (T) ein Typ*

Der Operator $\to$ ist rechtsassoziativ

Objekt-Ausdrücke sind λ-Terme im Sinne von Definition 2.3.2

- *Jede Variable $x \in \mathcal{V}$ ist ein λ-Term.*
- *Ist $x \in \mathcal{V}$ eine Variable und t ein beliebiger λ-Term, dann ist $\lambda x\,.\,t$ ein λ-Term.*
- *Sind t und f beliebige λ-Terme, dann ist $f\,t$ ein λ-Term.*
- *Ist t ein beliebiger λ-Term, dann ist (t) ein λ-Term.*

Die Applikation bindet stärker als λ-Abstraktion und ist linksassoziativ.

Man beachte hierbei wiederum, daß Variablen- und Typsymbole nicht unbedingt aus einzelnen Buchstaben bestehen müssen und daß sich die Alphabete $\mathcal{V}$ und $\mathcal{T}$ überlappen dürfen. Meist geht es aus dem Kontext hervor, um welche Art von Symbolen es sich handelt. Um die Unterscheidung zu erleichtern, benutzen wir Großbuchstaben wie `S,T,X` etc. für Typsymbole und Kleinbuchstaben wie `x,y,z,f,g` für Objektvariablen.

Die Rechtsassoziativität des Funktionenraumkonstruktes und die Linksassoziativität der Applikation ergibt sich aus dem gewünschten Zusammenhang zwischen Typen und λ-Termen. Ein ungeklammerter Typ der Form $S_1 \to S_2 \to T$ sollte zu einem ungeklammerten λ-Term der Form $\lambda x_1\,.\,\lambda x_2\,.\,t$ passen, der notwendigerweise rechtsassoziativ zu klammern ist. Wendet man nun eine Funktion $f \in S_1 \to S_2 \to T$ auf zwei Argumente t und u an, so erwartet man, daß t ein Element von S_1, u eines von S_2 ist und daß als Ergebnis ein Element von T herauskommt. Ein rechtsassoziative Klammerung $f\,(t(u))$ würde dagegen verlangen, daß $t(u)$ ein Element von $S_1 \to S_2$ ist, was nur zu einer Linksassoziativität von $\to$ paßt.

[69] Es macht hierbei wenig Sinn, zu fordern, daß t in Normalform sein muß, da dies die Auftrennung zwischen kanonischen und nichtkanonischen Termen sehr verkomplizieren würde. Wir werden in späteren Abschnitten und besonders im Kapitel 3 sehen, daß diese grobe Einteilung auch wichtig für eine effektive Begrenzung von Reduktionsstrategien (*lazy evaluation*) ist, ohne die ein formales Schließen über Programme ständig durch irrelevante Terminierungsprobleme belastet würde.

2.4.2 Semantik und Typzugehörigkeit

Bei der Definition der Typ- und Objekt-Ausdrücke hatten wir eine sehr einfache und natürliche Semantik vor Augen, die wir im folgenden nur informal beschreiben wollen. Typausdrücke sollen *Mengen* beschreiben und das Funktionssymbol $\rightarrow$ soll, wie in der gewöhnlichen Mathematik, einen *Funktionenraum* charakterisieren.

In Anlehnung an Definition 2.2.8 (Seite 26) interpretieren wir jedes Typsymbol $T \in \mathcal{T}$ durch eine Teilmenge $\iota(T) \subseteq \mathcal{U}$. Jeder Funktionentyp $S \rightarrow T$ wird dann durch den entsprechenden Funktionenraum interpretiert:

$$\iota(S \rightarrow T) = \iota(S) \rightarrow \iota(T)$$

Im folgenden werden wir λ-Termen einen Typ zuordnen. Terme, bei denen dies möglich ist, lassen sich ebenfalls sehr leicht interpretieren. Wird einem Term t der Typ T zugeordnet, dann können wir t als ein Element $\iota(t) \in \iota(T)$ interpretieren.

Dieser semantische Zusammenhang läßt sich verhältnismäßig einfach sicherstellen. Jede Variable x muß zu einem Typ T gehören. Jede λ-Abstraktion $\lambda x.t$ muß zu einem Raum $S \rightarrow T$ gehören, wobei x zu S und t zu T gehört. Auf diese Art wird garantiert, daß $\lambda x.t$ als Funktion eines entsprechenden Funktionenraums $\iota(S) \rightarrow \iota(T)$ interpretiert werden kann. Schließlich gehört eine Applikation $f\,t$ zum Typ T, wenn t zu einem Typ S gehört und f zu $S \rightarrow T$. In diesem Fall wird t immer durch einen Wert interpretiert, welcher im Definitionsbereich der durch f dargestellten Funktion liegt. Wir wollen wir diesen Zusammenhang zunächst an einem Beispiel erläutern.

Beispiel 2.4.2

Wir versuchen, dem Term λ`f.`λ`x. f x` einen Typ zuzuordnen.

Die Variable `x` muß zu einem Typ gehören, den wir der Einfachheit halber zunächst mit `S` bezeichnen. Da `f` auf `x` angewandt wird, muß `f` einen Typ der Gestalt `S`$\rightarrow$`T` haben, wobei `T` ebenfalls noch nicht näher festgelegt ist. Insgesamt hat der Ausdruck `f x` dann den Typ `T`.

Nun haben wir die Typen der Variablen `f` und `x` im Prinzip bereits festgelegt. Damit ergibt sich `S`$\rightarrow$`T` als Typ des Ausdrucks λ`x. f x` und `(S`$\rightarrow$`T)` $\rightarrow$ `(S`$\rightarrow$`T)` als Typ des gesamten Ausdrucks.

Dieses Beispiel zeigt, daß der Typ eines λ-Terms keineswegs eindeutig festliegt. Abgesehen davon, daß wir statt `S` und `T` auch andere Variablennamen hätten verwenden können, könnten wir an ihrer Stelle auch komplexere Typausdrücke einsetzen. Es wäre durchaus korrekt, dem Ausdruck λ`f.`λ`x. f x` den Typ

`(S`$\rightarrow$`(T`$\rightarrow$`S))` $\rightarrow$ `(S`$\rightarrow$`(T`$\rightarrow$`S)))`

oder den Typ

`(S`$\rightarrow$`S)` $\rightarrow$ `(S`$\rightarrow$`S)`

zuzuordnen. Beide Ausdrücke sind aber unnötig spezialisiert: der erste gibt zu viel Struktur vor und der zweite identifiziert die Ein- und Ausgabetypen von `f` miteinander. Üblicherweise versucht man, einem λ-Term das einfachste (allgemeinste!) aller Typschemata zuzuordnen, also eines, was durch seine einfache Struktur noch die meisten Freiheitsgrade läßt. Ein solches Schema nennt man *prinzipielles Typschema*.

Wir haben am Ende des vorigen Abschnitts gesehen, daß es nicht möglich ist, der Klasse aller λ-Terme eine einfache mengentheoretische Sematik zu geben. Da Typen aber eine einfache Semantik geradezu suggerieren, muß es λ-Terme geben, die nicht *typisierbar* sind. Auch hierfür wollen wir ein Beispiel geben.

Beispiel 2.4.3

Wir betrachten den Term λ`x. x x`.

Die Variable `x` muß zu einem Typ S gehören. Da `x` aber auch auf `x` angewandt wird, muß `x` einen Typ der Gestalt $S \rightarrow T$ haben. Um also dem Term λ`x. x x` einen Typ zuordnen zu können, müssen wir Typen S und T bestimmen, welche die Gleichung $S = S \rightarrow T$ erfüllen. Da aber (semantisch) keine Menge identisch ist mit ihrem eigenen Funktionenraum, wird dies nicht möglich sein.

Nach diesen Beispielen dürfte die präzise Definition der Typzugehörigkeit recht naheliegend sein.

Definition 2.4.4 (Typzugehörigkeit)

Typzugehörigkeit ist eine Relation $\in$ zwischen Objekt-Ausdrücken und Typausdrücken, die induktiv wie folgt definiert ist.

- $x \in T$, *falls* $x \in \mathcal{V}$ *eine Variable und* T *ein beliebiger Typ ist.*
- $\lambda x\,.\,t \in S \to T$, *falls* $t \in T$ *gilt, wann immer* $x \in S$ *ist.*
- $f\,t \in T$, *falls es einen Typ* S *gibt mit* $f \in S \to T$ *und* $t \in S$.
- $(t) \in T$, *falls* $t \in T$
- $t \in (T)$, *falls* $t \in T$

Ein λ-Term t heißt typisierbar, wenn es einen Typ T gibt mit $t \in T$.

Das prinzipielle Typschema eines typisierbaren λ-Terms t ist der kleinste[70] *Typ T, für den $t \in T$ gilt.*

In der obigen Definition haben wir das Konzept der Typzugehörigkeit durch rein syntaktische Mittel beschrieben. Dies erlaubt uns, Typzugehörigkeit durch symbolische Manipulationen wie z.B. durch den im Abschnitt 2.4.3 vorgestellten Kalkül zu überprüfen. Dennoch hängt der Typ eines λ-Terms t nicht von seinem syntaktischen Erscheinungsbild, sondern nur von seinen *Wert* ab. λ-Terme, die semantisch gleich (im Sinne von Definition 2.3.24) sind, gehören auch zum gleichen Typ. Damit untermauert Definition 2.4.4 tatsächlich die intuitive Interpretation von λ-Termen als Funktionen eines bestimmten Funktionenraumes.

Satz 2.4.5

Es seien t und t' beliebige typisierbare λ-Terme und T ein Typausdruck.

Wenn t und t' semantisch gleich sind, dann gilt $t \in T$ genau dann, wenn $t' \in T$ gilt.

Beweis:

Wir zeigen zunächst, daß die Typzugehörigkeit bei Reduktionen erhalten bleibt.

Aus $t \xrightarrow{\beta} t'$ folgt: $t \in T$ gilt genau dann, wenn $t' \in T$ gilt.

Wir beweisen dies durch eine Induktion über die Termstruktur (Tiefe) von t

- Ist t ein Term der Tiefe 1, so ist t eine Variable $x \in \mathcal{V}$. Da t bereits in Normalform ist, muß ein Term t' mit $t \xrightarrow{*} t'$ α-konvertibel[71] zu t, also ebenfalls eine Variable sein. Gemäß Definition 2.4.4 gilt dann $t \in T$ und $t' \in T$ für beliebige Typausdrücke T.
- Es sei für alle Terme u der Tiefe n, alle λ-Terme u' und alle Typen S gezeigt
  aus $u \xrightarrow{\beta} u'$ folgt: $u \in S$ gilt genau dann, wenn $u' \in S$ gilt.
- Es sei t' ein Term der Tiefe $n{+}1$ und es gelte $t \xrightarrow{*} t'$.
  - Falls t die Gestalt $\lambda x\,.\,u$ hat, so muß t' die Form $\lambda x\,.\,u'$ haben, wobei u' ein λ-Term mit $u \xrightarrow{\beta} u'$ ist. Gilt nun $t \in T$, so muß T von der Form $S_1 \to S_2$ sein und es ist $u \in S_2$, wann immer $x \in S_1$. Aufgrund der Induktionsannahme gilt somit $u' \in S_2$, wann immer $x \in S_1$ ist und somit $t' \equiv \lambda x\,.\,u' \in S_1 \to S_2 \equiv T$. Da die gleiche Argumentationskette auch umgekehrt geführt werden kann, folgt: $t \in T$ gilt genau dann, wenn $t' \in T$ gilt.
  - Falls t die Gestalt $(\lambda x\,.\,u)\ v$ hat und t' durch Reduktion des äußeren Redex entsteht, so ist $t' \equiv u[v/x]$. Gilt nun $t \in T$, so gibt es einen Typ S mit $\lambda x\,.\,u \in S \to T$ und $v \in S$. Nach Definition 2.4.4 ist nun $u \in T$, wann immer $x \in S$ gilt. Wegen $v \in S$ folgt hieraus $t' \equiv u[v/x] \in T$.
    Ist umgekehrt $t' \equiv u[v/x] \in T$, so gibt es einen Typ S mit $v \in S$ und der Eigenschaft, daß $u \in T$ ist, wann immer $x \in S$ gilt. Damit folgt $t \equiv (\lambda x\,.\,u)\ v \in T$.

[70] Unter dem kleinsten Typ verstehen wir einen Typ, dessen struktureller Aufbau im Sinne von Definition 2.4.1 mit dem geringsten Aufwand und der geringsten Spezialisierung verbunden ist. Man könnte auch sagen, daß jeder andere Typ T', für den $t \in T'$ gilt, sich durch Instantiierung der Typvariablen aus dem prinzipiellen Typschema ergeben muß.

[71] Genau besehen kann es einen Term t' mit $t \xrightarrow{\beta} t'$ (genau ein Reduktionsschritt!) nicht geben und damit folgt die Behauptung aus der Ungültigkeit der Annahme

– Falls t die Gestalt $f\,u$ hat, und t' nicht durch äußere Reduktion (wie oben) entsteht, so hat t' die Gestalt $f\,u'$, wobei u' ein λ-Term ist, für den $u \xrightarrow{\beta} u'$ gilt, oder die Form $f'\,u$, wobei $f \xrightarrow{\beta} f'$ gilt. Nach Induktionsannahme gehören u und u' (bzw. f und f') nun zum gleichen Typ und wie im ersten Fall folgt die Behauptung

Damit ist die Aussage bewiesen. Durch eine Induktion über die Länge der Ableitung kann man hieraus schließen:

Aus $t \xrightarrow{*} t'$ folgt: $t \in T$ gilt genau dann, wenn $t' \in T$ gilt.

Es seien nun t und t' semantisch gleich. Dann gibt es gemäß Definition 2.3.24 einen λ-Term u mit $t \xrightarrow{*} u$ und $t' \xrightarrow{*} u$. Es folgt: $t \in T$ gilt genau dann, wenn $u \in T$ gilt und dies ist genau dann der Fall, wenn $t' \in T$ gilt. □

2.4.3 Formales Schließen über Typzugehörigkeit

In Definition 2.4.4 haben wir bereits recht genaue Vorschriften dafür angegeben, wie einem λ-Term ein Typausdruck zuzuweisen ist. Wir wollen diese Vorschriften nun in der Form von syntaktischen Regeln präzisieren, um innerhalb eines formalen Kalküls *beweisen* zu können, daß eine gegebene Typzuweisung korrekt ist. Da wir die einfache Typisierung im Kapitel 3 zu einer Formalisierung beliebiger Programm*eigenschaften* ausbauen werden, ist dieser sehr einfache Kalkül ein wesentlicher Schritt in Richtung auf ein formales System zum automatisierten Schließen über die *Eigenschaften* von Programmen.

Wie in den vorhergehenden Abschnitten werden wir für die einfache Typentheorie einen Sequenzenkalkül aufstellen, dessen Regeln die semantischen Anforderungen wiederspiegeln. Wir müssen hierzu das in Definition 2.2.12 gegebene Grundkonzept geringfügig modifizieren, da nun in den Hypothesen die Deklarationen die zentrale Rolle spielen werden und in der Konklusion eine Typisierung anstelle der Formeln auftreten wird. Die Regeln, welche *hinreichende* Bedingungen an die Typzugehörigkeit eines λ-Terms angeben müssen, ergeben sich ganz natürlich aus Definition 2.4.4.

Ein Term der Form $\lambda x\,.\,t$ kann nur zu einem Typ gehören, der die Gestalt $S \to T$ hat. Um nachzuweisen, daß dies tatsächlich der Fall ist, nehmen wir an, daß x vom Typ S ist und zeigen dann, daß $t \in T$ gilt. Hierzu müssen wir die Hypothesenliste um die Deklaration $x\!:\!S$ erweitern und, falls x schon deklariert war, eine entsprechende Umbenennung vornehmen, was zu folgender Regel führt.

$$\Gamma \vdash \lambda x\,.\,t \in S \to T \quad \textbf{by } \texttt{???\_i}$$
$$\Gamma,\ x'\!:\!S \vdash t[x'/x] \in T$$

Bei Anwendung einer solchen Regel wandert der Term S ungeprüft in die Hypothesenliste, selbst wenn er kein korrekter Typ-Ausdruck ist. Natürlich könnte man dies vermeiden, indem man zu Beginn eines Beweises überprüft, ob der Ausdruck auf der rechten Seite des $\in$-Symbols überhaupt einen Typ-Ausdruck darstellt. Für die einfache Typentheorie ist ein solcher Test ohne Probleme durchführbar. Die Erweiterungen der Typentheorie, die wir in Kapitel 3 besprechen werden, lassen dies jedoch nicht mehr zu.[72] Deshalb werden wir die Überprüfung der *Wohlgeformtheit* eines Typausdrucks ebenfalls in unseren Kalkül mit aufnehmen. Formal geschieht dies dadurch, daß wir ein neues Symbol $\mathbb{U}$ einführen, welches als *Universum aller Typen* interpretiert wird. Die Wohlgeformtheit eines Typausdrucks S werden wir dann nachweisen, indem wir zeigen, daß S zu diesem Universum gehört – also, daß $S \in \mathbb{U}$ gilt. Entsprechend ergänzen wir die obige Regel um ein weiteres Unterziel: wird $x\!:\!S$ als Deklaration in die Hypothesenliste aufgenommen, so ist zu zeigen, daß $S \in \mathbb{U}$ gilt.

$$\Gamma \vdash \lambda x\,.\,t \in S \to T \quad \textbf{by } \texttt{lambda\_i}$$
$$\Gamma,\ x'\!:\!S \vdash t[x'/x] \in T$$
$$\Gamma \vdash S \in \mathbb{U}$$

Durch die Hinzunahme des Universums $\mathbb{U}$ entfällt übrigens auch die Notwendigkeit, in formalen Beweisen zwischen Variablensymbolen aus $\mathcal{V}$ und Typsymbolen aus $\mathcal{T}$ zu unterscheiden. Eine Deklaration der Form $x\!:\!\mathbb{U}$ deklariert x als als *Typsymbol*, während jede andere Deklaration $x\!:\!Y$ das Symbol x als *Variable* deklariert. Wir können daher ab sofort für alle Variablen beliebige Bezeicher verwenden.

[72] Unter anderem wird die Möglichkeit entstehen, Abstraktionen und Applikationen in Typausdrücken zu verwenden und diese zu reduzieren. Die Frage, ob ein Ausdruck zu einem Typausdruck reduziert werden kann, ist jedoch unentscheidbar.

$\Gamma \vdash \lambda x.t \in S{\to}T$ **by** `lambda_i`* $\Gamma,\ x'{:}S \vdash t[x'/x] \in T$ $\Gamma \vdash S \in \mathbb{U}$	$\Gamma \vdash f\,t \in T$ **by** `apply_i` S $\Gamma \vdash f \in S{\to}T$ $\Gamma \vdash t \in S$
$\Gamma \vdash S{\to}T \in \mathbb{U}$ **by** `function_i` $\Gamma \vdash S \in \mathbb{U}$ $\Gamma \vdash T \in \mathbb{U}$	$\Gamma,\ x{:}T,\ \Delta \vdash x \in T$ **by** `declaration` i

**: Die Umbenennung $[x'/x]$ erfolgt, wenn x in Γ frei vorkommt.*

Abbildung 2.8: Sequenzenkalkül für die einfache Typentheorie

Die Regel für die Applikation ist sehr einfach anzugeben: um $f\,t \in T$ zu zeigen, müssen wir zeigen, daß das Argument t von f zu einem bestimmten Typ S gehört und daß f eine Funktion von S nach T ist. Da der Typ S nicht automatisch[73] bestimmt werden kann, muß er als Parameter der Regel mit angegeben werden.

$\Gamma \vdash f\,t \in T$ **by** `apply_i` S
$\Gamma \vdash f \in S{\to}T$
$\Gamma \vdash t \in S$

Durch die Aufnahme der Wohlgeformtheit eines Typausdrucks in den Beweis ergibt sich die Notwendigkeit einer weiteren Regel. Um nachzuweisen, das $S{\to}T$ wohlgeformt ist, müssen wir nur nachweisen, daß dies für S und T gilt. Andere Möglichkeiten, Typausdrücke mit Regeln zu zerlegen, gibt es nicht.

$\Gamma \vdash S{\to}T \in \mathbb{U}$ **by** `function_i`
$\Gamma \vdash S \in \mathbb{U}$
$\Gamma \vdash T \in \mathbb{U}$

Die drei (Einführungs-)Regeln zur Analyse von Ausdrücken werden ergänzt um eine Regel, die besagt, daß man Deklarationen der Form $x{:}T$ zum Nachweis von $x{\in}T$ heranziehen kann. Diese Regel ist eine leichte Modifikation der Regel `hypothesis` aus Abschnitt 2.2.4. Sie kann sowohl auf Objektvariablen als auch auf Typvariablen angewandt werden.

$\Gamma,\ x{:}T,\ \Delta \vdash x \in T$ **by** `declaration` i

Alle Inferenzregeln der einfachen Typentheorie sind in Abbildung 2.8 zusammengefaßt. Da sich mittlerweile im Bezug auf die konkrete Ausgestaltung von Sequenzen gegenüber der Definition 2.2.12 (Seite 30) eine Reihe von Änderungen ergeben haben, geben wir der Vollständigkeit halber die entsprechend modifizierte Definition für die einfache Typentheorie.

Definition 2.4.6 (Deklaration, Typisierungen und Sequenzen)

1. *Eine Deklaration hat die Gestalt $x{:}X$ (Variablendeklaration) oder $T{:}\mathbb{U}$ (Typdeklaration), wobei x und T beliebige Bezeichner, X ein Ausdruck und $\mathbb{U}$ ein festes Symbol (Universum) ist.*

2. *Eine Typisierung hat die Gestalt $t{\in}T$ oder $T{\in}\mathbb{U}$, wobei t und T beliebige Ausdrücke sind.*

3. *Eine Sequenz hat die Gestalt $\Gamma \vdash C$, wobei die Hypothesenliste Γ eine Liste von durch Komma getrennten Deklarationen und die Konklusion C eine Typisierung ist.*

4. *Eine Hypothesenliste ist rein, wenn jeder Bezeichner genau einmal deklariert ist und jeder Bezeichner T, der auf der rechten Seite einer Deklaration erscheint, zuvor durch $T{:}\mathbb{U}$ deklariert wurde.*

5. *Eine Initialsequenz ist eine Sequenz der Gestalt $\Gamma \vdash t{\in}T$, deren Hypothesenliste rein ist, eine Typdeklaration für alle in T vorkommenden Bezeichner enthält und sonst keine Variablen deklariert.*

[73] Im Falle der einfachen Typentheorie könnte man für S durch den in Abschnitt 2.4.4 angegebenen Typechecking Algorithmus das prinzipielle Typschema bestimmen. Dies läßt sich jedoch nur begrenzt verallgemeinern und würde zudem die schematische Regel in eine Regel verwandeln, die nur im Zusammenhang mit diesem Algorithmus erklärt werden kann.

```
T:𝕌 ⊢ (λf.λx.f x)(λz.z) ∈ T→T
by apply_i  T→T
|\
| T:𝕌 ⊢ λf.λx.f x ∈ (T→T) → (T→T)
| by lambda_i
| |\
| | T:𝕌, f:T→T ⊢ λx.f x ∈ T→T
| | by lambda_i
| | |\
| | | T:𝕌, f:T→T, x:T ⊢ f x ∈ T
| | | by apply_i  T
| | | |\
| | | | T:𝕌, f:T→T, x:T ⊢ f ∈ T→T
| | | | by declaration 2
| | |  \
| | |   T:𝕌, f:T→T, x:T ⊢ x∈T
| | |   by declaration 3
| |  \
| |   T:𝕌, f:T→T ⊢ T∈𝕌
| |   by declaration 1
|  \
|   T:𝕌 ⊢ T→T ∈ 𝕌
|   by function_i
|   |\
|   | T:𝕌 ⊢ T∈𝕌
|   | by declaration 1
|    \
|     T:𝕌 ⊢ T∈𝕌
|     by declaration 1
 \
  T:𝕌 ⊢ λz.z ∈ T→T
  by lambda_i
  |\
  | T:𝕌, z:T ⊢ z ∈ T
  | by declaration 2
   \
    T:𝕌 ⊢ T∈𝕌
    by declaration 1
```

Abbildung 2.9: Sequenzenbeweis für eine Typisierung von `(λf.λx. f x)(λz.z)`

Der eigentliche Beweisbegriff wird aus Definition 2.2.12 unverändert übernommen. Dementsprechend gilt eine Typisierung $t \in T$ als bewiesen, wenn es einen vollständigen Beweis für die (eindeutig bestimmte) zu $t \in T$ gehörende Initialsequenz gibt. Wir wollen dies an einem einfachen Beispiel illustrieren.

Beispiel 2.4.7

Wir wollen zeigen, daß `(λf.λx. f x)(λz.z) ∈ T→T` gilt.

Die äußere Operation des gegebenen Terms ist eine Applikation. Wir müssen daher als erstes die Regel `apply_i` angeben und hierbei einen Typen für das Argument `λz.z` angeben. Eine solche Angabe erfordert eine Reihe von Vorüberlegungen. Der Typ für `λz.z` wird gleichzeitig der Typ der Variablen `f` werden und `λx. f x` wird den Typen `T→T` erhalten müssen. Sinnvoll ist es also `x∈T` und `f∈T→T` anzusetzen. Diese Überlegungen führen zur Anwendung der Regel `apply_i T→T`, welche uns folgende Teilziele hinterläßt.

`λf.λx. f x ∈ (T→T) → (T→T)`

und

`λz.z ∈ T→T`

Während das zweite Teilziel nun direkt durch `lambda_i` und die Regel `declaration` bewiesen werden kann, ist für das erste eine Reihe weiterer Schritte erforderlich. Hierbei bestimmt jeweils der äußere Operator (λ-Abstraktion, Applikation oder Funktionenraumbildung) eindeutig die anzuwendende Regel und nur bei der Applikation müssen wieder Parameter angegeben werden. Diese ergeben sich aber aus obigen Vorbetrachtungen und führen zu dem in Abbildung 2.9 angegebenen Beweis.

Man beachte, daß der in Abbildung 2.8 angegebene Kalkül ausschließlich das Schließen über Typzugehörigkeit unterstützt. Die semantische Gleichheit von λ-Termen wird gemäß Theorem 2.4.5 zwar respektiert, aber das Schließen über Werte ist nicht in den Kalkül der Typentheorie integriert. Dies werden wir im Kapitel 3 nachholen, wenn wir Logik, Berechnung und Typdisziplin in einem Kalkül vereinen.

2.4.4 Ein Typechecking Algorithmus

Mit dem im vorhergehenden Abschnitt angegebenen Kalkül können wir eine vorgegebene Typisierung eines λ-Terms *überprüfen*. Wie aber *finden* wir eine solche Typisierung? Ist es möglich, den Typ eines λ-Terms durch eine Analyse der in Definition 2.4.4 angegebenen Vorschriften automatisch zu bestimmen?

Für die einfache Typentheorie und geringfügige Erweiterungen davon kann diese Frage positiv beantwortet werden. Es ist in der Tat möglich, einen (immer terminierenden) Algorithmus anzugeben, welcher zu jedem typisierbaren Term das prinzipielle Typschema angibt. Dieser Algorithmus basiert auf einem Typechecking Algorithmus, der von Hindley und Milner[74] entwickelt wurde und einen wesentlichen Bestandteil des reichhaltigen Typkonzepts der funktionalen Programmiersprache ML[75] bildet.

Die Grundidee dieses Verfahrens ist simpel. Um einem Term t einen Typ T zuzuweisen, ordnen wir t einfach einen Typen zu und beginnen mit der Typprüfung. Wann immer der Beweis dies nötig macht, werden wir T weiter verfeinern, bis die Prüfung erfolgreich beendet wurde oder offensichtlich ist, daß kein Typ zugeordnet werden kann. Bevor wir den Algorithmus im Detail vorstellen wollen wir ihn an einem Beispiel erläutern.

Beispiel 2.4.8

Um dem Term λ`f.`λ`x. f x` einen Typ zuzuordnen (vgl. Beispiel 2.4.2), beginnen wir mit der Behauptung

λ`f.`λ`x. f x` $\in$ $\mathtt{X}_0$

Aufgrund der λ-Abstraktion muß $\mathtt{X}_0$ die Gestalt $\mathtt{X}_1 \to \mathtt{X}_2$ haben und

λ`x. f x` $\in$ $\mathtt{X}_2$

gelten, wobei `f` vom Typ $\mathtt{X}_1$ ist. Genauso folgern wir, daß $\mathtt{X}_2$ die Gestalt $\mathtt{X}_3 \to \mathtt{X}_4$ haben muß und

`f x` $\in$ $\mathtt{X}_4$

gilt, wobei `x` vom Typ $\mathtt{X}_3$ ist. Da der Typ des Argumentes von `f` feststeht, muß $\mathtt{X}_1$ – der Typ von `f` – identisch mit $\mathtt{X}_3 \to \mathtt{X}_4$ sein. Damit sind alle Bestandteile des Terms typisiert und es ist

λ`f.`λ`x. f x` $\in$ $(\mathtt{X}_3 \to \mathtt{X}_4) \to (\mathtt{X}_3 \to \mathtt{X}_4)$

Da die Namen der Typvariablen $\mathtt{X}_3$ und $\mathtt{X}_4$ nur Platzhalter im prinzipiellen Typschema sind, haben wir nun den Typ von λ`f.`λ`x. f x` bestimmt.

Das folgende Beispiel zeigt, wie das angedeutete Verfahren feststellt, daß eine Typisierung unmöglich ist.

Beispiel 2.4.9

Um dem Term λ`x.x x` einen Typ zuzuordnen (vergleiche Beispiel 2.4.3), beginnen wir mit λ`x.x x` $\in$ $\mathtt{X}_0$. Aufgrund der λ-Abstraktion muß $\mathtt{X}_0$ die Gestalt $\mathtt{X}_1 \to \mathtt{X}_2$ haben und `x x` $\in$ $\mathtt{X}_2$ gelten, wobei `x` vom Typ $\mathtt{X}_1$ ist. Da der Typ des Argumentes von `x` feststeht, muß $\mathtt{X}_1$ – der Typ von `x` – identisch mit $\mathtt{X}_1 \to \mathtt{X}_2$ sein. Der Versuch, $\mathtt{X}_1$ und $\mathtt{X}_1 \to \mathtt{X}_2$ durch *Unifikation*[76] gleichzumachen, wird jedoch fehlschlagen. Daher ist der Term λ`x.x x` nicht typisierbar.

[74] Die entsprechenden Originalarbeiten sind in [Hindley, 1969, Milner, 1978, Damas & Milner, 1982] dokumentiert. Weitere Darstellungen dieses Verfahrens im Kontext verschiedener Formalismen findet man in [Hindley, 1983, Hindley & Seldin, 1986, Cardone & Coppo, 1990].

[75] Die Metasprache des NuPRL Systems – eine der ersten Versionen der Programmiersprache ML – enthält diesen Algorithmus. Es ist daher relativ einfach, das prinzipielle Typschema eines λ-Terms mit dem System zu finden. Um zum Beispiel den Term λ`f.`λ`x. f x` zu typisieren, muß man diesen nur im ML top-loop eingeben. Man beachte hierbei jedoch, daß es sich um einen ML-Ausdruck und nicht etwa einen objektsprachlichen Ausdruck der Typentheorie von NuPRL handelt. Man gebe also ein:

`\f.\x. f x` (ohne 'Term-quotes')

Im Emacs Fenster wird dann als Antwort die Typisierung `\f.\x. f x : (*->**) -> (*->**)` erscheinen, wobei `*` und `**` Platzhalter für Typvariablen sind.

Algorithmus `TYPE-SCHEME-OF(`t`)`: *(t: (geschlossener) λ-Term)*

Initialisiere die Substituition σ (eine globale Variable) als identische Abbildung und rufe `TYPE-OF([],`t`)` auf.

Falls der Algorithmus fehlschlägt ist t nicht typisierbar.
Andernfalls ist das Resultat T das prinzipielle Typschema von t.

Hilfsalgorithmus `TYPE-OF(`*Env*`,`t`)`: *(Env: Liste der aktuellen Annahmen, t: aktueller Term)*

- Falls t die Gestalt $\underline{x}$ hat, wobei $x \in \mathcal{V}$ eine Variable ist:
  Suche in *Env* die (eindeutige) Deklaration der Gestalt $x:T$ und gebe T aus.
- Falls t die Gestalt $\underline{f\ u}$ hat:
  Bestimme S_1:=`TYPE-OF(`*Env*`,`f`)` und dann S_2:=`TYPE-OF(`*Env*`,`u`)`*. Wähle eine neue Typvariable $\mathtt{X}_{i+1}$ und versuche, $\sigma(S_1)$ mit $S_2 \to \mathtt{X}_{i+1}$ zu unifizieren. Falls die Unifikation fehlschlägt, breche mit einer Fehlermeldung ab. Andernfalls ergänze σ um die bei der Unifikation erzeugte Substitution σ' ($\sigma := \sigma' \circ \sigma$). Ausgabe ist $\sigma(\mathrm{X}_{i+1})$
- Falls t die Gestalt $\underline{\lambda x\,.\,u}$ hat: Wähle eine neue Typvariable $\mathtt{X}_{i+1}$
  Bestimme S_1:= $\begin{cases} \texttt{TYPE-OF(}\mathit{Env}\cdot\texttt{[}x\texttt{:X}_{i+1}\texttt{],}u\texttt{)} & \text{falls } x \text{ nicht in } \mathit{Env} \text{ vorkommt}^* \\ \texttt{TYPE-OF(}\mathit{Env}\cdot\texttt{[}x'\texttt{:X}_{i+1}\texttt{],}u[x'/x]\texttt{)} & \text{sonst } (x' \text{ neue Objektvariable})^* \end{cases}$
  Ausgabe ist $\sigma(\mathrm{X}_{i+1}) \to S_1$

*: Beachte, daß σ beim Aufruf von `TYPE-OF` ergänzt worden sein kann.

Abbildung 2.10: Typechecking Algorithmus für die einfache Typentheorie

Der in Abbildung 2.10 vorgestellte Algorithmus ist beschrieben für geschlossene λ-Terme, also λ-Terme ohne freie Variablen. Die Typen eventuell vorkommender freier Variablen können dadurch bestimmt werden, daß man den Term durch entsprechende λ-Abstraktionen abschließt.

Satz 2.4.10 (Hindley / Milner)

Es ist effektiv entscheidbar, ob ein λ-Term typisierbar ist oder nicht.

Beweis: Der in Abbildung 2.10 Algorithmus bestimmt das prinzipielle Typschema eines gegebenen λ-Terms t oder endet mit einer Fehlermeldung, falls keine Typisierung möglich ist. Einen Beweis für seine Korrektheit, der sich eng an den Kalkül aus Abschnitt 2.4.3 anlehnt findet man in [Milner, 1978]. □

Da der Hindley-Milner Algorithmus für jeden typisierbaren λ-Term das prinzipielle Typschema bestimmen kann, könnte man innerhalb der einfachen Typentheorie eigentlich auf den Typüberprüfungskalkül verzichten. Anstatt $t \in T$ zu beweisen, könnte man genauso gut das Typschema von t zu bestimmen und zu vergleichen, ob T diesem Schema entspricht. Da dieses Verfahren jedoch nicht bei allen notwendigen Erweiterungen der Typentheorie funktioniert, muß man sich darauf beschränken, den Hindley-Milner Algorithmus als unterstützende Strategie einzusetzen, die in vielen – aber eben nicht in allen – Fällen zum Ziele führt.

Wir wollen den (zeitlichen) Ablauf des Algorithmus an zwei Beispielen erläutern. Die Beispiele zeigen, wie ein Term durch den Algorithmus zunächst zerlegt wird, durch einen rekursiven Aufruf die Typen der Teilterme ermittelt werden und dann der Typ des zusammengesetzten Terms gebildet wird. Da innerhalb eines rekursiven

[76] *Unifikation* ist ein Verfahren zur Bestimmung einer Substitution σ, welche freie Variablen in zwei vorgegebenen Ausdrücken so ersetzt, daß die beiden Ausdrücke gleich werden. Das allgemeine Verfahren wurde in [Robinson, 1965] wurde ursprünglich für prädikatenlogische Terme entwickelt, ist aber genauso auf Typausdrücke anwendbar, solange Reduktion keine Rolle spielt. Es zerlegt simultan die syntaktische Struktur der beiden Terme solange, bis ein äußerer Unterschied auftritt. An dieser Stelle muß nun die Variable des einen Termes durch den entsprechenden Teilterm des anderen ersetzt werden, wobei ein *Occur*-check sicherstellt, daß die ursprünglichen Variablen nicht in dem eingesetzten Teilterm enthalten sind. Das Verfahren schlägt fehl, wenn eine solche Ersetzung nicht möglich ist.

Aufrufes weitere Zerlegungen stattfinden können, werden im Laufe der Zeit verschiedene Teilterme ab- und wieder aufgebaut. Zur Beschreibung der einzelnen Schritte geben wir in einer Tabelle jeweils die Werte der Variablen *Env*, t und σ beim Aufruf von `TYPE-OF` an, sowie die durchgeführte Unifikation, die im folgenden Schritt zu einer veränderten Substitution σ führt. Die letzte Spalte benennt den berechneten Typ.

Beispiel 2.4.11

1. Typisierung des Terms λ`f.`λ`x. f(f(f x))` mit dem Hindley-Milner Algorithmus:

Env	*Aktueller Term t*	σ	UNIFY	*Typ*
	λ`f.`λ`x. f(f(f x))`			
`f:`X_0	λ`x. f(f(f x))`			
`f:`X_0, `x:`X_1	`f(f(f x))`			
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`f(f x)`			
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`f x`			
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`x`			X_1
`f:`X_0, `x:`X_1	`f x`		$X_0 = X_1 \to X_2$	X_2
`f:`X_0, `x:`X_1	`f(f x)`	$[X_1 \to X_2 / X_0]$	$X_1 \to X_2 = X_2 \to X_3$	X_3
`f:`X_0, `x:`X_1	`f(f(f x))`	$[X_3, X_3, X_3 \to X_3 / X_2, X_1, X_0]$	$X_3 \to X_3 = X_3 \to X_4$	X_4
`f:`X_0	λ`x. f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 / X_3, X_2, X_1, X_0]$		$X_4 \to X_4$
	λ`f.`λ`x. f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 / X_3, X_2, X_1, X_0]$		$(X_4 \to X_4) \to X_4 \to X_4$

2. Typisierung des Terms `(`λ`f.`λ`x. f x) (`λ`x.x)` mit dem Hindley-Milner Algorithmus:

Env	*Aktueller Term t*	σ	UNIFY	*Typ*
	`(`λ`f.`λ`x. f x)(`λ`x.x)`			
	λ`f.`λ`x. f x`			
`f:`X_0	λ`x. f x`			
`f:`X_0, `x:`X_1	`f x`			
`f:`X_0, `x:`X_1	`f`			X_0
`f:`X_0, `x:`X_1	`x`			X_1
`f:`X_0, `x:`X_1	`f x`		X_0 = $X_1 \to X_2$	X_2
`f:`X_0	λ`x. f x`	$[X_1 \to X_2 / X_0]$		$X_1 \to X_2$
	λ`f.`λ`x. f x`	$[X_1 \to X_2 / X_0]$		$(X_1 \to X_2) \to X_1 \to X_2$
	λ`x.x`			
`x:`X_3	`x`			X_3
	λ`x.x`			$X_3 \to X_3$
	`(`λ`f.`λ`x. f x)(`λ`x.x)`	$[X_1 \to X_2 / X_0]$	$(X_1 \to X_2) \to X_1 \to X_2 = (X_3 \to X_3) \to X_4$	$X_3 \to X_3$

Der Term des ersten Beispiels kann im Prinzip wie wie Funktion `twice` aus Beispiel 2.3.29 (Seite 62) auf sich selbst angewandt werden. Ein Ausdruck wie

`(`λ`f.`λ`x. f(f(f x))) (`λ`f.`λ`x. f(f(f x)))`

ist legal und kann auch typisiert werden. Dennoch wird durch die Typisierung die Möglichkeit der Selbstanwendung stark eingeschränkt. Es zeigt sich nämlich, daß jedes der beiden Vorkommen von λ`f.`λ`x. f(f(f x))` anders typisiert wird. Das zweite erhält den Typ `(X`$\to$`X)` $\to$ `X`$\to$`X`, während dem ersten Vorkommen der Typ das `((X`$\to$`X)` $\to$ `X`$\to$`X)` $\to$ `(X`$\to$`X)` $\to$ `X`$\to$`X`, also ein Typ einer wesentlich höheren Ordnung zugewiesen wird. Diese Konsequenz einer Typisierung wird sich besonders stark bei der Rekursion auswirken.

2.4.5 Eigenschaften typisierbarer λ-Terme

Wir wollen nun nachweisen, daß sich durch die Hinzunahme der Typdisziplin zum λ-Kalkül eine Reihe günstiger Eigenschaften ergeben, welche die Auswertung von λ-Termen und das automatische Schließen über ihre Eigenschaften erheblich vereinfachen. Im Gegensatz zum ungetypten λ-Kalkül besitzt jeder typisierbare λ-Term eine Normalform (siehe Abschnitt 2.4.5.1). Dabei ist es unerheblich, welche Reduktionsstrategie man verwendet, denn jede Reduktionsfolge wird terminieren (siehe Abschnitt 2.4.5.2). Das hat zur Folge, daß man in der Typentheorie erheblich effizientere Reduktionsstrategien einsetzen kann. Außerdem ermöglicht diese Eigenschaft auch einen wesentlich einfacheren Beweis für das Church-Rosser Theorem, welches die Eindeutigkeit der Normalform eines λ-Terms zum Inhalt hat (siehe Abschnitt 2.4.5.3). Eine Konsequenz hiervon ist, daß semantische Gleichheit von λ-Termen entscheidbar wird (siehe Abschnitt 2.4.5.4). Insgesamt liefert uns also die Typsdisziplin eine *entscheidbare Theorie totaler Funktionen*.

2.4.5.1 Schwache Normalisierbarkeit

Wir wollen als erstes zeigen, daß jeder typisierbare λ-Term normalisierbar ist – daß es also mindestens eine Reduktionsstrategie gibt, die in jedem Fall zu einer Normalform führt. Aufgrund von Lemma 2.3.33 (Seite 64) wissen wir bereits, daß die Reduktion des jeweils äußersten Redex (call-by-name) immer zum Ziel führt, wenn es überhaupt eine Normalform gibt. Diese Strategie ist allerdings relativ ineffizient. Eine wesentlich bessere Strategie wird uns der Beweis der sogenannten *schwachen Normalisierbarkeit* typisierbarer λ-Terme liefern, da er sich auf kombinatorische Argumente und die Komplexität der Typstruktur eines λ-Terms stützen kann.

Die Grundidee ist verhältnismäßig einfach. Ein Funktionenraum der Gestalt $(S_1 \rightarrow S_2) \rightarrow (T_1 \rightarrow T_2)$ beschreibt Funktionen, welche wiederum einfache Funktionen als Argumente und Ergebnis verwenden. Die Funktionenraumstruktur hat also die *Tiefe* 2, da der Raum in zwei Ebenenen aufgebaut wurde. Wenn wir es nun erreichen würden, durch eine Reduktionsstrategie die Tiefe des Typs eines gegebenen λ-Terms schrittweise zu reduzieren, dann muß diese Reduktionskette irgendwann einmal terminieren.[77] Wir geben hierzu eine Definition der Tiefe von Typausdrücken und typisierbaren λ-Termen.

Definition 2.4.12

Die Tiefe $d(T)$ eines Typausdrucks T ist induktiv wie folgt definiert.

- *$d(T) = 0$, falls T ein atomarer Typ ist ($T \in \mathcal{T}$)*
- *$d(S \rightarrow T) = 1 + max(d(S), d(T))$.*

Die Tiefe eines Redex $(\lambda x.t)\, u$ ist die Tiefe des Typs von $\lambda x.t$.

Die Tiefe eines typisierbaren λ-Terms t ist die maximale Tiefe der in t enthaltenen Redizes (und 0, wenn t keine Redizes enthält).

Wie können wir nun eine Strategie entwickeln, die mit Sicherheit die Tiefe eines λ-Terms verringern wird? Die naheliegenste Idee ist natürlich, ein Redex maximaler Tiefe zu reduzieren, da hierdurch ja ein solches Redex aus dem Term entfernt wird. Diese Überlegung alleine reicht jedoch nicht aus, wie das folgende Beispiel zeigt.

Beispiel 2.4.13

Betrachte den Term `trice ( trice (λx.x) )`, wobei `trice` $\equiv$ `λf.λx. f(f(f x))` (vgl. Beispiel 2.4.11.1).

Da $(X \rightarrow X) \rightarrow (X \rightarrow X)$ der prinzipielle Typ von `trice` ist, folgt gemäß Definition 2.4.12, daß sowohl `trice (λx.x)` als auch `trice ( trice (λx.x) )` Redizes der Tiefe 2 sind. Würden wir nun (entsprechend der Leftmost-Reduktionsstrategie) das äußerste Redex maximaler Tiefe, also `trice ( trice (λx.x) )`, zuerst reduzieren, so erhielten wir

```
λx. (trice (λx.x)) ( (trice (λx.x)) ( (trice (λx.x)) x))
```

also einen Term, welcher nunmehr 3 Redizes der Tiefe 2 enthält.

[77] Bei ungetypten λ-Termen ist dieses Argument nicht anwendbar, da die Tiefe des zugehörigen Funktionenraumes nicht festgestellt werden kann.

Die Reduktion eines Redex maximaler Tiefe kann also dazu führen, daß mehr Redizes maximaler Tiefe entstehen als zuvor vorhanden waren. Mit kombinatorischen Argumenten könnte die Terminierung der eben angedeuteten Vorgehensweise daher nicht bewiesen werden. Dieser unerwünschte Effekt kann jedoch vermieden werden, wenn wir das am weitesten rechts stehende Redex maximaler Tiefe verringern, da dieses keine Teilterme maximaler Tiefe mehr enthält.

Lemma 2.4.14 (Rightmost-Maxdepth Strategie)

Reduziert man in einem typisierbaren λ-Term t das am weitesten rechts stehende Redex maximaler Tiefe, so verringert sich die Anzahl der Redizes maximaler Tiefe.

Beweis: Es sei t ein typisierbarer λ-Term und $r{=}(\lambda x.u)\,v$ das am weitesten rechts stehende Redex maximaler Tiefe. Weder u noch v enthalten ein Redex maximaler Tiefe und die Reduktion von r hat die folgenden Effekte.

- Alle Redizes von t außerhalb von r bleiben unverändert.
- Ein Redex maximaler Tiefe, nämlich r, wird entfernt.
- Der entstehende Term $u[v/x]$ enthält keine Redizes maximaler Tiefe (auch wenn es durchaus möglich ist, daß andere Redizes vervielfältigt werden).

Damit wird durch die Reduktion von r die Anzahl der Redizes maximaler Tiefe verringert, während die Anzahl der Redizes geringerer Tiefe wachsen kann. □

Wir wollen den Effekt der Rightmost-Maxdepth Strategie an dem obigen Beispiel illustrieren.

Beispiel 2.4.15

Wenn wir in `trice ( trice (λx.x) )` das am weitesten rechts stehende Redex maximaler Tiefe, also `trice (λx.x)`, reduzieren, so entsteht der Term

```
trice (λx. (λx.x) ( (λx.x) ( (λx.x) x)))
```

Insgesamt ist hierbei die Anzahl der Redizes gewachsen, aber die Anzahl der Redizes maximaler Tiefe ist gesunken. Reduzieren wir nun wiederum das am weitesten rechts stehende Redex maximaler Tiefe, so wird der Term in seiner Größe zwar gewaltig anwachsen, aber es gibt nur noch Redizes der Tiefe 1.

```
λx. (λx. (λx.x) ( (λx.x) ( (λx.x) x)))
      ( (λx. (λx.x) ( (λx.x) ( (λx.x) x)))
           ( (λx. (λx.x) ( (λx.x) ( (λx.x) x))) x) )
```

Ab nun werden schrittweise alle Redizes der Tiefe 1 abgebaut, wobei keine neuen Redizes mehr entstehen können, da es Redizes der Tiefe 0 nicht geben kann.

Das obige Beispiel deutet bereits an, warum die Rightmost-Maxdepth Strategie terminieren muß. In jedem Schritt verringert sich die Anzahl der Redizes maximaler Tiefe, bis diese auf Null gesunken ist. Anschließend werden die Redizes der nächstgeringeren Teife abgebaut und das Ganze so lange fortgeführt bis kein Redex der mehr übrig ist und der Term in Normalform ist.

Satz 2.4.16 (Terminierung der Rightmost-Maxdepth Strategie)

Zu jedem typisierbaren λ-Term t kann man eine Anzahl n von Schritten bestimmen, innerhalb derer die Rightmost-Maxdepth-Strategie *bei Anwendung aus t terminiert.*

Beweis: Durch eine Doppelinduktion über d und m zeige man unter Verwendung von Lemma 2.4.14:

Für alle $d \in \mathbb{N}$ und alle $m \in \mathbb{N}$ gilt: wenn t ein λ-Term der Tiefe d ist und maximal m Redizes der Tiefe d enthält, dann gibt es eine Normalform t' von t und es gilt $t \xrightarrow{n} t'$ für ein $n \in \mathbb{N}$.

Die Ausführung des Induktionsbeweises sei dem Leser als Übung überlassen. □

Da wir nun eine Reduktionsstrategie kennen, die auf allen typsisierbaren λ-Termen terminiert, wissen wir auch, daß alle typisierbaren λ-Terme eine Normalform besizten müssen.

Korollar 2.4.17 (Schwache Normalisierbarkeit typisierbarer λ-Terme)

Jeder typisierbare λ-Term ist normalisierbar.

2.4.5.2 Starke Normalisierbarkeit

Wir haben soeben gezeigt, daß jeder typisierbare λ-Term eine terminierende Reduktionsfolge besitzt und daß es eine Strategie gibt, diese zu finden. Für einfach getypte λ-Terme kann man jedoch noch erheblich mehr zeigen, nämlich daß *jede* Reduktionsfolge terminieren muß. Diese Eigenschaft nennt man in der Sprache der Termersetzung *starke Normalisierbarkeit*.

Definition 2.4.18 (Starke Normalisierbarkeit)

Es sei $\xrightarrow{r}$ *eine Reduktionsrelation und* t *ein Term.*

1. t *heißt* stark normalisierbar *(SN), wenn jede mit* t *beginnende Reduktionsfolge endlich ist.*
2. $\xrightarrow{r}$ *ist stark normalisierbar, wenn jeder Term* t *unter* $\xrightarrow{r}$ *stark normalisierbar ist.*

Da jeder Term t nur an endlich vielen Stellen reduziert werden kann, bedeutet starke Normalisierbarkeit insbesondere, daß es für diesen Term eine Anzahl von Schritten gibt, innerhalb derer *alle* in t beginnenden Reduktionsfolgen terminieren. Es ist daher legitim, bei stark normalisierbaren Termen von der (maximalen) Anzahl der Reduktionsschritte zu sprechen. Eine Reduktionsrelation $\xrightarrow{r}$ ist stark normalisierbar, wenn jeder beliebige Term – unabhängig von der Reduktionsstrategie – nach endlich vielen Schritten in einer nicht weiter reduzierbaren Form ist.

Wir wollen nun zeigen, daß die β-Reduktion $\xrightarrow{\beta}$ auf typisierbaren λ-Termen genau diese Eigenschaft besitzt. Prinzipiell wäre es möglich, für die Theorie der einfachen Typen wie bei der schwachen Normalisierbarkeit ein kombinatorisches Argument zu konstruieren, also zu zeigen, daß irgendeine Größe durch *jede* Reduktion eines typisierbaren λ-Terms verringert wird. Wir werden im folgenden jedoch eine Beweismethode vorstellen, die sich auf aufwendigere Typsysteme, Reduktion höherer Ordnung und Polymorphie verallgemeinern läßt. Im Beweis werden wir daher Situationen betrachten, die im gegenwärtigen Kontext trivial erscheinen mögen, aber in allgemeineren Situationen von Bedeutung sein werden.

Die nach ihrem Erfinder benannte *TAIT computability method* [Tait, 1967] (mit technischen Verbesserungen von Girard [Girard, 1972]) ist im wesentlichen ein Induktionsbeweis, der in zwei Phasen vorgeht. Im ersten Schritt wird nachgewiesen, daß typisierbare λ-Terme eine wesentlich stärkere Eigenschaft besitzen als starke Normalisierbarkeit. Diese Eigenschaft, die als *Berechenbarkeit* bezeichnet wird, liefert erheblich stärkere Annahmen beim Führen des Induktionsschrittes und ist deshalb besser für eine Beweisführung geeignet. Im zweiten Schritt wird dann gezeigt, daß Berechenbarkeit tatsächlich stärker ist als starke Normalisierbarkeit.

Definition 2.4.19

Berechenbare λ*-Terme sind induktiv wie folgt definiert.*

- *Wenn* $t \in T$ *für einen atomaren Typ* T *gilt und* t *stark normalisierbar ist, dann ist* t *berechenbar.*
- *Gilt* $t \in S \to T$ *und ist* $t\,s$ *berechenbar für jeden berechenbaren Term* $s \in S$*, dann ist* t *berechenbar.*

Ein λ*-Term ist* neutral*, wenn er nicht die Gestalt* $\lambda x.t$ *hat.*

Der Unterschied zwischen Berechenbarkeit und starker Normalisierbarkeit liegt vor allem bei der Betrachtung von durch λ-Abstraktion definierten Funktionen, also den *kanonischen* Termen des λ-Kalküls. Während es bei starker Normalisierbarkeit ausreicht, daß alle Teilterme dieses Terms stark normalisierbar sind, verlangt die Berechenbarkeit, daß dies (d.h. Berechenbarkeit) auch dann gilt, wenn man für die gebundene Variable einen beliebigen berechenbaren Term einsetzt. Technisch ausgedrückt heißt dies, daß die Bildung eines Terms mit dem zugehörigen nichtkanonischen Operator (Applikation) zu einem berechenbaren Term führen muß. Diese Sprechweise deutet auch an, auf welche Art die obigen Begriffe auf andere Typkonstrukte verallgemeinert werden können.

Lemma 2.4.20

Es sei T ein beliebiger Typausdruck. t und t' seien beliebige λ-Terme vom Typ T. Dann gilt

1. *Wenn t berechenbar ist, dann ist t stark normalisierbar.*
2. *Wenn t berechenbar ist und $t \xrightarrow{\beta} t'$ gilt, dann ist t' berechenbar.*
3. *Wenn t neutral ist und jede β-Reduktion eines Redex in t zu einem berechenbaren Term t' führt, dann ist t berechenbar.*
4. *Wenn t neutral und in Normalform ist, dann ist t berechenbar.*

Beweis: Wir beweisen die Behauptungen durch eine simultane Induktion über die Struktur des Typs T.

Atomare Typen: Falls T atomar ist, so ist gemäß Definition 2.4.19 ein Term vom Typ T genau dann berechenbar, wenn er stark normalisierbar ist.

Es seien t und t' beliebige λ-Terme vom Typ T.

1. Wenn t berechenbar ist, dann ist t gemäß Definition 2.4.19 stark normalisierbar.
2. Wenn t berechenbar ist und $t \xrightarrow{\beta} t'$ gilt, dann sind alle Reduktionsfolgen von t' auch Teil einer in t beginnenden Reduktionsfolge. Da t auch stark normalisierbar ist, müssen all diese Reduktionsfolgen terminieren. Dies bedeutet, daß t' stark normalisierbar und damit auch berechenbar ist.
3. Wenn t neutral ist und jede β-Reduktion eines Redex in t zu einem berechenbaren Term t' führt, dann passiert jede in t beginnende Reduktionsfolge im ersten Schritt einen stark normalisierbaren Term t' vom Typ T. Dies bedeutet, daß diese Reduktionsfolge terminieren muß, woraus die starke Normalisierbarkeit von t, also auch die Berechenbarkeit folgt.
4. Die Behauptung folgt unmittelbar aus der dritten, da auf einen Term in Normalform keine β-Reduktion mehr angewandt werden kann.

Funktionenräume: Falls T die Gestalt $T_1 \to T_2$ hat, dann ist gemäß Definition 2.4.19 ein Term t vom Typ T genau dann berechenbar, wenn jede Applikation auf berechenbare Terme wieder einen berechenbaren Term liefert. Als Induktionsannahme setzen wir voraus, daß die Behauptungen 1. –4. für alle Terme des Typs T_1 bzw. T_2 bereits bewiesen sind.

Es seien t und t' beliebige λ-Terme vom Typ $T = T_1 \to T_2$.

1. Es sei t berechenbar und x eine Variable vom Typ T_1. Dann ist x neutral und in Normalform und somit gemäß Induktionsannahme 4. auch berechenbar. Damit ist nach Definition 2.4.19 auch $t\,x$ ein berechenbarer Term vom Typ T_2. Gemäß Induktionsannahme 1. ist $t\,x$ auch stark normalisierbar.
   Es sei nun $t \xrightarrow{\beta} t_1 \xrightarrow{\beta} t_2 \xrightarrow{\beta} \ldots$ eine beliebige in t beginnende Reduktionsfolge. Betrachten wir die hieraus entstehende Reduktionsfolge $t\,x \xrightarrow{\beta} t_1\,x \xrightarrow{\beta} t_2\,x \xrightarrow{\beta} \ldots$ für $t\,x$, so wissen wir, daß diese wegen der starken Normalisierbarkeit terminieren muß. Da diese Folge sich direkt durch eine zusätzliche Applikation aus der in t beginnenden Originalfolge ergibt, muß die Originalfolge ebenfalls terminieren. Damit terminiert jede in t beginnende Reduktionsfolge, d.h. t ist stark normalisierbar.
2. Es sei t berechenbar und es gelte $t \xrightarrow{\beta} t'$. Sei s ein beliebiger berechenbarer Term vom Typ T_1. Dann ist nach Definition der Berechenbarkeit $t\,s$ ein berechenbarer Term vom Typ T_2 und es gilt $t\,s \xrightarrow{\beta} t'\,s$. Nach Induktionsannahme 2. ist dann $t'\,s$ berechenbar. Nach Definition 2.4.19 ist damit t' berechenbar.
3. Es sei t neutral und jede β-Reduktion eines Redex in t führe zu einem berechenbaren Term t'. Sei s ein beliebiger berechenbarer Term vom Typ T_1. Dann ist s nach Induktionsannahme 1. auch stark normalisierbar. Um Induktionsannahme 3. verwenden zu können, zeigen wir nun durch Induktion über die Anzahl der Reduktionsschritte von s, daß jede β-Reduktion eines Redex in $t\,s$ zu einem berechenbaren Term t^* führt.
   - Falls s bereits in Normalform ist, dann hat t^* die Gestalt $t'\,s$ und es gilt $t \xrightarrow{\beta} t'$. Nach Voraussetzung ist t' berechenbar und damit auch $t'\,s \equiv t^*$.
   - Wir nehmen an, daß für alle in n Schritten normalisierenden berechenbaren Terme s' vom Typ T_1 gezeigt ist, daß jede β-Reduktion eines Redex in $t\,s'$ zu einem berechenbaren Term t^+ führt.
   - s reduziere in n+1 Schritten zur Normalform und es gelte $t\,s \xrightarrow{\beta} t^*$. Da t neutral ist, kann $t\,s$ selbst kein Redex sein. Es bleiben zwei Möglichkeiten:

(a) t^* hat die die Gestalt $t'\,s$ und es gilt $t \xrightarrow{\beta} t'$. Nach Voraussetzung ist t' berechenbar und damit auch $t'\,s \equiv t^*$.

(b) t^* hat die die Gestalt $t\,s'$ und es gilt $s \xrightarrow{\beta} s'$. Dann normalisiert s' in n Schritten und gemäß Induktionsannahme 2. ist s' berechenbar. Damit ist die innere Induktionsannahme anwendbar: jede β-Reduktion eines Redex in $t\,s'$ führt zu einem berechenbaren Term t^+.
Nun können wir die äußere Induktionsannahme 3. auf $t\,s'$, einen neutralen Term vom Typ T_2 anwenden und es folgt, daß $t\,s' \equiv t^*$ berechenbar ist.

In beiden Fällen ist also t^* berechenbar.

Damit ist gezeigt, daß jede β-Reduktion eines Redex in $t\,s$ zu einem berechenbaren Term t^* führt. Da $t\,s$ einen neutraler Term vom Typ T_2 ist, können wir Induktionsannahme 3. auf $t\,s$ anwenden und folgern, daß $t\,s$ berechenbar ist.
Da s beliebig gewählt war, sind die Voraussetzungen der Definition 2.4.19 erfüllt und t ist berechenbar.

4. Wie zuvor folgt die Behauptung unmittelbar aus der dritten, da auf einen Term in Normalform keine β-Reduktion mehr angewandt werden kann. □

In Definition 2.4.19 wurde die Berechenbarkeit von Funktionen über ihr Verhalten auf *berechenbaren* Argumenten definiert. Mit dem folgenden Lemma zeigen wir, daß wir von diesem Spezialfall abstrahieren können: anstelle von berechenbaren Argumenten dürfen auch beliebige Variablen des gleichen Typs verwendet werden. Der Vorteil hiervon ist, daß eine Variable *alle* Terme eines Typs darstellt und auch den Sonderfall handhaben kann, daß ein Typ überhaupt keine Elemente enthält. Die Verwendung von Variablen erlaubt also eine einheitliche Behandlung aller Typen.[78]

Lemma 2.4.21

Es sei t ein beliebiger typisierbarer λ-Term.
Wenn $t[s/x]$ berechenbar ist für alle berechenbaren λ-Terme s, dann ist $\lambda x.t$ berechenbar.

Beweis: Es sei t ein beliebiger λ-Term und $t[s/x]$ berechenbar für alle berechenbaren λ-Terme s. Um zu beweisen, daß $\lambda x.t$ berechenbar ist, müssen wir gemäß Definition 2.4.19 zeigen, daß $(\lambda x.t)\,s$ berechenbar ist für alle berechenbaren λ-Terme s.

Die Variable x ist neutral und in Normalform, also berechenbar nach Lemma 2.4.20.4. Damit muß $t{\equiv}t[x/x]$ nach Voraussetzung ebenfalls berechenbar sein. Es sei nun s ein berechenbarer Term vom Typ T_1. Dann sind nach Lemma 2.4.20.1. s und t stark normalisierbar.

Wir zeigen nun durch eine Doppelinduktion über die Anzahl der Reduktionsssschritte von t und s, daß $(\lambda x.t)\,s$ berechenbar ist. Hierbei werden wir Gebrauch machen von Lemma 2.4.20.3. Da $(\lambda x.t)\,s$ neutral ist, reicht es zu zeigen, daß jeder Term t^* berechenbar ist, für den gilt $(\lambda x.t)\,s \xrightarrow{\beta} t^*$.

Es gelte also $(\lambda x.t)\,s \xrightarrow{\beta} t^*$. Dann gibt es insgesamt 3 Möglichkeiten für t^*:

1. t^* hat die Gestalt $t[s/x]$. Dann ist t^* nach Voraussetzung berechenbar.
(*Dies deckt auch den Basisfall der Induktion ab*)

2. t^* hat die Gestalt $(\lambda x.t')\,s$, wobei $t \xrightarrow{\beta} t'$ gilt. Da t berechenbar ist, gilt nach Lemma 2.4.20.2., daß auch t' berechenbar ist und in weniger Schritten normalisiert als t. Wir verwenden nun die Induktionsannahme für t' und folgern hieraus, daß $(\lambda x.t')\,s \equiv t^*$ berechenbar ist.
(*Dies deckt den Schritt der äußeren Induktion für t zusammen mit dem Basisfall von s ab.*)

3. t^* hat die Gestalt $(\lambda x.t)\,s'$, wobei $s \xrightarrow{\beta} s'$ gilt. Da s berechenbar ist, gilt nach Lemma 2.4.20.2., daß auch s' berechenbar ist und in weniger Schritten normalisiert als s. Wir verwenden nun die Induktionsannahme für s' und folgern hieraus, daß $(\lambda x.t)\,s' \equiv t^*$ berechenbar ist.
(*Dies deckt den Schritt der inneren Induktion für s im Basisfall und im Induktionsschritt für t ab.*)

Damit sind die Voraussetzungen von Lemma 2.4.20.3. erfüllt und wir dürfen folgern, daß $(\lambda x.t)\,s$ berechenbar ist. Aus Definition 2.4.19 folgt nun die Berechenbarkeit von $\lambda x.t$. □

[78] Für die Theorie der einfachen Typen wäre dies – wie bereits erwähnt – sicherlich nicht notwendig. Die hier präsentierte Beweismethodik läßt sich aber leichter generalisieren, wenn wir sie bereits im Spezialfall möglichst universell formulieren.

Das folgende Lemma ist eine weitere Verstärkung des angestrebten Theorems. Es läßt sich leichter in einem Induktionsschritt verwenden und hat die gewünschte Aussage als Spezialfall.

Lemma 2.4.22

Es sei t ein beliebiger typisierbarer λ-Term. $x_1 \dots x_n$ seien die freien Variablen von t und $b_1 \dots b_n$ seien beliebige berechenbare λ-Terme, wobei b_i denselben Typ wie die Variable x_i in t hat.
Dann ist der Term $t[b_1 \dots b_n \,/\, x_1 \dots x_n]$ berechenbar.

Beweis: Wir führen einen Induktionsbeweis über die Struktur von $t{=}t[x_1 \dots x_n]$.

- Falls t eine Variable x_i ist, dann $t[b_1 \dots b_n \,/\, x_1 \dots x_n] = b_i$ und somit nach Voraussetzung berechenbar.
- Es sei die Behauptung gezeigt für typisierbare λ-Terme u und f.
- Es gibt zwei Möglichkeiten für die Struktur von t.
  - Falls t die Gestalt $\lambda x.u$ hat, dann ist $t[b_1 \dots b_n \,/\, x_1 \dots x_n] = \lambda x.u[b_1 \dots b_n \,/\, x_1 \dots x_n]$. (Die Variable x ist *nicht* frei in t). Nach Induktionsannahme ist für alle berechenbaren Terme b der Term $u[b_1 \dots b_n, b \,/\, x_1 \dots x_n, x]$ ein berechenbarer Term. Mit Lemma 2.4.21 folgt hieraus die Berechenbarkeit von $t[b_1 \dots b_n \,/\, x_1 \dots x_n]$.
  - Falls t die Gestalt $f\,u$ hat, dann ist $t[b_1 \dots b_n \,/\, x_1 \dots x_n] = f[b_1 \dots b_n \,/\, x_1 \dots x_n]\ \ u[b_1 \dots b_n \,/\, x_1 \dots x_n]$. Nach Induktionsannahme sind beide Teilterme einzeln berechenbar, woraus gemäß Definition 2.4.19 die Berechenbarkeit der Applikation $f[b_1 \dots b_n \,/\, x_1 \dots x_n]\ \ u[b_1 \dots b_n \,/\, x_1 \dots x_n]$. □

Nach all diesen – zugegebenermaßen recht aufwendigen – Vorarbeiten ist der Beweis der starken Normalisierbarkeit typisierbarer λ-Terme relativ einfach.

Satz 2.4.23 (Starke Normalisierbarkeit typisierbarer λ-Terme)

1. *Alle typisierbaren λ-Terme sind berechenbar.*
2. *Alle typisierbaren λ-Terme sind stark normalisierbar.*

Beweis: Es sei t ein beliebiger typisierbaren λ-Term.

1. Sind $x_1 \dots x_n$ die freien Variablen von t, so ist $t = t[x_1 \dots x_n \,/\, x_1 \dots x_n]$. Da alle x_i berechenbar sind (Lemma 2.4.20.4.), folgt mit Lemma 2.4.22 die Berechenbarkeit von t.
2. Nach Teil 1. ist t berechenbar. Mit Lemma 2.4.20.1. folgt hieraus, daß t auch stark normalisierbar ist. □

Die starke Normalisierbarkeit typisierbarer λ-Terme hat zur Folge, daß wir zur Auswertung typisierbarer Terme sehr effiziente Reduktionsstrategien einsetzen können, deren Verwendung im ungetypten λ-Kalkül Gefahr laufen würde, zu nichtterminierenden Reduktionsfolgen zu führen. Das folgende Beispiel zeigt, wie sich dieser Unterschied auswirkt.

Beispiel 2.4.24

Wir reduzieren den Term `trice(trice(λx.x))`, wobei `trice` $\equiv$ `λf.λx. f(f(f x))` (vgl. Beispiel 2.4.13) mit drei verschiedenen Strategien.

Rightmost (Call-by-value):

`trice(trice(λx.x))`
$\xrightarrow{\beta}$ `trice (λx. (λx.x) ( (λx.x) ( (λx.x) x)))`
$\xrightarrow{\beta}$ `trice (λx. (λx.x) ( (λx.x) x))`
$\xrightarrow{\beta}$ `trice (λx. (λx.x) x)`
$\xrightarrow{\beta}$ `trice (λx.x)`
$\xrightarrow{\beta}$ `λx. (λx.x) ( (λx.x) ( (λx.x) x))`
$\xrightarrow{\beta}$ `λx. (λx.x) ( (λx.x) x)`
$\xrightarrow{\beta}$ `λx. (λx.x) x`
$\xrightarrow{\beta}$ `λx.x`

Rightmost-maxdepth:

trice(trice(λx.x))
$\xrightarrow{\beta}$ trice (λx. (λx.x) ((λx.x) ((λx.x) x)))
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx. (λx.x) ((λx.x) ((λx.x) x))) x))
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx.x) ((λx.x) ((λx.x) x))))
$\xrightarrow{3}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) ((λx. (λx.x) ((λx.x) ((λx.x) x))) x)
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) ((λx.x) ((λx.x) ((λx.x) x)))
$\xrightarrow{3}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) x
$\xrightarrow{\beta}$ λx. (λx.x) ((λx.x) ((λx.x) x))
$\xrightarrow{3}$ λx.x

Leftmost (Call-by-name):

trice(trice(λx.x))
$\xrightarrow{\beta}$ λx. (trice (λx.x)) ((trice (λx.x)) ((trice (λx.x)) x))
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) ((trice (λx.x)) ((trice (λx.x)) x))
$\xrightarrow{\beta}$ λx. ((λx.x) ((λx.x) ((λx.x) (trice (λx.x))))) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. ((λx.x) ((λx.x) (trice (λx.x)))) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. ((λx.x) (trice (λx.x))) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. (trice (λx.x)) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. (trice (λx.x)) ((trice (λx.x)) x)
$\xrightarrow{6}$ λx. (trice (λx.x)) x
$\xrightarrow{6}$ λx. x

2.4.5.3 Konfluenz

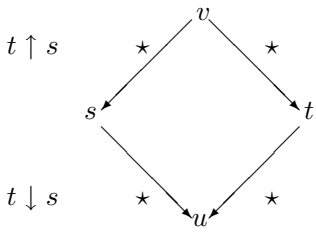
Für den ungetypten λ-Kalkül ist bekannt, daß jede terminierende Reduktionsfolge eines λ-Terms zu derselben Normalform führt. Der Beweis dieser Aussage (Das Church-Rosser Theorem auf Seite 64) ist wegen der Möglichkeit nichtterminierender Reduktionen allerdings ausgesprochen kompliziert. Für typisierbare λ-Terme ist wegen der starken Normalisierbarkeit der Beweis des Church-Rosser Theorem, den wir in diesem Abschnitt präsentieren, erheblich einfacher. Wir werden zu diesem Zwecke auf die Terminologie der Termersetzungssysteme zurückgreifen. In dieser Denkweise bedeutet die Eindeutigkeit der Normalform eines λ-Terms t, daß alle in t beginnenden Reduktionsfolgen wieder zu demselben Term zusammengeführt werden können, also *konfluent* sind. Um dies präzise zu definieren, führen wir einige Notationen ein.

Definition 2.4.25

Es seien t und s Terme und $\xrightarrow{r}$ eine Reduktionsrelation.

1. $\underline{t \xrightarrow{n} s}$ *gilt, falls s sich aus t durch n Reduktionsschritte ergibt.*
   - $t \xrightarrow{0} s$, *falls* $t{=}s$
   - $t \xrightarrow{n+1} s$, *falls es einen Term u gibt mit $t \xrightarrow{r} u$ und $u \xrightarrow{n} s$*
2. $\underline{t \xrightarrow{*} s}$ *gilt, falls s sich aus t durch endlich viele Reduktionen ergibt (d.h. $t \xrightarrow{n} s$ für ein $n \in \mathbb{N}$).*
3. $\underline{t \xrightarrow{+} s}$ *gilt, falls $t \xrightarrow{n} s$ für ein $n{>}0$.*
4. $\underline{t \uparrow s}$, *falls es einen Term u gibt mit $u \xrightarrow{*} t$ und $u \xrightarrow{*} s$*
5. $\underline{t \downarrow s}$, *falls es einen Term u gibt mit $t \xrightarrow{*} u$ und $s \xrightarrow{*} u$*

Diese Definitionen gelten für beliebige Reduktionsrelation, die – wie die β-Reduktion – nicht notwendigerweise transitiv sein müssen. $t{\uparrow}s$ bedeutet, daß t und s aus demselben Term u entstanden sind, während $t{\downarrow}s$ besagt, daß t und s zu demselben Term u zusammenfließen können.



Konfluenz bedeutet nun, daß das nebenstehende Diagramm kommutieren muß. Um diese Eigenschaft für die Typentheorie nachzuweisen, führen wir zusätzlich noch den Begriff der *lokalen Konfluenz* ein. Er besagt, daß t und s zu demselben Term u zusammenfließen können, wenn sie in einem Reduktionsschritt aus demselben Term entstanden sind.

Definition 2.4.26 (Konfluenz)

Es sei $\xrightarrow{r}$ *eine Reduktionsrelation.*

1. $\xrightarrow{r}$ *ist konfluent, wenn für alle Terme* t *und* s *gilt: aus* $t{\uparrow}s$ *folgt* $t{\downarrow}s$
2. $\xrightarrow{r}$ *ist lokal konfluent, wenn für alle Terme* t *und* s *gilt:*
   falls es einen Term u *gibt mit* $u \xrightarrow{r} t$ *und* $u \xrightarrow{r} s$, *dann folgt* $t{\downarrow}s$

Für stark normalisierbare Reduktionsrelationen kann man nun zeigen, daß lokale Konfluenz und Konfluenz identisch sind.

Lemma 2.4.27

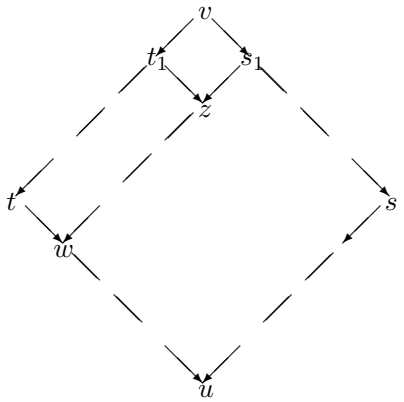
Es sei $\xrightarrow{r}$ *eine stark normalisierbare und lokal konfluente Reduktionsrelation.*
Dann ist $\xrightarrow{r}$ *auch konfluent.*

Beweis: Da $\xrightarrow{r}$ stark normalisierbar ist, gibt es für jeden Term v eine Anzahl n derart, daß jede in v beginnende Reduktionsfolge in maximal n Schritten terminiert. Wir zeigen durch Induktion über die Anzahl der Reduktionsschritte von v: *Gilt* $v \xrightarrow{*} t$ *und* $v \xrightarrow{*} s$, *dann folgt* $t{\downarrow}s$.

- Falls v in 0 Schritten reduziert, so folgt aus $v \xrightarrow{*} t$ und $v \xrightarrow{*} s$, daß $v{=}t{=}s$ sein muß. $t{\downarrow}s$ gilt somit trivialerweise.
- Es sei für alle Terme r, die in n Schritten reduzieren gezeigt, daß aus $r \xrightarrow{*} t$ und $r \xrightarrow{*} s$ folgt $t{\downarrow}s$.
- v reduziere in $n{+}1$ Schritten und es gelte $v \xrightarrow{*} t$ und $v \xrightarrow{*} s$.

  Falls $v \xrightarrow{0} t$ oder $v \xrightarrow{0} s$, so folgt $v{=}t$ oder $v{=}s$ und $t{\downarrow}s$ gilt trivialerweise.

  Andernfalls gibt es Terme t_1 und s_1 mit $v \xrightarrow{r} t_1 \xrightarrow{*} t$ und $v \xrightarrow{r} s_1 \xrightarrow{*} s$. Da v in $n{+}1$ Schritten reduziert, müssen t_1 und s_1 notwendigerweise in maximal n Schritten terminieren und die Induktionsannahme ist anwendbar.

  Wegen der lokalen Konfluenz gilt $t_1{\downarrow}s_1$, d.h. es gibt einen Term z mit $t_1 \xrightarrow{*} z$ und $s_1 \xrightarrow{*} z$.

  Aufgrund der Induktionsannahme für t_1 folgt aus $t_1 \xrightarrow{*} t$ und $t_1 \xrightarrow{*} z$, daß $t{\downarrow}z$ gilt. Es gibt also einen Term w mit $t \xrightarrow{*} w$ und $z \xrightarrow{*} w$.

  Für den Term w gilt $s_1 \xrightarrow{*} z \xrightarrow{*} w$. Da außerdem $s_1 \xrightarrow{*} s$ gilt, folgt mit der Induktionsannahme für s_1, daß $w{\downarrow}s$ gilt, d.h. daß es ein u gibt mit $w \xrightarrow{*} u$ und $s \xrightarrow{*} u$.

  Für diesen Term gilt auch $t \xrightarrow{*} w \xrightarrow{*} u$ und damit folgt $t{\downarrow}s$.

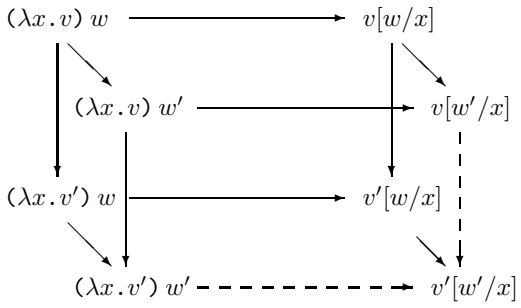
Damit ist die Induktionsbehauptung bewiesen: aus $t{\uparrow}s$ folgt immer $t{\downarrow}s$. □

Lemma 2.4.28

Die β-Reduktion $\xrightarrow{\beta}$ des λ-Kalküls ist lokal konfluent.

Beweis:

Es seien s, t zwei verschiedene λ-Terme und es gelte $u \xrightarrow{\beta} t$ und $u \xrightarrow{\beta} s$ für einen λ-Term u. Da s und t durch verschiedene β-Reduktionen aus u entstehen müssen, gibt es in u einen Teilterm der Gestalt $v\,w$ oder $(\lambda x\,.\,v)\,w$, so daß s durch Reduktion von v, w oder des äußeren β-Redex und t durch eine andere Reduktion entsteht.
Das nebenstehende Diagramm zeigt, wie für alle drei Möglichkeiten, einen Term der Form $(\lambda x\,.\,v)\,w$ zu reduzieren, die Reduktionskette wieder zusammengeführt werden kann. (Die Betrachtung von Termen der Gestalt $v\,w$ ist implizit darin enthalten.) Damit ist die β-Reduktion lokal konfluent.



□

Um das Church-Rosser Theorem für typisierbare λ-Terme zu beweisen, brauchen wir jetzt nur noch die beiden Lemmata 2.4.27 und 2.4.28 zusammenzusetzen.

Satz 2.4.29 (Church-Rosser Theorem der Typentheorie)

In der Typentheorie ist die β-Reduktion $\xrightarrow{\beta}$ konfluent.

Als ein Korollar erhalten wir die Eindeutigkeit der Normalform typisierbarer λ-Terme.

Korollar 2.4.30

Jeder typisierbare λ-Term hat eine eindeutig bestimmte Normalform.

2.4.5.4 Entscheidbarkeit

Aufgrund der starken Normalisierbarkeit und durch das Church-Rosser Theorem wissen wir, daß typisierbare λ-Terme sich sehr gut als Basis einer effizient zu verarbeitenden Programmiersprache eignen. Darüber hinaus folgt aber auch noch, daß die Typentheorie sehr gut geeignet ist für das automatisierte Schließen über das Verhalten von Programmen. Denn die Gleichheit zweier typisierbarer λ-Terme s und t läßt sich nun dadurch beweisen, daß man zunächst mit dem Hindley-Milner Algorithmus (Abbildung 2.10) ihr prinzipielles Typschema bestimmt, dann die beiden Terme normalisiert und dann gemäß Korollar 2.3.35 (Seite 65) auf Kongruenz überprüft.

Satz 2.4.31

Die Gleichheit typisierbarer λ-Terme ist entscheidbar.

Insgesamt wissen wir also, daß alle wesentlichen extensionalen Eigenschaften typisierbarer λ-Terme – im Gegensatz zu den untypisierten λ-Termen (vgl. Abschnitt 2.3.7) – entscheidbar sind.

- Das Halteproblem – die *Terminierung* von Algorithmen für ein gegebenes Argument – ist trivialerweise entscheidbar, da alle Funktionen terminieren.
- *Totalität* von typisierbaren λ-Funktionen ist aus dem gleichen Grunde entscheidbar.
- Die *Korrektheit* einer Funktion, also die Frage '$f\,u = t$' ist wegen Theorem 2.4.31 entscheidbar.
- Das gleiche gilt für die *Äquivalenz* von Programmen: '$f = g$'.
- Darüber hinaus lassen sich alle Eigenschaften eines Programms automatisch beweisen, die durch seinen Typ ausgedrückt werden können, da Typkorrektheit gemäß Theorem 2.4.10 ebenfalls entscheidbar ist.

Durch die Hinzunahme der Typdisziplin haben wir also die Möglichkeiten einer automatischen Überprüfung von Programmeigenschaften erheblich gesteigert.

2.4.6 Schließen über den Wert getypter λ-Terme

Bei unseren bisherigen Betrachtungen haben wir den λ-Kalkül und die Typdisziplin als separate Kalküle behandelt, um ihre Eigenschaften isoliert voneinander untersuchen zu können. Es ist jedoch nicht sehr schwer, den λ-Kalkül in die Typentheorie zu integrieren, also das Schließen über Gleichheit mit dem Schließen über Typzugehörigkeit zu verbinden. Man muß hierzu nur einen gemeinsamen Oberbegriff schaffen, nämlich das Schließen über die *Gleichheit zweier Terme innerhalb eines Typs*. Statt $s{=}t$ und $t{\in}T$ hätte man als Konklusion einer Sequenz also ein Urteil der Form

$$s{=}t \ \in \ T$$

welches besagt, daß s und t Elemente von T sind und innerhalb dieses Typs als gleich anzusehen sind.[79] Die Typzugehörigkeitsrelation $t{\in}T$ ergibt sich hieraus, indem man s und t identisch wählt. Man kann also $t{\in}T$ als definitorische Abkürzung für $t{=}t \ \in \ T$ ansehen. Dementsprechend kann man auch die Regeln des λ-Kalküls und die der Typentheorie vereinigen und erhält zum Beispiel die folgenden Regeln:

$$\begin{array}{l}\Gamma \vdash f\,t \ \texttt{=}\ g\,u \ \in \ T \quad \textbf{by}\ \texttt{applyEq}\ S\\ \qquad \Gamma \vdash f \ \texttt{=}\ g \ \in \ S{\to}T\\ \qquad \Gamma \vdash t \ \texttt{=}\ u \ \in \ S\end{array}$$

$$\begin{array}{l}\Gamma \vdash \lambda x.t \ \texttt{=}\ \lambda y.u \ \in \ S{\to}T \quad \textbf{by}\ \texttt{lambdaEq}\\ \qquad \Gamma,\ x'{:}S \vdash t[x'/x] \ \texttt{=}\ u[x'/y] \ \in \ T\\ \qquad \Gamma \vdash S \in \mathbb{U}\end{array}$$

$$\begin{array}{l}\Gamma \vdash (\lambda x.u)\ s \ \texttt{=}\ t \ \in \ T \quad \textbf{by}\ \texttt{reduction}\\ \qquad u[s/x] = t \ \in \ T\end{array}$$

Diese Integration von Typdisziplin und Berechnung in einem Kalkül werden wir im Kapitel 3 weiter vertiefen.

2.4.7 Die Curry-Howard Isomorphie

Bisher haben wir Typen im wesentlichen als Ausdrücke angesehen, welche die Bereiche kennzeichnen, zu denen ein durch einen λ-Term beschriebenes Objekt gehört. Wir haben soeben gezeigt, wie man λ-Kalkül und Typentheorie auf relativ einfache Art integrieren kann. Aber auch die Prädikatenlogik läßt sich im Prinzip mit der Typentheorie vereinigen. Vergleicht man nämlich die Regeln der Typentheorie mit denen der Prädikatenlogik aus Abbildung 2.6 (Seite 43), so entdeckt man eine gewisse Verwandschaft zwischen den Regeln für die Implikation und denen für die Einführung kanonischer und nichtkanonischer Terme in der Typentheorie. Diese Analogie wurde erstmalig von H. Curry [Curry *et.al.*, 1958], W. Tait [Tait, 1967] und W. Howard [Howard, 1980] beschrieben und wird in der Literatur als *Curry-Howard Isomorphie* bezeichnet.

Die Regel `lambda_i` der λ-Abstraktion besagt, daß ein λ-Term $\lambda x.t$ des Typs $S{\to}T$ dadurch aufgebaut werden kann, daß man im Beweis angibt, wie aus einer Variablen x vom Typ S ein Term t vom Typ T aufgebaut wird. Dies ist sehr ähnlich zu der Einführungsregel `imp_i` für die Implikation: um einen Beweis für $A \Rightarrow B$ aufzubauen, muß man zeigen wie aus der Annahme A ein Beweis für B aufgebaut wird.

$$\begin{array}{l}\Gamma \vdash \lambda x.t \in S{\to}T \quad \textbf{by}\ \texttt{lambda\_i}\\ \qquad \Gamma,\ x'{:}S \vdash t[x'/x] \in T\\ \qquad \Gamma \vdash S \in \mathbb{U}\end{array}$$

$$\begin{array}{l}\Gamma \vdash A \Rightarrow B \quad \textbf{by}\ \texttt{imp\_i}\\ \qquad \Gamma, A \vdash B\end{array}$$

Von der Struktur her kann `lambda_i` also als Verallgemeinerung der Regel `imp_i` angesehen werden. Die Typen S und T entsprechen den Formeln A und B und der Funktionenraumkonstruktor $\to$ der Implikation $\Rightarrow$. Allerdings verarbeitet die Typentheorie noch weitere Informationen, nämlich die Elemente der entsprechenden Typen und ihren strukturellen Aufbau. Dieser Zusammenhang gilt in ähnlicher Form auch für die Regel `apply_i` für die Applikation und die Eliminationsregel der Implikation, wenn man sie in ihrer ursprünglichen Form als Regel des *modus ponens* niederschreibt, welche auf der Konklusion statt auf den Hypothesen arbeitet.[80] Um B zu beweisen, reicht es aus, $A \Rightarrow B$ und A zu zeigen.

[79] Daß es sinnvoll sein kann, die Gleichheit vom Typ abhängig zu machen, zeigt das Beispiel der Darstellung von Rationalzahlen als Paare ganzer Zahlen. Verschiedene Paare können durchaus dieselbe rationale Zahl bezeichnen.

[80] Daß diese Form im Sequenzenkalkül nicht benutzt wird, hat strukturelle Gründe. Die Regel `imp_e` verwendet existierende Hypothesen, während `modus ponens` die Implikation als Konklusion eines Teilziels erzeugt und hierzu die Formel A benötigt.

$$\frac{\Gamma \vdash S \to T \quad \Gamma \vdash t \in S}{\Gamma \vdash f\,t \in T}\ \textbf{by}\ \texttt{apply\_i}\ S \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \textbf{by}\ \texttt{modus ponens}\ A$$

Wieder ist die typentheoretische Regel allgemeiner, da sie neben der Angabe, wie denn die Formeln (Typen) aufzubauen sind, auch noch sagt, wie sich die entsprechenden Terme zusammensetzen.

Der Zusammenhang zwischen Logik und Typentheorie wird noch deutlicher, wenn wir logische Aussagen nicht nur von ihrem Wahrheitsgehalt her betrachten, sondern uns auch noch für die Menge aller Beweise dieser Aussage interessieren. Wenn wir zum Beispiel bei der Verwendung von `imp_i` zeigen, wie wir aus der Annahme A die Formel B beweisen, dann geben wir genau genommen eine Konstruktion an, wie wir aus einem beliebigen Beweis a für A einen Beweis b für B erzeugen. Da hierbei a mehr oder weniger eine freie Variable in b ist, können wir einen Term der Art $\lambda a\,.\,b$ als Beweis für $A \Rightarrow B$ ansehen. Wenn wir umgekehrt bei der Verwendung von `modus ponens` einen Beweis pf_{AB} von $A \Rightarrow B$ und einen Beweis a von A kennen, dann wissen wir genau, wie wir A beweisen: wir wenden einfach den Beweis von $A \Rightarrow B$ auf den von A an, was wir kurz mit $pf_{AB}\ \ a$ bezeichnen können.

Zwischen der einfachen Typentheorie und dem Implikationsfragment der Logik (der Kernbestandteil des logischen Schließens) besteht also eine *Isomorphie*, wenn wir folgende Konzepte miteinander identifizieren.

Typ	Formel
Variable vom Typ S	Annahme der Formel S
Term vom Typ T (mit freien Variablen vom Typ S_i)	Beweis von T (unter den Annahmen S_i)
Typechecking	Beweisführung
(Regel der) λ-Abstraktion	$\Rightarrow$-Einführung
(Regel der) Applikation	$\Rightarrow$-Elimination

Diese Isomorphie zeigt, daß die Typentheorie nicht nur für das Schließen über Werte und Typzugehörigkeit von Programmen geeignet ist, sondern im Prinzip auch die Prädikatenlogik simulieren kann.[81] Damit bietet sie einen vielversprechenden Ansatz im Hinblick auf einen einheitlichen Formalismus für mathematisches Schließen und Programmierung.

2.4.8 Die Ausdruckskraft typisierbarer λ-Terme

Wir haben die Typdisziplin auf λ-Termen eingeführt, um den λ-Kalkül besser kontrollieren zu können, und bewußt in Kauf genommen, einen Teil der Ausdruckskraft des λ-Kalküls zu verlieren. Aus der theoretischen Informatik ist bekannt, daß jedes entscheidbare Berechenbarkeitskonzept, das – wie die Typentheorie – nur totale Funktionen zuläßt, dazu führt, daß manche berechenbare totale Funktion nicht mehr beschreibbar ist. Gewisse Einbußen müssen wir also hinnehmen. In diesem Abschnitt wollen wir nun untersuchen, wie viel Ausdruckskraft wir im typisierten λ-Kalkül behalten haben. Wie das folgende Beispiel zeigt, sind die elementarsten Funktionen auf natürlichen Zahlen leicht zu typisieren.

Beispiel 2.4.32

In Definition 2.3.17 haben wir natürliche Zahlen durch Church-Numerals beschrieben. Dabei war das Church-Numeral einer Zahl n definiert durch

$$\overline{n} \equiv \lambda \mathtt{f}\,.\,\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$$

Es läßt sich relativ leicht zeigen, daß jedes Church-Numeral den prinzipiellen Typ $(\mathtt{X} \to \mathtt{X}) \to \mathtt{X} \to \mathtt{X}$ besitzt (vergleiche Beispiel 2.4.11.1 auf Seite 77). Wir könnten daher den Datentyp IN als definitorische Abkürzung ansehen.

[81] So kann man zum Beispiel die Konjunktion $A \wedge B$ mit der Bildung eines Produktraumes $S \times T$ vergleichen: Ist a ein Beweis für A und b einer für B, so kann das Paar $\langle a, b\rangle$ als Beweis für $A \wedge B$ angesehen werden. Wieder entspräche die Einführungsregel der Regel für die kanonischen Elemente und die Eliminationsregel der Regel für die nichtkanonischen Elemente. Ähnlich sieht es aus mit den anderen logischen Operatoren. Da sich allerdings nicht alle hierzu nötigen Typkonstrukte in der einfachen Typentheorie simulieren lassen, werden wir hierauf erst im Kapitel 3 im Detail zurückkommen.

$$\mathbb{N} \equiv (\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}.$$

Mit dieser Abkürzung können wir überprüfen, daß gilt

$$\begin{array}{ll}
\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{n}\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x}) & \equiv \mathbf{s} \in \mathbb{N}\to\mathbb{N} \\
\lambda\mathtt{m}.\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{m}\ \mathtt{f}\ (\mathtt{n}\ \mathtt{f}\ \mathtt{x}) & \equiv \mathbf{add} \in \mathbb{N}\to\mathbb{N}\to\mathbb{N} \\
\lambda\mathtt{m}.\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{m}\ (\mathtt{n}\ \mathtt{f})\ \mathtt{x} & \equiv \mathbf{mul} \in \mathbb{N}\to\mathbb{N}\to\mathbb{N}
\end{array}$$

Auch die in Definition 2.3.15 angegebenen Erweiterungen für Paare können durch den Hindley-Milner Algorithmus ohne Probleme typisiert werden.

Beispiel 2.4.33

$$\begin{array}{ll}
\lambda\mathtt{p}.\ \mathtt{p}\,s\,t & \equiv \langle s,t\rangle \in (\mathtt{X}\to\mathtt{Y}\to\mathtt{Z}) \to \mathtt{Z} \\
pair\ (\lambda\mathtt{x}.\lambda\mathtt{y}.u) & \equiv \mathsf{let}\ \langle x,y\rangle = pair\ \mathsf{in}\ u \in \mathtt{Z}
\end{array}$$

Dabei ist $s\in\mathtt{X}$, $t\in\mathtt{Y}$, $pair \in (\mathtt{X}\to\mathtt{Y}\to\mathtt{Z})\to\mathtt{Z}$ und $u\in\mathtt{Z}$. Es erscheint damit legitim, den Produktraum $S\times T$ zweier Typen S und T wie folgt zu definieren:

$$S\times T \equiv (S\to T\to\mathtt{Z}) \to \mathtt{Z}$$

Auch die beiden Projektionen arbeiten in diesem Sinne korrekt. Bei Eingabe von $\langle s,t\rangle \in (S\to T\to\mathtt{Z}) \to \mathtt{Z}$ liefert die erste Projektion den Term $s\in S$ und die zweite den Term $t\in T$. Dies deckt sich mit der prinzipiellen Typisierung der beiden Funktionsanwendungen.

$$\begin{array}{ll}
\langle s,t\rangle\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x}) & \equiv \langle s,t\rangle.1 \in S \\
\langle s,t\rangle\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}) & \equiv \langle s,t\rangle.2 \in T
\end{array}$$

Man beachte, daß hierbei der Typ $\mathtt{Z}$ eine freie Variable ist, während die Typen S und T als die Typen von s und t bereits festliegen.

Typisierungen sind ohne weiteres auch möglich für boolesche Operatoren und Listen. Dennoch zeigen sich bei etwas näherem Hinsehen, daß diese Typisierung nicht ganz dem entspricht, was man gerne erreichen möchte.

Beispiel 2.4.34

Die Typisierung der booleschen Operatoren aus Definition 2.3.13 ergibt

$$\begin{array}{ll}
\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x} & \equiv \mathbf{T} \in \mathtt{X}\to\mathtt{Y}\to\mathtt{X} \\
\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y} & \equiv \mathbf{F} \in \mathtt{X}\to\mathtt{Y}\to\mathtt{Y} \\
b\,s\,t & \equiv \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \in \mathtt{Z}
\end{array}$$

Dabei ist im letzten Fall $b\in\mathtt{X}\to\mathtt{Y}\to\mathtt{Z}$, $s\in\mathtt{X}$ und $t\in\mathtt{Y}$. Wenn nun $\mathbf{T}$ und $\mathbf{F}$ zum gleichen Typ gehören sollen und als vorgesehene Eingaben im Conditional auftauchen sollen, so müssen $\mathtt{X}$, $\mathtt{Y}$ und $\mathtt{Z}$ identisch sein und wir bekommen

$$\mathbb{B} \equiv \mathtt{X}\to\mathtt{X}\to\mathtt{X}.$$

Dies würde aber verlangen, daß die Terme s und t innerhalb des Conditional immer vom gleichen Typ sein müssen.

Während die Typeinschränkungen bei den Termen des Conditionals noch akzeptabel erscheinen mögen (sie gelten für die meisten typisierten Sprachen), zeigt das folgende Beispiel, daß die bisherigen Typkonstrukte für praktische Zwecke nicht ganz ausreichen.

Beispiel 2.4.35

1. Die Exponentiierungsfunktion **exp** war in Definition 2.3.17 angegeben als

   $$\mathbf{exp} \equiv \lambda\mathtt{m}.\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{n}\ \mathtt{m}\ \mathtt{f}\ \mathtt{x}$$

   Der Prinzipielle Typ dieser Funktion ist

   $$(\mathtt{X}\to\mathtt{Y}\to\mathtt{Z}\to\mathtt{T}) \to \mathtt{X} \to \mathtt{Y} \to \mathtt{Z} \to \mathtt{T}$$

   wobei $\mathtt{m}\in\mathtt{X}\to\mathtt{Y}\to\mathtt{Z}\to\mathtt{T}$, $\mathtt{n}\in\mathtt{X}$, $\mathtt{f}\in\mathtt{Y}$ und $\mathtt{x}\in\mathtt{Z}$ ist. Das Dilemma entsteht nun dadurch, daß der Typ von $\mathtt{m}$ und $\mathtt{n}$ gleich – nämlich $\mathbb{N}$ – sein soll, was aber auf keinen Fall gewährleistet werden kann.

2. Das gleiche Problem entsteht bei der Typisierung der einfachen primitiven Rekursion.

   $\mathbf{PRs}[base,h] \equiv \lambda \mathtt{n}.\ \mathtt{n}\ h\ base$

   Der Prinzipielle Typ dieser Funktion ist

   $(\mathtt{X}\to\mathtt{Y}\to\mathtt{Z}) \to \mathtt{X} \to \mathtt{Y} \to \mathtt{Z}$

   Der Konzeption nach sollte *base* vom Typ $\mathbb{N}$ sein, h vom Typ $\mathbb{N}\to\mathbb{N}$, die Eingabe $\mathtt{n}$ vom Typ $\mathbb{N}$ und ebenso das Ergebnis. Setzt man dies ein, so erhält man als Typ für die einfache primitiven Rekursion

   $((\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}) \to (\mathbb{N}\to\mathbb{N}) \to \mathbb{N} \to \mathbb{N}$

   Dies bedeutet, daß die Eingabe $\mathtt{n}$ gleichzeitig den Typ $\mathbb{N}$ und $(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}$ haben muß. Dies aber ist nicht möglich, da die Gleichung

   $\mathbb{N} = (\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}$

   keine Lösung hat.

Das Hauptproblem im letzten Beispiel ist die uneingeschränkte polymorphe Verwendung von $\mathtt{n}$ als Element von $\mathbb{N} \equiv (\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$ für *verschiedene* Instanzen der freien Variablen $\mathtt{X}$ innerhalb *derselben* Anwendung. Es ist daher nicht möglich, die Variable $\mathtt{X}$ mit einem einheitlichen Wert zu belegen. Stattdessen müßte für jede Zahl n dem Church-Numeral $\overline{n}$ eine andere Instanz von $(\mathtt{X}\to\mathtt{X})\to\mathtt{X}\to\mathtt{X}$ zugewiesen werden. Damit ist es nicht möglich, die primitive Rekursion ihrem Sinn gemäß zu typisieren.

Dies bedeutet, daß die Ausdruckskraft der einfachen Typentheorie unter derjenigen der primitiv rekursiven Funktionen liegt, was für praktische Anwendungen völlig unakzeptabel ist.[82] Somit ist die *einfache* Typentheorie als Kalkül zum Schließen über Programme ebenso ungeeignet wie der ungetypte λ-Kalkül. Dies liegt aber nicht so sehr am Typkonzept als solches, sondern daran, daß das bisher betrachtete Typkonstrukt – die simple Funktionenraumbildung – nicht ausdrucksstark genug ist. Im Anbetracht der vielen guten Eigenschaften, die wir von der einfachen Typentheorie nachweisen konnten, lohnt es sich, nach einer Erweiterung des Typsystems zu suchen, welche aus praktischer Hinsicht zufriedenstellend ist.

2.5 Diskussion

Wir haben in diesem Kapitel formale Kalküle zum Schließen über Logik, Berechnung und Typzugehörigkeit vorgestellt und ihre Eigenschaften untersucht. Mit der *Prädikatenlogik* können wir die *Struktur* logischer Aussagen analysieren und diejenigen unter ihnen beweisen, deren Wahrheitsgehalt ausschließlich aus dieser Struktur folgt. Mit dem *λ-Kalkül* können wir den *Wert* eines durch einen Ausdruck beschriebenen Objektes bestimmen und entsprechend über die Gleichheit von Werten Schlüsse ziehen. Die *einfache Typentheorie* stellt die Beziehung zwischen einem Programm (λ-Term) und seinen Eigenschaften (Typ) her und wird darüber hinaus dazu benutzt, um durch Typrestriktionen diejenigen Probleme zu umgehen, welche sich aus der Mächtigkeit des λ-Kalküls ergeben. Ihre bisherige Formulierung ist jedoch noch zu einfach und muß zugunsten einer verbesserten Ausdruckskraft um weitere Konzepte erweitert werden.

Jeder einzelne der drei Kalküle ist für ein einzelnes Teilgebiet des formalen Schließens über Programme sehr gut geeignet. In allen erwies sich der Stil des Sequenzenkalküls als geeignetes Mittel, die semantischen Zusammenhänge durch formale Regeln auszudrücken. Prinzipiell wäre es nun möglich, einen einheitlichen Kalkül für das Schließen über Programme durch eine simple Vereinigung aller Regeln der drei Kalküle zu erreichen. Dies würde jedoch die Schnittstelle *zwischen* den Kalkülen völlig ungeklärt belassen. Eine sehr interessante Möglichkeit, einen integrierten und leistungsfähigen Kalkül zum einheitlichen Schließen über alle Aspekte der Mathematik und Programmierung aufzubauen, ergibt sich aus den am Ende des letzten Abschnitts andiskutierten Ideen.

[82] Dies betrifft weniger die Klasse der Funktionen, die man im einfach typisierten λ-Kalkül ausdrücken kann, als die Programmiertechniken, die man einsetzen darf. Schon die einfachste Form der Schleife ist nicht erlaubt.

- Durch die Einführung einer *typisierten Gleichheit* (siehe Abschnitt 2.4.6), die ohnehin einem natürlichen Verständnis der Gleichheit näher kommt als die ungetypte Gleichheit, können wir das Schließen über Berechnung (d.h. den λ-Kalkül) in die Typentheorie *integrieren*.
- Die in Abschnitt 2.4.7 angesprochene Curry-Howard Isomorphie zwischen Logik und Typkonstrukten ermöglicht es, die Prädikatenlogik innerhalb der Typentheorie zu *simulieren*.

Eine entsprechend ausdrucksstarke Typentheorie wird also für das formal-mathematische Schließen und die Programmierung eine ähnlich grundlegende Rolle bekommen wie die Mengentheorie für die abstrakte Mathematik. Um dies zu erreichen, müssen wir allerdings die einfache Typentheorie gehörig ausdehnen.

Dabei können wir zwar die grundlegenden *Ideen* der einfachen Typentheorie beibehalten, müssen uns aber von dem Gedanken lösen, die Typentheorie als eine reine Ergänzung zum einfachen λ-Kalkül zu betrachten. So ist zum Beispiel eine naive Typisierung von natürlichen Zahlen auf der Basis der Church Numerals (vergleiche Abschnitt 2.4.8) problematisch. Die Diskussion zu Beginn des folgenden Kapitels wird deutlich machen, daß wir parallel mit der Erweiterung des Typsystems auch den λ-Kalkül ausdehnen müssen.

Die Entwicklung einer formalen Theorie, mit der wir alle Aspekte der Programmierung formalisieren können, ist also in einem gewissen Sinne ein inkrementeller Prozeß. Der ungetypte λ-Kalkül war konzeptionell der einfachste Programmierkalkül, aber zu mächtig, um ein automatisiertes Schließen zu ermöglichen. Die einfache Typentheorie war wiederum der einfachste Weg, den λ-Kalkül auf syntaktischem Wege einzugrenzen, aber ihre Restriktionen waren zu stark, um noch genügend Ausdruckskraft übrig zu lassen. Die Lösung liegt, wie immer, in der Mitte. Da hierfür aber genauere Abgrenzungen nötig sind, wird der integrierte Kalkül zum Schließen über Programme und ihre Eigenschaften insgesamt komplexer werden müssen als die bisher besprochenen Einzelkalküle.

2.6 Ergänzende Literatur

Gute Einführungen in elementare Logik für Informatiker findet man in Lehrbüchern wie [Boyer & Moore, 1979, Gallier, 1986, Manna & Waldinger, 1985, Turner, 1984]. Leser, die an stärker mathematischen Zugängen zu logischen Kalkülen interessiert sind, sollten Bücher wie [Richter, 1978, Schütte, 1977, Takeuti, 1975] konsultieren, in denen viele Aspekte gründlich ausgearbeitet sind. Lesenswert sind auch die Einführungskapitel von [Andrews, 1986, Bibel, 1987, Girard *et.al.*, 1989, Lakatos, 1976, Paulson, 1987], in denen viele Beispiele zu finden sind. Das Buch von Prawitz [Prawitz, 1965] kann als Standardreferenz für natürliche Deduktion angesehen werden. In [Dummett, 1977] findet man eine gute Beschreibung von Philosophie, Semantik und Inferenzsystemen der intuitionistischen Logik. Lesenswert sind auch [Curry, 1970, Heyting, 1971, van Dalen, 1986, Nerode, 1988].

Eine gute Einführung in den λ-Kalkül wurde von Hindley und Seldin [Hindley & Seldin, 1986] und von Huet [Huet, 1986] geschrieben. Das Buch von Barendregt [Barendregt, 1981] dagegen ist sehr ausführlich und detailliert. Wertvolle Details findet man auch im Buch von Stenlund [Stenlund, 1972].

Einführungen in die einfache Typentheorie findet man meist im Zusammenhang mit der Abhandlung komplexerer Theorien wie zum Beispiel in [Martin-Löf, 1984, Constable *et.al.*, 1986, Nordström *et.al.*, 1990, Backhouse *et.al.*, 1988a, Backhouse *et.al.*, 1988b]. Viele der hier vorgestellten Beweise findet man auch in den Büchern von Stenlund [Stenlund, 1972] und Girard [Girard *et.al.*, 1989].

Kapitel 3

Die Intuitionistische Typentheorie

Im vorhergehenden Kapitel haben wir die verschiedenen Arten des logischen Schließens besprochen, die beim Beweis mathematischer Aussagen und bei der Entwicklung korrekter Programme eine Rolle spielen. Wir haben die Prädikatenlogik, den λ-Kalkül und die einfache Typentheorie separat voneinander untersucht und festgestellt, daß jeder dieser drei Kalküle für jeweils ein Teilgebiet – logische Struktur, Werte von Ausdrücken und Eigenschaften (Typen) von Programmen – gut geeignet ist. Um nun *alle* Aspekte des Schließens über Programmierung innerhalb eines einzigen universellen Kalküls behandeln zu können, ist es nötig, diese Kalküle zu vereinigen und darüber hinaus das etwas zu schwache Typsystem der einfachen Typentheorie weiter auszubauen.

Möglichkeiten zur Integration der Logik und des λ-Kalküls in die Typentheorie haben wir im Abschnitt 2.4 bereits andiskutiert. Aufgrund der Curry-Howard Isomorphie läßt sich die Logik durch Typkonstrukte simulieren während der λ-Kalkül in einer Erweiterung des Typkonzepts zu einer typisierten Gleichheit aufgeht. Eine ausdrucksstarke Typentheorie ist somit in der Lage, alle drei bisher vorgestellten Konzepte zu subsumieren.

In der Literatur gibt es eine ganze Reihe von Ansätzen, eine solche Theorie zu entwickeln,[1] von denen bekannt ist, daß sie prinzipiell in der Lage sind, alle Aspekte von Mathematik und Programmierung zu formalisieren. Einen optimalen Kalkül gibt es hierbei jedoch nicht, da die Komplexität der Thematik es leider nicht erlaubt, einen Kalkül anzugeben, der zugleich einfach, elegant und leicht zu verwenden ist. So muß man bei der Entscheidung für die Vorteile eines bestimmten Kalkül immer gewisse Nachteile in Kauf nehmen, welche andere Kalküle nicht besitzen. In diesem Kapitel werden wir die *intuitionistische Typentheorie* des NuPRL Systems [Constable *et.al.*, 1986] vorstellen, die auf entsprechende Vorarbeiten von Per Martin-Löf [Martin-Löf, 1982, Martin-Löf, 1984] zurückgeht und auf eine Verarbeitung mit einem interaktiven Beweissystem angepaßt ist. Aus praktischer Hinsicht erscheint diese Theorie derzeit die geeigneteste zu sein, auch wenn man an dem zugehörigen System noch beliebig viel verbessern kann.

Da die intuitionistische Typentheorie gegenüber der einfachen Typentheorie eine erhebliche Erweiterung darstellt, werden wir zunächst in Abschnitt 3.1 die erforderlichen und wünschenswerten Aspekte diskutieren, die eine Theorie erfüllen sollte, um für das logische Schließen über alle relevanten Aspekte von Mathematik und Programmierung verwendbar zu sein. Aufgrund der Vielfalt der zu integrierenden (Typ-)konzepte werden wir im Anschluß daran (Abschnitt 3.2) eine Systematik für einen einheitlichen Aufbau der Theorie entwickeln und am Beispiel der grundlegendensten Typkonzepte illustrieren. Zugunsten einer in sich geschlossenen und vollständigen Präsentation werden wir dabei die gesamte Begriffswelt (Terme, Berechnung, Semantik, Beweise etc.) komplett neu definieren.[2] Wir müssen dabei etwas tiefer gehen als zuvor, um den gemeinsamen Nenner der bisherigen Kalküle uniform beschreiben zu können.

[1] Die wichtigsten Vertreter findet man beschrieben in
[Girard, 1971, Bruijn, 1980, Martin-Löf, 1982, Martin-Löf, 1984, Constable *et.al.*, 1986, Girard, 1986, Girard *et.al.*, 1989, Coquand & Huet, 1988, Paulin-Mohring, 1989, Constable & Howe, 1990a, Nordström *et.al.*, 1990]

[2] Die Konzepte des vorhergehenden Kapitels sind Spezialfälle davon, für die innerhalb des NuPRL Systems einige Details unterdrückt werden mußten.

Aufbauend auf dieser Systematik werden wir dann die Typentheorie von NuPRL in drei Phasen vorstellen. Zunächst betten wir über die Curry-Howard Isomorphie die Prädikatenlogik in die Typentheorie ein (Abschnitt 3.3) und zeigen, daß sich viele wichtige Eigenschaften der Prädikatenlogik unmittelbar aus grundlegenden Eigenschaften typisierbarer Terme ergeben. In Abschnitt 3.4 ergänzen wir die bis dahin besprochenen Typkonstrukte um solche, die für realistische Programme benötigt werden, und besprechen, wie die Entwicklung von korrekten Programmen aus einer gegebenen Spezifikation mit dem Konzept der Beweisführung zusammenhängt. Abschnitt 3.5 widmet sich der Frage, wie die Rekursion, die durch die Einführung des Typkonzeptes zunächst aus der Theorie verbannt wurde, unter Erhaltung der guten Eigenschaften wieder in die Typentheorie integriert werden kann. Im Abschnitt 3.6 stellen wir schließlich einige Ergänzungen des Inferenzsystems vor, die aus theoretischer Sicht zwar nicht erforderlich sind, das praktische Arbeiten mit dem Kalkül jedoch deutlich vereinfachen.

3.1 Anforderungen an einen universell verwendbaren Kalkül

Die Typentheorie erhebt für sich den Anspruch, alle in der Praxis relevanten Aspekte von Mathematik und Programmierung formalisieren zu können. Damit wird ihr die gleiche grundlegende Rolle für konstruktive formale Systeme beigemessen wie der Mengentheorie für die abstrakte Mathematik. Dies bedeutet insbesondere, daß jeder formale Ausdruck eine feste Bedeutung hat, die keiner weiteren Interpretation durch eine andere Theorie bedarf.

Wir wollen nun die Anforderungen diskutieren, die man an einen universell verwendbaren Kalkül stellen sollte, damit dieser seinem Anspruch einigermaßen gerecht werden kann. Diese Anforderungen betreffen die Semantik einer derart grundlegenden Theorie, ihre Ausdruckskraft sowie die Eigenschaften von Berechnungsmechanismen und Beweiskalkül.

3.1.1 Typentheorie als konstruktive mathematische Grundlagentheorie

Da der Typentheorie eine ähnlich fundamentale Rolle beigemessen werden soll wie der Mengentheorie, muß sie eine universelle Sprache bereitstellen, in der alle anderen Konzepte formuliert werden können. Die Bedeutung und Eigenschaften dieser Konzepte müssen sich dann sich ohne Hinzunahme weiterer Informationen aus den Gesetzen der Typentheorie ableiten lassen. Ihre eigenen Gesetze dagegen können nicht durch eine andere Theorie erklärt werden sondern müssen sich aus einem intuitiven Verständnis der verwendeten Begriffe ergeben. Natürlich müssen diese Gesetze konsistent sein, also einander nicht widersprechen, aber eine tiefergehende Abstützung kann es nicht geben.

Diese Sichtweise auf die Typentheorie erscheint auf den ersten Blick sehr ungewöhnlich zu sein, da man üblicherweise allen mathematischen Theorien eine Semantik zuordnen möchte, die auf mengentheoretischen Konzepten abgestützt ist. Die Mengentheorie selbst jedoch wird nicht weiter hinterfragt und besitzt genau die oben genannten Grundeigenschaften. Als eine grundlegende Theorie *kann* sie nur noch auf der Intuition abgestützt werden.[3] Sie ist der Ausgangspunkt für einen systematischen Aufbau der Mathematik und das einzige, was man fragen muß, ist, ob dieser Ausgangspunkt tatsächlich mächtig genug ist, alle mathematischen Konzepte zu beschreiben.

Die *intuitionistische Typentheorie* nimmt für sich nun genau die gleiche Rolle in Anspruch und man mag sich fragen, wozu man eine neue und kompliziertere Theorie benötigt, wenn doch die Mengentheorie mit dem Konzept der Mengenbildung und die Element-Beziehung "$x \in M$" bereits auskommt. Der Grund hierfür ist – wie wir es schon bei der Prädikatenlogik diskutiert hatten – eine unterschiedliche Sichtweise auf die Mathematik. Während die (klassische) Mengentheorie hochgradig unkonstruktiv ist und ohne die Hinzunahme

[3] Ansonsten müsste es ja eine andere Theorie geben, die noch fundamentaler ist, und wir stünden bei dieser Theorie wieder vor derselben Frage. An irgendeinem Punkt muß man nun einmal anfangen.

semantisch schwer zu erklärender Berechenbarkeitsmodelle nicht in der Lage ist, den Begriff der Konstruktion bzw. des Algorithmus zu erfassen, versteht sich die Typentheorie als eine *konstruktive Mengentheorie*. Ausgehend von dem Gedanken, daß Konstruktionen den Kern eines mathematischen Gedankens bilden, muß die Konstruktion natürlich auch den Kernbestandteil einer Formalisierung der mathematischen Grundlagen bilden, was in der klassischen Mengentheorie nicht der Fall ist. Dieses Konzept herunterzubrechen auf die Ebene der Mengenbildung würde es seiner fundamentalen Rolle berauben.[4] So gibt es in der Typentheorie zwar die gleichen Grundkonzepte wie in der Mengentheorie, also Mengen (Typen) und Elemente, aber bei der Erklärung, wie Mengen zu bilden sind, welche Elemente zu welchen Mengen gehören und was eine Menge überhaupt ausmacht, wird die Konstruktion und ihre formale Beschreibung eine zentrale Rolle spielen.

Wie jeder formale Kalkül besteht die Typentheorie aus einer formalen Sprache, deren *Syntax* und *Semantik* wir zu erklären haben, und einer Menge von *Inferenzregeln*, welche die Semantik der formalen Sprache wiederspiegeln sollen. Wegen ihres konstruktiven Aspekts müssen wir davon ausgehen, daß die formale Sprache der Typentheorie erheblich umfangreicher sein wird als die der Mengentheorie, zumal wir elementare Konstruktionen nicht weiter zerlegen werden. Zwar wäre es technisch durchaus möglich, Konzepte wie Zahlen weiter herunterzubrechen – z.B. durch eine Simulation durch Church-Numerals – aber dies wäre ausgesprochen unnatürlich, da Zahlen nun einmal ein elementares Grundkonzept sind und nicht umsonst "natürliche" Zahlen genannt werden. Andererseits sollte eine universell verwendbare Theorie auch einen *einheitlichen Aufbau* haben. Um dies sicherzustellen, benötigt die Theorie zusätzlich eine erkennbare *Systematik*, nach der formale Ausdrücke aufgebaut sind, wie ihre Semantik – also ihr Wert und der Zusammenhang zu anderen Ausdrücken – bestimmt werden kann und wie das Inferenzsystem aufzubauen ist. Diese Systematik werden wir in Abschnitt 3.2 entwickeln.

Die Entwicklung einer Grundlagentheorie, welche die gesamte intuitive – das steckt ja hinter dem Wort "intuitionistisch" – Mathematik und Programmierung beschreiben kann, ist eine komplizierte Angelegenheit. Der λ-Kalkül scheint mit seiner Turing-Mächtigkeit zu weit zu gehen, da er auch contra-intuitive Konstruktionen wie eine beliebige Selbstanwendbarkeit zuläßt. Eine Grundlagentheorie muß beschreiben können, was *sinnvoll* ist, und muß sich nicht auf alles einlassen, was *machbar* ist. Es ist natürlich die Frage, ob eine solche Theorie überhaupt formalisierbar ist. Die Erfahrungen zeigen, daß man mit einer einfache Formalisierung entweder weit über das Ziel hinausschießt oder wesentliche Aspekte nicht beschreiben kann. Aus diesem Grunde muß die Theorie *offen* (*open-ended*) gestaltet werden, also spätere Erweiterungen durch Hinzunahme neuer Konstrukte zulassen, die sich nicht simulieren lassen, aber essentielle mathematische Denkweisen wiederspiegeln.[5]

3.1.2 Ausdruckskraft

Die Anforderungen an die Ausdruckskraft der Typentheorie ergeben sich unmittelbar aus dem Anspruch, als grundlegende Theorie für die Formalisierung von Mathematik und Programmierung verwendbar zu sein. Dies verlangt, daß alle elementaren Konzepte von Mathematik und Programmiersprachen ein natürliches Gegenstück innerhalb der Typentheorie benötigen – entweder als fest vordefinierten Bestandteil der Theorie oder als leicht durchzuführende definitorische Erweiterung.

Natürlich wäre es wünschenswert, wirklich alle praktisch relevanten Konzepte direkt in die Theorie einzubetten. Dies aber ist unrealistisch, da hierdurch die Theorie extrem umfangreich würde und der Nachweis wichtiger Eigenschaften somit kaum noch zu führen wäre. So ist es sinnvoll, sich darauf zu beschränken, was für die Beschreibung der Struktur von Mengen grundlegend ist bzw. was in Programmiersprachen als zentraler (vorzudefinierender) Bestandteil anzusehen ist. Da die Theorie durch Definitionen jederzeit konservativ erweitert werden kann, ist dies keine essentielle Einschränkung.

[4] Der Wunsch, die gesamte Mathematik durch nur zwei oder drei Konstrukte zu beschreiben, ist zwar verständlich, aber es erhebt sich die Frage, ob das mathematische Denken im Endeffekt so einfach ist, daß es sich durch so wenige Konstrukte ausdrücken läßt. Dies zu glauben oder abzulehnen ist wiederum eine Frage der Weltanschauung. Eine absolute Wahrheit kann es hier nicht geben.

[5] Eine solche Denkweise ist zum Beispiel die *Reflektion*, also die Fähigkeit, über sich selbst etwas auszusagen, oder die *Analogie*.

Zu den grundlegenden Konstrukten gehören sicherlich alle im vorhergehenden Kapitel besprochenen Konzepte, also die Bestandteile der Prädikatenlogik, des λ-Kalküls und der einfachen Typentheorie. Dabei bilden Funktionsdefinition und -anwendung des λ-Kalküls und der Konstruktor $\rightarrow$ der einfachen Typentheorie die Kernbestandteile des Datentyps "Funktionenraum". Da wir uns bereits darauf festgelegt haben, die Logik dadurch zu integrieren, daß wir eine Formel mit dem Datentyp all ihrer Beweise (aufgeschrieben als Terme) identifizieren, müssen wir logische Operatoren durch entsprechende Typkonstruktoren simulieren und entsprechend durch formale Definitionen repräsentieren. Dabei haben wir bereits festgestellt, daß die Implikation sich genauso verhält wie der Operator $\rightarrow$ und angedeutet, daß die Konjunktion mit der Produktbildung verwandt ist. Weitere Zusammenhänge zu den im folgenden als grundlegend vorgestellten Typkonzepten werden wir in Abschnitt 3.3 ausführlich diskutieren. Wir können uns daher darauf beschränken, grundlegende Typkonstrukte zusammenzustellen – also Operatoren zur Konstruktion von Mengen, die zugehörigen kanonischen Elemente und die zulässigen Operationen auf den Elementen. Diese sind

- Der *Funktionen*raum $S \rightarrow T$ zweier Mengen S und T mit kanonischen Elementen der Form $\lambda x\,.\,t$ und Applikation $f\,t$ als zulässiger Operation.

- Der *Produkt*raum $S \times T$ zweier Mengen S und T mit kanonischen Elementen der Form $\langle s\,,t\rangle$ und den Projektionen bzw. allgemeiner dem spread-Operator $\mathsf{let}\ \langle x\,,y\rangle = pair\ \mathsf{in}\ u$ als zulässiger Operation.

  Die in der Programmierung benutzten Datentypen *Feld* (array) und *Verbund* (record) sind durch mehrfache Produktbildung leicht zu simulieren, wobei für Felder auch der Funktionenraum `[1..n]`$\rightarrow T$ benutzt werden kann.

- Die *Summe* (disjunkte Vereinigung) $S+T$ zweier Mengen S und T. Diese unterscheidet sich von der aus der Mengentheorie bekannten Vereinigung der Mengen S und T dadurch, daß man den Elementen von $S+T$ ansehen kann, ob sie nun aus S oder aus T stammen.[6] Kanonische Elemente bestehen also aus den Elementen von S und denen von T zusammen mit einer Kennzeichnung des Ursprungs, wofür sich als Notation $\mathsf{inl}(s)$ bzw. $\mathsf{inr}(t)$ (linke bzw. rechte Injektion) eingebürgert hat. Die einzig sinnvolle Operation auf diesem Typ besteht darin, eine Eingabe e zu analysieren und ihren Ursprung sowie das in der Injektion eingekapselte Element zu bestimmen. Abhängig von diesem Resultat können dann verschiedene Terme zurückgegeben werden. Als Notation hierfür kann man zum Beispiel die folgende, an Programmiersprachen angelehnte Bezeichnungsweise $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v$ verwenden.

  Der Datentyp $\mathbb{B}$ kann als ein Spezialfall hiervon angesehen werden, nämlich als Summe von Mengen, auf deren Elemente es nicht ankommt. T entspricht dann der linken und F der rechten Injektion. Das Conditional $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$ analysiert den Ursprung eines booleschen Wertes b (aber nicht mehr das eingekapselte Element) und liefert dementsprechend einen der beiden Terme s und t als Ergebnis.

- *Zahlen* – zumindest natürliche ($\mathbb{N}$) oder ganze Zahlen ($\mathbb{Z}$) – bilden die Grundlage jeglicher Programmierung und müssen in der Theorie explizit enthalten sein. Kanonische Elemente sind `0,1,-1,2,-2,...` während die zulässigen Operationen aus den gängigen Funktionen `+,-,*,/` etc. und der Möglichkeit einer induktiven Verarbeitung von Zahlen bestehen müssen. Eine Simulation von Zahlen durch andere Konstruktionen ist aus praktischen Gründen abzulehnen.

- Für die Bildung endlicher, aber beliebig großer Strukturen sollte der Raum S `list` der *Listen* (Folgen) über einer gegebenen Menge S und seine Operatoren (vergleiche Abschnitt 2.3.3, Seite 56) ebenfalls ein Grundbestandteil der Theorie sein. Auch hier wäre eine Simulation möglich, aber unpraktisch.

- Die Bildung von *Teilmengen* der Form $\{x{:}T \mid P(x)\}$ ist erforderlich, wenn ein Typ als Spezialfall eines anderen repräsentiert werden soll, wie zum Beispiel die natürlichen Zahlen als Spezialfall der ganzen Zahlen. An kanonischen Elementen und zulässigen Operatoren sollte sich hierdurch nichts ändern (die hierdurch entstehenden Schwierigkeiten diskutieren wir in Abschnitt 3.4.4).

[6] Eine beliebige Vereinigung zweier Mengen wie in der Mengentheorie ist hochgradig nichtkonstruktiv, da die beiden Mengen einfach zusammengeworfen werden. Es besteht somit keine Möglichkeit, die entstandene Menge weiter zu analysieren.

- Ebenso mag es notwendig sein, die Gleichheit innerhalb eines Typs durch Bildung von Restklassen umzudefinieren. Mit derartigen *Quotiententypen* (siehe Abschnitt 3.4.5) kann man zum Beispiel rationale Zahlen auf der Basis von Paaren ganzer Zahlen definieren.

- Über die Notwendigkeit, Textketten (strings) zusätzlich zu Listen in einen Kalkül mit aufzunehmen, mag man sich streiten. Für praktische Programme sind sie natürlich wichtig, aber sie haben wenig Einfluß auf das logische Schließen.

- Die Notwendigkeit *rekursiver Datentypen* und *partiell-rekursiver Funktionen* für die Typentheorie werden wir in Abschnitt 3.5 begründen.

Diese Liste ist bereits sehr umfangreich und man wird kaum Anwendungen finden, die sich mit den obengenannten Konstrukten nicht ausdrücken lassen. Dennoch reichen sie noch nicht ganz aus, da die bisherige Konzeption des Funktion- und Produktraumes zu einfach ist, um alle Aspekte von Funktionen bzw. Tupeln zu charakterisieren. Wir wollen hierzu ein paar Beispiele betrachten.

Beispiel 3.1.1

1. Innerhalb des λ-Kalküls beschreibt der Term $f \equiv \lambda \mathtt{b}.\ \mathsf{if}\ \mathtt{b}\ \mathsf{then}\ \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}\ \mathsf{else}\ \lambda \mathtt{x}.\mathtt{x}$ eine durchaus sinnvolle[7] – wenn auch ungewöhnliche – Funktion, die bei Eingabe eines booleschen Wertes `b` entweder eine Projektionsfunktion oder die Identitätsfunktion als Ergebnis liefert.

   Versucht man, diese Funktion zu typisieren, so stellt man fest, daß der *Typ* des Ergebnisses vom *Wert* der Eingabe abhängt. Bei Eingabe von **T** hat der Ergebnistyp die Struktur $\mathtt{S}\to\mathtt{T}\to\mathtt{S}$, während er die Struktur $\mathtt{S}\to\mathtt{S}$ hat, wenn **F** eingegeben wird.

   Eine einfache Typisierung der Art $f \in \mathbb{B}\to\mathtt{T}$ ist also nicht möglich. Was uns bisher noch fehlt, ist eine Möglichkeit, diese Abhängigkeit zwischen dem Eingabewert $\mathtt{b}:\mathbb{B}$ und dem Ausgabetyp $\mathtt{T}[\mathtt{b}]$[8] auszudrücken. Wir müssen also den Funktionenraumkonstruktor $\to$ so erweitern, daß derartige Abhängigkeiten erfaßt werden können. Man spricht in diesem Fall von einem *abhängigen Funktionenraum* (*dependent function type*), und verwendet als allgemeine Notation $\underline{x:S\to T}$ anstelle des einfacheren $S\to T$, um auszudrücken daß der Typ T vom Wert $x \in S$ abhängen kann.

2. In der Programmiersprache Pascal gibt es den sogenannten *variant record* – ein Konstrukt, daß es erlaubt, verschieden*artige* Tupel in einem Datentyp zusammenzufassen. Eine sinnvolle Anwendung ist z.B. die Beschreibung geometrischer Objekte, für die man – je nachdem ob es sich um Rechtecke, Kreise oder Dreiecke handelt – unterschiedlich viele Angaben benötigt.

```
RECORD
  CASE kind:(RECT,TRI,CIRC) of
     RECT: (länge, breite:real)
     TRI : (a,b,c:real)
     CIRC: (radius:real)
END
```

   Die Elemente dieses Datentyps bestehen also je nach Wert der ersten Komponente `kind` aus drei, vier oder zwei Komponenten. Der Datentyp ist also ein Produktraum der Gestalt $S\times T$, wobei der Typ T je nach Wert des Elements aus S entweder $\mathbb{R}\times\mathbb{R}$, $\mathbb{R}\times\mathbb{R}\times\mathbb{R}$ oder einfach nur $\mathbb{R}$ ist.

   Diese Abhängigkeit des Typs der zweiten Komponente vom Wert der ersten ist in dem üblichen Verständnis des Produktraumkonstruktors $\times$ nicht enthalten. Daher muß auch dieser entsprechend erweitert werden zu einem *abhängigen Produktraum* (*dependent product type*), den man mit $\underline{x:S\times T}$ bezeichnet.

[7] Funktionen dieser Art treten zum Beispiel auf, wenn man Lösungsstrategien für Suchprobleme in der KI programmieren möchte. Das Ergebnis eines Testes legt fest, wieviele Werte man noch abfragen muß, bevor man eine Lösung bestimmen kann.

[8] In Anlehnung an die in Definition 2.3.5 auf Seite 49 vereinbarte Notation kennzeichnet $\mathtt{T}[\mathtt{b}]$ das Vorkommen der freien Variablen `b` im Typausdruck `T`.

3. Auch in der Mathematik kommen derartige Abhängigkeiten öfters vor. So werden zum Beispiel endliche Automaten als 5-Tupel $(Q, \Sigma, q_0, \delta, F)$ definiert, wobei Q und Σ endliche Mengen sind, $q_0 \in Q$, $F \subseteq Q$ und $\delta\colon (Q \times \Sigma) \to Q$. Offensichtlich hängen also die Typen der dritten, vierten und fünften Komponente ab von den Werten der ersten beiden. Die Menge aller endlichen Automaten läßt sich somit nur durch die Verwendung abhängiger Produkträume präzise beschreiben.

4. Ein weiterer Grund, das Typsystem um abhängige Typkonstruktoren zu erweitern, ist die Tatsache, daß wir die Logik innerhalb der Typentheorie simulieren möchten. Aufgrund der Curry-Howard Isomorphie (siehe Abschnitt 2.4.7) können wir Formeln mit Typen identifizieren, deren Elemente genau den Beweisen für diese Formeln entsprechen. Wendet man diesen Gedanken nun auf die logischen Quantoren an, so ergeben sich folgende Zusammenhänge:

   - Um eine allquantifizierte Formel der Gestalt $\forall x\colon T\,.\,P(x)$ zu beweisen, müssen wir zeigen, wie wir für ein beliebiges Element $x \in T$ einen Beweis für $P(x)$ konstruieren. Genau besehen konstruieren wir also eine Funktion, die bei Eingabe eines Wertes $x \in T$ einen Beweis für $P(x)$ bestimmt. Der Ergebnis*typ* dieser Funktion – der mit $P(x)$ identifizierte Typ – hängt also ab vom *Wert* der Eingabe $x \in T$.
   - Um eine existentiell quantifizierte Formel der Gestalt $\exists x\colon T\,.\,P(x)$ zu beweisen, müssen wir ein Element $x \in T$ angeben und zeigen, daß für dieses Element die Eigenschaft $P(x)$ gilt. Genau besehen konstruieren wir also ein Tupel $\langle x, p\rangle$, wobei $x \in T$ und p ein Beweis für $P(x)$ ist. Der Typ der zweiten Komponente, also $P(x)$, hängt also ab vom *Wert* der ersten Komponente $x \in T$.

   Somit sind zur Simulation der Quantoren innerhalb der Typentheorie abhängige Typkonstruktoren unumgänglich.

Insgesamt sind also *abhängige Datentypen* ein notwendiger und sinnvoller Bestandteil einer ausdrucksstarken Theorie, die für sich in Anspruch nimmt, alle relevanten Aspekte von Mathematik und Programmierung formalisieren zu können.[9] Die Hinzunahme abhängiger Datentypen in die Typentheorie bringt jedoch auch einige Probleme mit sich. Während es in der einfachen Typentheorie möglich war, Typausdrücke und Objektausdrücke sauber voneinander zu trennen, verlangt die Abhängigkeit eines Typs T von einem gegebenen Wert $s \in S$, daß wir Objektausdrücke innerhalb von Typausdrücken zulassen müssen und eine *syntaktische* Trennung der beiden Ausdrucksarten kaum möglich sein wird. Die Identifizierung korrekter Objekt- bzw. Typausdrücke wird somit zu einem Problem der Semantik bzw. des Beweiskalküls.

3.1.3 Eigenschaften von Berechnungsmechanismen und Beweiskalkül

Bisher haben wir im wesentlichen über die Verwendbarkeit der Typentheorie als *mathematische* Grundlagentheorie gesprochen. Für ihre Anwendung als praktisch nutzbarer Beweiskalkül zum rechnergestützten Schließen über Programme müssen wir natürlich noch weitere Anforderungen stellen. Dazu gehören insbesondere die algorithmischen Eigenschaften des Reduktionsmechanismus, mit dem der Wert eines Ausdrucks bestimmt werden kann. Diese werden natürlich beeinflußt von der Ausdruckskraft des Kalküls: je mehr wir beschreiben können wollen, um so größere Abstriche müssen wir bei den Eigenschaften machen.

- Es ist offensichtlich, daß wir innerhalb des Schlußfolgerungskalküls *nur terminierenden Berechnungen* gebrauchen können. Andernfalls würde das Halteproblem jegliche Form einer automatischen Unterstützung

[9] Aus mathematischer Sicht kann man einen abhängigen Produktraum $x\colon S \times T$ auch als disjunkte Vereinigung aller Räume $T[x]$ mit $x \in S$ – bezeichnet mit $\sum_{x \in S} T$ – auffassen. Diese Notation, die von vielen Autoren benutzt wird, erfaßt aber nicht, daß es sich bei den Elementen des Datentyps nach wie vor um Tupel der Form $\langle s, t\rangle$ mit $s \in S$ und $t \in T[s]$ handelt, und erscheint daher weniger natürlich. Dasselbe gilt für die abhängigen Funktionenräume, die man entsprechend der Sicht von Funktionen als Menge von Paaren $\langle s, f(s)\rangle$ als unbeschränktes Produkt aller Räume $T[x]$ mit $x \in S$ – bezeichnet mit $\prod_{x \in S} T$ – auffassen kann. Diese Notation berücksichtig nicht, daß es sich bei den Elementen des Datentyps nach wie vor um λ-Terme handelt.

ausschließen. Wir werden in Abschnitt 3.3 sehen, daß sich starke Normalisierbarkeit nicht mit extensionaler Gleichheit und dem Konzept der logischen Falschheit verträgt. Beide Konzepte sind aber von essentieller Bedeutung für das formale Schließen. Starke Normalisierbarkeit ist allerdings auch nicht unbedingt erforderlich, da es – wie bei allen Programmiersprachen – ausreicht, mit einem bestimmten Verfahren zu einem Ergebnis zu kommen.

- Auch wenn wir für automatische Berechnungen eine feste Reduktionsstrategie fixieren, werden wir auf *Konfluenz* nicht verzichten können. Der Grund hierfür ist, daß wir beim Führen formaler Beweise innerhalb des Kalküls mehr Freiheitsgrade lassen müssen als für die Reduktion. Es darf aber nicht angehen, daß wir durch verschiedene Beweise zu verschiedenen Ergebnissen kommen.

- Automatische *Typisierbarkeit* wäre wünschenswert, ist aber nicht zu erwarten. Durch das Vorkommen abhängiger Datentypen bzw. logischer Quantoren ist Typisierung in etwa genauso schwer wie automatisches Beweisen in der Prädikatenlogik. Man muß die Typisierung daher im Beweiskalkül unterbringen.

Eine letzte wünschenswerte Eigenschaft ergibt sich aus der Tatsache, daß die Prädikatenlogik innerhalb der Typentheorie simuliert wird. In seiner bisherigen Konzeption läßt der Kalkül nur die *Überprüfung* der Typkorrektheit zu, was der Kontrolle, ob ein vorgegebener Beweis tatsächlich ein Beweis für eine gegebene Formel ist, entspricht (*proof-checking*). Wir wollen Beweise jedoch nicht nur überprüfen sondern – wie wir es aus Abschnitt 2.2 gewohnt sind – mit dem Kalkül *entwickeln*. Der Kalkül muß daher so ausgelegt werden, daß er eine Beweis*konstruktion* und damit auch die Konstruktion von Programmen aus Spezifikationen zuläßt.

3.2 Systematik für einen einheitlichen Aufbau

Wegen der vorgesehenen Rolle als fundamentale Theorie für Mathematik und Programmierung ist die Intuitionistische Typentheorie als eine *konstruktive semantische Theorie* ausgelegt, also als eine Theorie, die nicht nur aus Syntax und Inferenzregeln besteht, sondern darüber hinaus auch noch die Semantik ihrer Terme beschreiben muß. Letztere wird erklärt über das Konzept der Auswertung (Reduktion) von Termen und durch eine methodische Anlehnung an die in Abschnitt 2.4.5.2 vorgestellte *TAIT computability method* (d.h. Abstützung auf 'berechenbare' Elemente) wird sichergestellt, daß diese Reduktionen immer terminieren.

In diesem Abschnitt werden wir die allgemeinen Prinzipien vorstellen, nach denen diese Theorie aufgebaut ist, und am Beispiel dreier Typkonstrukte – dem Funktionenraum, dem Produktraum und der disjunkten Vereinigung – erläutern. Diese Prinzipien regeln den allgemeinen Aufbau von Syntax, Semantik und Inferenzsystem und gehen im wesentlichen nach folgenden Gesichtspunkten vor.

Syntax: Um die formale Sprache festzulegen, sind zulässige Terme (Ausdrücke), bindende Vorkommen von Variablen in diesen Termen und die Ergebnisse von Substitutionen zu definieren. Eine syntaktische Unterscheidung von Objekt- und Typausdrücken wird nicht vorgenommen. Dies wird ausschließlich auf semantischer Ebene geregelt.

Semantik: Die Terme werden unterteilt in *kanonische* und *nichtkanonische* Terme. Der *Wert* eines kanonischen Terms ist der Term selbst, wobei man davon ausgeht, daß ein kanonischer Term immer mit einer intuitiven Bedeutung verbunden ist. Der Wert eines (geschlossenen) nichtkanonischen Ausdrucks wird durch *Reduktion* auf einen kanonischen Term bestimmt.[10]

Die grundsätzlichen Eigenschaften von Termen und die Beziehungen zwischen Termen werden durch *Urteile* (judgments) definiert. Innerhalb der Typentheorie regeln diese Urteile Eigenschaften wie Gleichheit, Typzugehörigkeit und die Tatsache, daß ein Term einen Typ beschreibt. Für kanonische Terme

[10] Hierbei ist anzumerken, daß sich das Konzept "kanonischer Term" oft nur auf die äußere Struktur eines Terms bezieht, der im Inneren durchaus noch reduzierbare Ausdrücke enthalten kann. Damit unterscheidet sich das Konzept der "Normalform" eines Terms von der in Abschnitt 2.3.7 definierten Form, die erst erreicht ist, wenn *alle* reduzierbaren Ausdrücke eliminiert sind.

wird die Semantik dieser Urteile explizit (rekursiv) definiert, für nichtkanonische Terme ergibt sie sich aus der Semantik der gleichwertigen kanonischen Terme.[11] Das Konzept des Urteils wird erweitert auf *hypothetische Urteile* um ein Argumentieren unter bestimmten Annahmen zu ermöglichen.

Inferenzsystem: Urteile und hypothetische Urteile werden syntaktisch durch Sequenzen repräsentiert[12] und die Semantik von Urteilen durch Inferenzregeln ausgedrückt. Dabei ist aber zu berücksichtigen, daß sich dieser Zusammenhang nicht vollständig beschreiben läßt, was insbesondere für die Eigenschaft, ein Typ zu sein, gilt.

Aus diesem Grunde muß die Syntax um einige Ausdrücke (wie Typuniversen und einen Gleichheitstyp) erweitert werden, deren einziger Zweck es ist, die in den Urteilen benannte Zusammenhänge innerhalb der formalen Sprache auszudrücken. Semantisch liefern diese Ausdrücke keinerlei neue Informationen, da ihre Semantik nahezu identisch mit der des zugehörigen Urteils ist.

An einigen Stellen mußten bei der Entwicklung der Typentheorie Entwurfsentscheidungen getroffen werden, die genausogut auch anders hätten ausfallen können. Wir werden diese und die Gründe, die zu dieser Entscheidung geführt haben, an geeigneter Stelle genauer erklären.

In der nun folgenden Beschreibung der Methodik für einen einheitlichen Aufbau der Theorie werden wir uns nicht auf alle Details einlassen, die bei der systematischen Konzeption des NuPRL Systems eine Rolle gespielt haben, sondern nur diejenigen Aspekte betrachten, die für einen eleganten Aufbau der eigentlichen Theorie und ihr Erscheinungsbild auf dem Rechner bedeutend sind.

3.2.1 Syntax formaler Ausdrücke

Bei der Festlegung der Syntax der formalen Sprache, also der zulässigen Ausdrücke, dem freien und gebundenen Vorkommen von Variablen und der Substitution von Variablen durch Terme könnte man einfach die Definitionen aus Abschnitt 2.4.1 übernehmen und entsprechend für die neu hinzugekommenen Konzepte erweitern. Für die Datentypen (abhängiger) Funktionenraum, (abhängiger) Produktraum und disjunkte Vereinigung und ihre zugehörigen Operationen ergäbe sich dann folgende Definition von Ausdrücken:

Es sei $\mathcal{V}$ eine unendliche Menge von Variablen. Ausdrücke sind induktiv wie folgt definiert.

1. *Jede Variable $x \in \mathcal{V}$ ist ein Ausdruck.*
2. *Ist t ein beliebiger Ausdruck, dann ist (t) ein Ausdruck.*
3. *Ist $x \in \mathcal{V}$ eine Variable und t ein beliebiger Ausdruck, dann ist $\lambda x.t$ ein Ausdruck.*
4. *Sind t und f beliebige Ausdrücke, dann ist $f\ t$ ein Ausdruck.*
5. *Ist $x \in \mathcal{V}$ eine Variable und sind S und T beliebige Ausdrücke, so sind $x{:}S{\to}T$ und $S{\to}T$ Ausdrücke.*
6. *Sind s und t beliebige Ausdrücke, dann ist $\langle s, t\rangle$ ein Ausdruck.*
7. *Sind $x, y \in \mathcal{V}$ Variablen sowie e und u beliebige Ausdrücke, dann ist $\mathsf{let}\ \langle x, y\rangle = e\ \mathsf{in}\ u$ ein Ausdruck.*
8. *Ist $x \in \mathcal{V}$ eine Variable und sind S und T beliebige Ausdrücke, so sind $x{:}S{\times}T$ und $S{\times}T$ Ausdrücke.*
9. *Sind s und t beliebige Ausdrücke, dann sind $\mathsf{inl}(s)$ und $\mathsf{inr}(t)$ Ausdrücke.*
10. *Sind $x, y \in \mathcal{V}$ Variablen sowie e, u und v beliebige Ausdrücke, dann ist $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v$ ein Ausdruck.*
11. *Sind S und T beliebige Ausdrücke, dann ist $S{+}T$ ein Ausdruck.*

[11] Hierdurch wird klar, warum Berechnungen terminieren müssen. Ohne diese Eigenschaft wäre die Semantik von Aussagen über nichtkanonische Terme nicht definiert.

[12] Es sei an dieser Stelle angemerkt, daß der Unterschied zwischen einem Urteil und seiner Repräsentation als Sequenz oft genauso verwischt wird wie der Unterschied zwischen einer Zahl und ihrer Dezimalschreibweise. Die Urteile beschreiben das semantische Konzept, während die Sequenzen die textliche Form sind, dies aufzuschreiben. Die Möglichkeit, durch die Einbettung der Sequenz in die formale Sprache wieder über die Semantik der Theorie selbst zu reflektieren ist durchaus gewollt.

Der Nachteil dieser Definition (die wir absichtlich nicht numeriert haben) ist, daß nicht nur eine Fülle von Ausdrücken definiert wird sondern auch völlig unterschiedliche Strukturen von Ausdrücken eingeführt werden, die deswegen jeweils eine unabhängige Zeile in der induktiven Definition verlangen. Dieser Nachteil wird noch deutlicher, wenn man sich die entsprechende Definition freier und gebundener Variablen ansieht.

Es seien $x, y, z \in \mathcal{V}$ Variablen sowie s, t, u, v, e, S und T Ausdrücke. Das freie und gebundene Vorkommen der Variablen x in einem Ausdruck ist induktiv durch die folgenden Bedingungen definiert.

1. *Im Ausdruck x kommt x frei vor. Die Variable y kommt nicht vor, wenn x und y verschieden sind.*
2. *In (t) bleibt jedes freie Vorkommen von x in t frei und jedes gebundene Vorkommen gebunden.*
3. *In $\lambda x.t$ wird jedes freie Vorkommen von x in t gebunden. Gebundene Vorkommen von x in t bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn x und y verschieden sind.*
4. *In $f\,t$ bleibt jedes freie Vorkommen von x in f oder t frei und jedes gebundene Vorkommen von x in f oder t gebunden.*
5. *In $x{:}S{\to}T$ wird jedes freie Vorkommen von x in T gebunden. Freie Vorkommen von x in S bleiben frei. Gebundene Vorkommen von x in S oder T bleiben gebunden. Jedes freie Vorkommen der Variablen y in T bleibt frei, wenn x und y verschieden sind.*

   In $S{\to}T$ bleibt jedes freie Vorkommen von x in S oder T frei und jedes gebundene Vorkommen gebunden.
6. *In $\langle s,t\rangle$ bleibt jedes freie Vorkommen von x in s oder t frei und jedes gebundene Vorkommen gebunden.*
7. *In $\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u$ wird jedes freie Vorkommen von x und y in u gebunden. Freie Vorkommen von x oder y in e bleiben frei. Gebundene Vorkommen von x oder y in e oder u bleiben gebunden. Jedes freie Vorkommen der Variablen z in u bleibt frei, wenn z verschieden von x und y ist.*
8. *In $x{:}S{\times}T$ wird jedes freie Vorkommen von x in T gebunden. Freie Vorkommen von x in S bleiben frei. Gebundene Vorkommen von x in S oder T bleiben gebunden. Jedes freie Vorkommen der Variablen y in T bleibt frei, wenn x und y verschieden sind.*

   In $S{\times}T$ bleibt jedes freie Vorkommen von x in S oder T frei und jedes gebundene Vorkommen gebunden.
9. *In $\mathsf{inl}(s)$ bzw. $\mathsf{inr}(t)$ bleibt jedes freie Vorkommen von x in s bzw. t frei und jedes gebundene Vorkommen gebunden.*
10. *In $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v$ wird jedes freie Vorkommen von x in u bzw. von y in v gebunden. Freie Vorkommen von x oder y in e bleiben frei. Gebundene Vorkommen von x oder y in e, u oder v bleiben gebunden. Jedes freie Vorkommen der Variablen z in u bzw. v bleibt frei, wenn z verschieden von x bzw. y ist.*
11. *In $S{+}T$ bleibt jedes freie Vorkommen von x in S oder T frei und jedes gebundene Vorkommen gebunden.*

Wie man sieht, ufern die Definitionen sehr schnell aus, obwohl wir bisher nur drei der uns interessierenden Typkonstrukte beschrieben haben. Eine Systematik – besonders bei den bindenden Vorkommen von Variablen in Termen – ist ebenfalls nicht zu erkennen, da die Variablen innerhalb eines Termes an nahezu beliebigen Positionen vorkommen dürfen. Würden wir diese Vorgehensweise bei der Einführung der weiteren Typkonzepte beibehalten, so könnte alleine die Syntax der Sprache schnell unübersichtlich werden. Zudem wären wir gezwungen, die Definition der Ausdrücke ständig zu erweitern.

Das Hauptproblem der obigen Vorgehensweise liegt darin, daß die verschiedenen Ausdrücke in ihrer Struktur extrem unheitlich sind, da sie sich an lange vertraute Notationen anlehnen. Ein einheitlicher und systematischer Aufbau kann daher nur erreicht werden, wenn man von der Notation abstrahiert und die Beschreibung von Ausdrücken auf das eigentlich Wesentliche reduziert. Wir wollen hierfür einige Beispiele betrachten.

Beispiel 3.2.1

Die Paarbildung $\langle s,t\rangle$ ist eigentlich nur eine elegantere Notation für die Anwendung eines Tupeloperators auf die Ausdrücke s und t, also eine Abkürzung für einen Ausdruck der Form $\mathbf{pair}(s;t)$.

Genauso ist die Applikation $f\ t$ nur eine Abkürzung für einen Ausdruck der Form $\mathbf{apply}(f;t)$[13] und dasselbe gilt für den Ausdruck $\mathbf{union}(S;T)$, der einen Ausdruck der Form $S+T$ ankürzt.

Eine wesentlich einheitlichere Notation für Ausdrücke ist also die aus Definition 2.2.2 (Seite 24) bekannte Termschreibweise. Zum Zwecke der Systematisierung der Theorie sollte ein Ausdruck (Term) immer die Gestalt $op(t_1;\ldots;t_n)$ haben, wobei op ein Operator – wie pair, apply oder union – ist und die t_i Ausdrücke sind.

Mit dieser Beschreibungsform können jedoch noch nicht alle Ausdrücke erfaßt werden, da sie nicht in der Lage ist, die *Bindung* von Variablen in Teiltermen zu kennzeichnen. Wir müssen daher eine gewisse Erweiterung vornehmen.

Beispiel 3.2.2

Die λ-Abstraktion $\lambda x.t$ beschreibt einen Ausdruck, in dem die Variable x innerhalb des Terms t gebunden werden soll. Eine Ausdruck der Form $\mathsf{lam}(x;t)$ würde diesen Zusammenhang nicht wiederspiegeln können, da hier x und t als unabhängige Teilterme auftreten, die jeweils ihren eigenen Wert haben. In Wirklichkeit wollen wir jedoch kennzeichnen, daß x eine *Variable* und nicht etwa ein Ausdruck ist, die innerhalb von t vorkommen kann und dort gebunden werden soll. x und t bilden also eine Einheit, die man als *gebundenen Term* bezeichnet und zum Beispiel mit $\underline{x.t}$ kennzeichnen kann. Eine angemessene Termschreibweise für $\lambda x.t$ wäre also $\mathbf{lam}(x.t)$.

Die einheitliche Beschreibungsform für den abhängigen Funktionenraum $x{:}S{\to}T$ ist dementsprechend $\mathbf{fun}(S;x.T)$ und die für den abhängigen Produktraum $x{:}S{\times}T$ ist $\mathbf{prod}(S;x.T)$. Die einheitliche Notation macht übrigens auch deutlich, daß die Variable x im Typ T gebunden ist und nicht etwa in S. Die Tatsache, daß die Werte für x aus dem Typ S zu wählen sind, ist ein semantischer Zusammenhang und hat mit einer syntaktischen Codierung nichts zu tun.

Der Term, der sich hinter der Notation $\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u$ verbirgt, lautet $\mathbf{spread}(e;x,y.u)$. Hier werden zwei Variablen innerhalb des Terms u gebunden, was durch die Notation $\underline{x,y.u}$ gekennzeichnet wird. Die Notation $\mathbf{decide}(e;x.u;y.v)$ (für $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v$) macht ebenfalls deutlicher, welche Variablen in welchen Termen gebunden werden.

Entsprechend der obigen Beispiele müssen wir also die Beschreibungsform für Ausdrücke ausdehnen auf die Gestalt $op(b_1;\ldots;b_n)$, wobei op ein Operator ist und die b_i gebundene Terme sind. Gebundene Terme wiederum haben die Gestalt $x_1,\ldots,x_n.t$, wobei die x_i Variablen sind und t ein Term (Ausdruck) ist.

Die bisherigen Beispiele zeigen, wie wir aus Termen und Operatoren neue Terme aufbauen können. Womit aber fangen wir an? Was sind die Basisterme unserer Theorie? Auch hierzu geben wir ein Beispiel.

Beispiel 3.2.3

In allen bisherigen Definitionen der Syntax formaler Sprachen wurden Variablen $x\in\mathcal{V}$ als atomare Terme bezeichnet. Diese Gleichsetzung ist aber eine grobe Vereinfachung, denn genau besehen sind Terme strukturierte *Objekte*, die einen Wert besitzen können, während Variablen *Platzhalter* sind, deren wesentliches Merkmal ihr Name ist. Die Aussage "jede Variable $x\in\mathcal{V}$ ist ein Term" nimmt implizit eine Konversion vor. Bei einer präzisen Definition müßte man allerdings eine Unterscheidung vornehmen zwischen der Variablen x und dem Term, der nur die Variable x beinhaltet, und die Konversion explizit machen. Wenn wir unser bisheriges Konzept beibehalten und sagen, daß Terme nur durch Anwendung von Operatoren entstehen können, dann muß diese Konversion von Variablen in Terme einem Operator var entsprechen, der auf Variablen angewandt wird. Diese Anwendung des Operators var auf eine Variable ist allerdings etwas anderes als die Anwendung von Operatoren auf (gebundene) Terme, die als Argumente des Operators auftauchen. Deshalb soll diese Anwendungsform auch in ihrer Notation deutlich von der anderen

[13]Diese Abkürzung hat sich aber mittlerweile so sehr eingebürgert, daß sie oft verwechselt wird mit der Anwendung des Operators f auf den Ausdruck t. In Wirklichkeit wird aber nur ein neuer Ausdruck gebildet und die Funktion f wird erst auf t angewandt, wenn der Reduktionsmechanismus gestartet wird. Die Schreibweise $\mathbf{apply}(f;t)$ bringt die Natur dieses Ausdrucks also auch viel deutlicher zum Vorschein.

unterschieden werden und wir schreiben $\mathtt{var}\{x\mathtt{:variable}\}()$, um den Term zu präzisieren, der gängigerweise einfach mit x bezeichnet wird. x ist also ein Parameter des Operators var und das Schlüsselwort variable kennzeichnet hierbei, daß es sich bei x um einen Variablennamen handelt.

Diese Präzisierung mag vielleicht etwas spitzfindig erscheinen, aber sie erlaubt eine saubere Trennung verschiedenartiger Konzepte innerhalb einer einheitlichen Definition des Begriffs "Term". Der Aufwand wäre sicherlich nicht gerechtfertigt, wenn Konversionen nur bei Variablen implizit vorgenomen würden. Bei Zahlen und anderen Konstanten, die wir als Terme ansehen wollen, stoßen wir jedoch auf dieselbe Problematik. Gemäß Definition 2.2.2 müssen alle Konstanten als Anwendungen nullstelliger Operatoren angesehen werden. Die Parametrisierung von Operatoren ermöglicht es nun, diese Operatoren unter *einem* Namen zu einer Familie gleichartiger Operatoren zusammenzufassen

Da es noch weitere, hier nicht betrachtete Gründe geben mag, Operatoren zu parametrisieren, dehnen wir die Beschreibungsform für Ausdrücke ein weiteres Mal aus. Insgesamt haben Ausdrücke nun die Gestalt

$$opid\{P_1,\ldots,P_n\}(b_1;\ldots;b_n),$$

wobei *opid* der Name für eine Operatorfamilie ist, die P_i seine Parameter (samt Kennzeichnung, um welche Art Parameter es sich handelt) sind und die b_i gebundene Terme. Gebundene Terme wiederum haben die Gestalt $x_1,\ldots,x_n.t$, wobei die x_i Variablen sind und t ein Term (Ausdruck) ist.

Diese einheitliche Schreibweise, die übrigens auch jede Art zusätzlicher Klammern überflüssig macht, beschreibt die wesentlichen Aspekte von Termen und ermöglicht sehr einfache Definitionen des Termbegiffs, gebundener und freier Variablen und von Substitutionen.

Natürlich wollen wir auf die vertrauten Notationen nicht ganz verzichten, denn dies wäre ja im Sinne einer praktisch nutzbaren Theorie ein ganz gewaltiger Rückschritt. Die einheitliche, abstrakte Notation hilft beim systematischen Aufbau einer Theorie und ihrer Verarbeitung mit dem Rechner. Für den Menschen, der mit den Objekten (Termen) dieser Theorie umgehen muß, ist sie nicht sehr brauchbar. Aus diesem Grunde wollen wir die textliche Darstellung eines Terms – die sogenannte *Display Form* – von seiner abstrakten Darstellung in der einheitlichen Notation – seine *Abstraktionsform* – unterscheiden und beide Formen simultan betrachten. In formalen Definitionen, welche die Theorie als solche erklären, verwenden wir die formal einfachere Abstraktionsform, während wir bei der Charakterisierung und Verarbeitung konkreter Terme soweit wie möglich auf die Display Form zurükgreifen werden. Die Eigenschaften dieser Terme ergeben sich dann größtenteils aus den allgemeinen Definitionen und dem Zusammenhang zwischen einer konkreten Abstraktion und ihrer Display Form.[14] Wir wollen diese Entscheidung als ein erstes wichtiges Prinzip der Theorie fixieren.

Entwurfsprinzip 3.2.4 (Trennung von Abstraktion und textlicher Darstellung)

In der Typentheorie werden die Abstraktionsform eines Terms und seine Darstellungsform getrennt voneinander behandelt aber simultan betrachtet.

Dieses Prinzip gilt gleichermaßen für die vordefinierten Terme der Typentheorie und für konservative Erweiterungen durch einen Anwender der Theorie.[15] Die Display Form beschreibt darüber hinaus auch alle wichtigen Eigenschaften, die für eine eindeutige Lesbarkeit der textlichen Darstellung nötig sind, wie zum

[14] Im NuPRL System wird diese Vorgehensweise dadurch unterstützt, daß man üblicherweise die Display Form eines Termes gezeigt bekommt, die im Prinzip frei wählbar ist, während das System intern die Abstraktionsform verwaltet, die man nur auf Wunsch zu sehen bekommt.

[15] Dies hat auch Auswirkungen auf die Art, wie konservative Erweiterungen innerhalb der Theorie behandelt werden. Während im Abschnitt 2.1.5 eine formale Definition im wesentlichen als textliche Abkürzung verstanden wurde, die innerhalb eines Beweiseditors als Textmacro behandelt werden kann, besteht eine konservative Erweiterung nun aus zwei Komponenten: einer *Abstraktion*, die ein neues Operatorsymbol in die sogenannte Operatorentabelle einträgt, und einer *Display Form*, welche die textliche Darstellung von Termen beschreibt, die mit diesem Operator gebildet werden (siehe Abschnitt 4.1.7). Um also zum Beispiel den Typ $\mathbb{N}$ der natürlichen Zahlen als Teilmenge der ganzen Zahlen zu definieren, definiert man zunächst als Abstraktion:

$\mathtt{nat}\{\}() \equiv \{\mathtt{n}\mathtt{:}\mathbb{Z}\,|\,\mathtt{n}\geq\mathtt{0}\}$

Hierdurch wird ein Operator mit Namen nat definiert und seine Bedeutung erklärt.
Weiterhin wird festgelegt, wie dieser Operator üblicherweise zu schreiben ist.

$\mathbb{N} \equiv \mathtt{nat}\{\}()$

`variable` : Variablennamen (ML Datentyp `var`)
Zulässige Elemente sind Zeichenketten der Form $[\texttt{a}-\texttt{z}\,\texttt{A}-\texttt{Z}\,\texttt{0}-\texttt{9}\,\texttt{\_}\,\texttt{-}\,\texttt{\%}]^+$

`natural` : Natürliche Zahlen einschließlich der Null (ML Datentyp `int`).
Zulässige Elemente sind Zeichenketten der Form $\texttt{0} + [\texttt{1}-\texttt{9}][\texttt{0}-\texttt{9}]^*$.

`token` : Zeichenketten für Namen (ML Datentyp `tok`).
Zulässige Elemente bestehen aus allen Symbolen des NuPRL Zeichensatzes mit Ausnahme der Kontrollsymbole.

`string` : Zeichenketten für Texte (ML Datentyp `string`).
Zulässige Elemente bestehen aus allen Symbolen des NuPRL Zeichensatzes mit Ausnahme der Kontrollsymbole.

`level-expression` : Ausdrücke für das Level eines Typuniversums (ML Datentyp `level_exp`)
Die genaue Syntax wird in Abschnitt 3.2.3.1 bei der Diskussion der Universenhierarchie besprochen.

Die Namen für Parametertypen dürfen durch ihre ersten Buchstaben abgekürzt werden.

Abbildung 3.1: Parametertypen und zugehörige Elemente

Beispiel Prioritäten und Assoziativität. Durch die Trennung von Abstraktion und Darstellungsform, die vom NuPRL System voll unterstützt wird, gewinnen wir maximale Flexibilität bei der formalen Repräsentation von Konstrukten aus Mathematik und Programmierung innerhalb eines computerisierten Beweissystems.

Nach all diesen Vorüberlegungen ist eine präzise Definition von Termen leicht zu geben.

Definition 3.2.5 (Terme)

1. *Es sei $\mathcal{I}$ eine Menge von Namen für Indexfamilien. Ein Parameter hat die Gestalt $p{:}F$, wobei $F \in \mathcal{I}$ eine Indexfamilie (Parametertyp) und p (Parameterwert) ein zulässiges Element von F ist.*

   Die Namen der vordefinierten Indexfamilien und ihrer zugehörigen Elemente sind den Einträgen der aktuellen Indexfamilientabelle zu entnehmen, die mindestens die Familie `variable` *enthalten muß.*

2. *Ein Operator hat die Gestalt $opid\{P_1, \ldots, P_n\}$ ($n \in \mathbb{N}$), wobei opid ein Operatorname und die P_i Parameter sind.*

3. *Es sei $\mathcal{V}$ eine unendliche Menge von Variablen (Elementen der Indexfamilie* `variable`*).*
   Terme und gebundene Terme sind induktiv wie folgt definiert.
   - *Ein gebundener Term hat die Gestalt $x_1, \ldots, x_m.t$ ($m \in \mathbb{N}$), wobei t ein Term ist und die x_i Variablen sind.*
   - *Ein Term hat die Gestalt $op(b_1; \ldots; b_n)$ ($n \in \mathbb{N}$), wobei op ein Operator und die b_i gebundene Terme sind.*

4. *Die Namen der vordefinieren Operatoren, ihre Parameter, Stelligkeiten und die zulässigen (gebundenen) Teilterme sind den Einträgen in der Operatorentabelle zu entnehmen, die mindestens den Operator* `var` *enthalten muß.*[16]

Diese Definition regelt die grundlegende Struktur von Termen, nicht aber die konkrete Ausprägung der vordefinierten Konzepte der Theorie. Sowohl die Indexfamilien als auch die Operatoren sind über Tabellen zu

Im Endeffekt bedeutet eine solche Definition also eine Erweiterung der Tabelle der vordefinierten Operatoren. Der einzige Unterschied zu diesen ist, daß aufgrund der Abstützung auf bereits bekannte Terme die Semantik und die Inferenzregeln des neu definierten Operators nicht neu erklärt und auf Konsistenz überprüft werden müssen. Beides ergibt sich automatisch aus der Semantik und den Regeln der Grundoperatoren und ist konsistent mit dem Rest der Theorie.

[16] Durch diese Festlegung erhalten wir die Bedingung: '*Ist $x \in \mathcal{V}$, so ist* `var{x:v}()` *ein Term*' als einen Grundbestandteil jeder auf dieser Definition aufbauenden Theorie, also mehr oder weniger die Aussage '*jede Variable $x \in \mathcal{V}$ ist ein Term*'.

definieren, wobei in NuPRL als Einschränkung gilt, daß Operatornamen nichtleere Zeichenketten über dem Alphabet {a−z A−Z 0−9 _ - !} sein müssen. Die Operatorentabelle werden wir in diesem Kapitel schrittweise zusammen mit der Diskussion der verschiedenen Typkonstrukte aufbauen. Die aktuelle Tabelle der Indexfamilien (Parametertypen) ist dagegen relativ klein und komplett in Abbildung 3.1 zusammengestellt.

Für syntaktische Manipulationen und die Beschreibung des Berechnungsmechanismus ist das Konzept der Substitution von frei vorkommenden Variablen(-termen) durch Terme von großer Bedeutuug. Aufgrund der einfachen und einheitlichen Notation für Terme ist es nicht schwer, die hierbei notwendigen Begriffe präzise zu definieren. Im wesentlichen läuft dies darauf hinaus, daß Terme alle vorkommenden Variablen frei enthalten, solange diese nicht innerhalb eines gebundenen Terms an die Variable vor dem Punkt gebunden sind.

Definition 3.2.6 (Freies und gebundenes Vorkommen von Variablen in Termen)

Es seien $x, x_1, \ldots, x_m, y \in \mathcal{V}$ Variablen, $t, b_1, \ldots, b_n$ Terme und op ein Operator. Das freie und gebundene Vorkommen der Variablen x in einem Term ist induktiv durch die folgenden Bedingungen definiert.

1. *Im Term $\mathsf{var}\{x:\mathtt{v}\}$ kommt x frei vor. y kommt nicht vor, wenn x und y verschieden sind.*
2. *In $op(b_1;\ldots;b_n)$ bleibt jedes freie Vorkommen von x in den b_i frei und jedes gebundene Vorkommen gebunden.*
3. *In $x_1, \ldots, x_m.t$ wird jedes freie Vorkommen der Variablen x_i in t gebunden. Gebundene Vorkommen von x_i in t bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn y verschieden von allen x_i ist.*

Das freie Vorkommen der Variablen $x_1, \ldots, x_n$ in einem Term t wird mit $t[x_1, \ldots, x_n]$ gekennzeichnet.

Eine Term ohne freie Variablen heißt geschlossen.

Auch das Konzept der Substitution hat nun eine relativ einfache Definition, die den Definitionen 2.2.20 und 2.3.7 (Seite 37 bzw. 50) relativ nahe kommt. Substitution der Variablen x durch den Term t innerhalb eines Terms u beedeutet, daß jedes freie Vorkommen von x in u durch t ersetzt wird, während jedes gebundene Vorkommen von x in u unverändert bleibt. Hierbei ist jedoch zu beachten, daß nicht etwa die Variable x selbst durch t ausgetauscht wird[17] sondern der Term, der aus der Variablen x gebildet wird, also $\mathsf{var}\{x:\mathtt{v}\}()$.

Definition 3.2.7 (Substitution)

Eine Substitution ist eine endliche Abbildung σ von der Menge der Terme der Gestalt $\mathsf{var}\{x:\mathtt{v}\}()$ mit $x \in \mathcal{V}$ in die Menge der Terme.

Gilt $\sigma(\mathsf{var}\{x_1:\mathtt{v}\}())=t_1$, ..., $\sigma(\mathsf{var}\{x_n:\mathtt{v}\}())=t_n$, so schreiben wir kurz $\sigma = [t_1, \ldots, t_n/x_1, \ldots, x_n]$.

Die Anwendung einer Substitution $\sigma=[t/x]$ auf einen Term u – bezeichnet durch $u[t/x]$ – ist induktiv wie folgt definiert.

$\mathsf{var}\{x:\mathtt{v}\}()[t/x]$	$=$	t
$\mathsf{var}\{x:\mathtt{v}\}()[t/y]$	$=$	$\mathsf{var}\{x:\mathtt{v}\}()$, *wenn x und y verschieden sind.*
$(op(b_1;\ldots;b_n))[t/x]$	$=$	$op(b_1[t/x];\ldots;b_n[t/x])$
$(x_1,\ldots,x_n.u)[t/x]$	$=$	$x_1,\ldots,x_n.u$, *wenn x eines der x_i ist.*
$(x_1,\ldots,x_n.u)[t/x]$	$=$	$x_1,\ldots,x_n.u[t/x]$ *wenn x verschieden von allen x_i ist, und es der Fall ist, daß x nicht frei in u ist oder daß keines der x_i frei in t vorkommt.*
$(x_1,..,x_j,..,x_n.u)[t/x]$	$=$	$(x_1,..,z,..,x_n.u[z/x_j])[t/x]$ *wenn x verschieden von allen x_i ist, frei in u vorkommt und x_j frei in t erscheint. z ist eine neue Variable, die weder in u noch in t vorkommt.*

Dabei sind $x, x_1, \ldots, x_m, y \in \mathcal{V}$ Variablen, $t, b_1, \ldots, b_n$ Terme und op ein Operator.

Die Anwendung komplexerer Substitutionen auf einen Term – $u[t_1, \ldots, t_n/x_1, \ldots, x_n]$ – wird entsprechend durch ein simultane Induktion definiert.

[17] Die Ersetzung von x durch λx.x im Term x würde in diesem Fall zu einem Konstrukt führen, dessen Abstraktionsform **var**{**lam**(x.**var**{x:v}()):v}() ist anstelle des gewünschten **lam**(x.**var**{x:v}()).

Operator und Termstruktur	*Display Form*
var$\{x:\mathtt{v}\}()$	x
fun$\{\}(S;x.T)$	$x:S\to T$
lam$\{\}(x.t)$	$\lambda x.t$
apply$\{\}(f;t)$	$f\ t$
prod$\{\}(S;x.T)$	$x:S\times T$
pair$\{\}(s;t)$	$\langle s,t\rangle$
spread$\{\}(e;x,y.u)$	$\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u$
union$\{\}(S;T)$	$S+T$
inl$\{\}(s)$, **inr**$\{\}(t)$	$\mathsf{inl}(s)$, $\mathsf{inr}(t)$
decide$\{\}(e;x.u;y.v)$	$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v$

Abbildung 3.2: Operatorentabelle für Funktionenraum, Produktraum und Summe

Wir wollen die Auswirkungen dieser Definitionen am Beispiel von Funktionenraum, Produktraum und disjunkter Vereinigung illustrieren.

Beispiel 3.2.8

Abbildung 3.2 zeigt die Einträge der Operatorentabelle für Variablen, Funktionenraum, Produktraum und Summe. Mit Ausnahme des Operators var sind alle Operatoren parameterlos. Ein Eintrag wie **fun**$\{\}(S;x.T)$ besagt, daß ein Operator mit Namen fun definiert wird, der zwei Argumente besitzt, von denen das erste ein ungebundener Term und das zweite ein mit einer Variablen gebundener Term sein muß. x, S und T sind Meta-Variablen zur Beschreibung dieser Tatsache und ihr Vorkommen in der Display Form gibt an, an welcher Stelle die verschiedenen Komponenten der Argumente von fun in der textlichen Darstellung wiederzugeben sind.

Gemäß Definition 3.2.6 wird eine Variable in T gebunden, wenn sie mit der für x eingesetzten Variablen identisch ist. Alle anderen Variablen behalten ihren Status.

Die weiteren Einträge spiegeln genau die 'Definitionen' von Termen und freien/gebundenen Vorkommen von Variablen wieder, die wir zu Beginn dieses Abschnitts aufgelistet haben.

3.2.2 Semantik: Werte und Eigenschaften typentheoretischer Ausdrücke

Anders als bei den im vorigen Kapitel vorgestellten Kalkülen wollen wir die Semantik der Typentheorie nicht auf einer anderen Theorie wie der Mengentheorie abstützen, sondern auf der Basis eines intuitiven Verständnisses gewisser mathematischer Grundkonstrukte formal definieren. Dabei gehen wir davon aus, daß elementare Grundkonstrukte wie die natürliche Zahlen eine natürliche und unumstrittene Bedeutung haben und erklären die Bedeutung komplexerer Konstrukte durch (rekursive) Definitionen unter Verwendung einer mathematischen Sprache, die ebenfalls nicht weiter erklärt werden muß. Wir erklären zunächst, wie der Wert eines typentheoretischen Ausdrucks zu bestimmen ist, und beschreiben dann auf dieser Basis die semantischen Eigenschaften von Termen.

3.2.2.1 Der Wert eines Ausdrucks

Typentheoretische Ausdrücke werden als Repräsentanten mathematischer Objekte angesehen. Unter diesen gibt es Terme wie `0,1,-1,2,-2,...` oder λ`x.4`, die wir als Standard-Darstellung eines Objektes ansehen und solche, die als noch auszuwertende Ausdrucke betrachtet werden. Bei den ersteren gehen wir davon aus, daß sie mehr oder weniger einen direkten Bezug zu einem Objekt haben, der intuitiv klar ist bzw. nur mit intuitiven mathematischen Konzepten erklärt werden kann. Diese *kanonischen* Terme können also als (Standard-Repräsentanten für) Werte der Theorie angesehen werden. Die Bedeutung anderer, *nichtkanonischer* Terme muß dagegen durch eine Auswertung zu einem kanonischen Term bestimmt werden.

Für eine autarke Beschreibung der Semantik der Typentheorie ist also die Auswertung oder *Reduktion* von essentieller Bedeutung. Um zu garantieren, daß diese Reduktion (für typisierbare Terme gemäß Abschnitt 3.2.2.2) immer terminiert, nehmen wir eine gewisse methodische Anlehnung an die in Abschnitt 2.4.5.2 vorgestellte *TAIT computability method* vor. Bestimmte Terme werden durch syntaktische Kriterien als kanonisch definiert und gelten als nicht weiter reduzierbare Werte. Für die nichtkanonischen Ausdrücke werden bestimmte Argumente als *Hauptargumente* gekennzeichnet und Reduktion (Berechenbarkeit) wird definiert durch das Ergebnis einer Auswertung nichtkanonischer Ausdrücke mit kanonischen Hauptargumenten. Dieses Ergebnis kann anhand einer Tabelle von *Redizes* und *Kontrakta* bestimmt werden, die sich auf das (syntaktische) Konzept der Substitution stützt. So ist zum Beispiel f das Hauptargument eines Ausdrucks **apply**{}$(f\,;t)$ und Reduktion wird erklärt durch Auswertung von Redizes der Art **apply**{}(**lam**{}$(x\,.\,u)\,;t)$ (also $(\lambda x\,.\,u)\ t$) zum Term $u[t/x]$.

Dies alleine garantiert natürlich noch keine immer terminierenden Reduktionen. Es hängt davon ab, *welche* Terme als kanonisch bezeichnet werden. Für eine Theorie, die so ausdrucksstark ist wie in Abschnitt 3.1.2 gefordert, müssen wir dafür sorgen, daß Reduktionen frühzeitig gestoppt werden. Die einfachste Vorgehensweise, mit der dies erreicht werden kann, ist, die Eigenschaft "kanonisch" ausschließlich an der *äußeren* Struktur eines Terms festzumachen, also zum Beispiel eine λ-Abstraktion $\lambda x\,.\,t$ grundsätzlich als kanonischen Term zu deklarieren – unabhängig davon, ob der Teilterm t selbst kanonisch ist.[18] Der Auswerungsmechanismus, der sich hieraus ergibt, heißt in der Fachsprache *lässige Auswertung* (*lazy evaluation*): man reduziere die Hauptargumente eines Terms, bis sie in kanonischer Form sind, ersetzte das entstehende Redex durch sein Kontraktum und verfahre dann weiter wie zuvor, bis man eine kanonische Form erreicht hat.

Entwurfsprinzip 3.2.9 (Lazy Evaluation)

Die Semantik der Typentheorie basiert auf Lazy Evaluation

Wir wollen nun das bisher gesagte durch entsprechende Definitionen präzisieren.

Definition 3.2.10 (Werte)

Ein Term heißt kanonisch, wenn sein Operatorname in der Operatorentabelle als werteproduzierend *(kanonisch) gekennzeichnet wurde. Ansonsten heißt er nichtkanonisch.*

Ein geschlossener kanonischer Term wird als Wert[19] *bezeichnet.*

[18] Dies ist bei genauerem Hinsehen nicht weniger natürlich als ein Wertebegriff, der eine maximal mögliche Reduktion verlangt. Der wesentliche Aspekt eines Terms $\lambda x\,.\,t$ ist, daß er eine Funktion beschreibt. Die genaue Bedeutung dieser Funktion kann ohnehin erst erfaßt werden, wenn man sie auf einem Eingabeargument auswertet. Hierbei macht es keinen Unterschied, ob der Term t zuvor schon soweit wir möglich reduziert war, da man ohnehin mindestens einen Auswertungsschritt vornehmen muß. Zugegebenermaßen ist aber die reduzierte Form etwas leichter lesbar. Da man diese nicht immer erreichen kann, müssen wir hierauf allerdings verzichten.

[19] Das Wort *Wert* (englisch *value*) ist an dieser Stelle etwas irreführend, da die meisten Menschen ein anderes Verständnis von diesem Begriff haben. Das Problem liegt darin begründet, daß Menschen normalerweise keinen Unterschied machen zwischen einem semantischen Konzept und dem Text, der zur Beschreibung dieses Konzeptes benötigt wird. Der Wert im Sinne von Definition 3.2.10 ist ein "Beschreibungswert", also genau besehen nur ein Stück Text. Es ist daher immer noch möglich, daß verschiedene solcher Werte dasselbe semantische Objekt beschreiben, also *semantisch gleich* im Sinne von Definition 3.2.15 sind.

Warum ist es sinnvoll, in dieser scheinbar so verwirrenden Weise die Grundlagen einer formalen Theorie aufzubauen? Gibt es nicht immer nur *einen* Beschreibungswert für ein Objekt? Das Beispiel der reellen Zahlen zeigt, daß dies im Normalfall nicht mehr garantiert werden kann. Denn unter allen Beschreibungen einer irrationalen Zahl durch Folgen rationaler Zahlen kann man keine fixieren, die eine gute "Standard"-Beschreibung ist in die alle anderen Beschreibungen reeller Zahlen umgewandelt werden können. Selbst in den Einzelfällen, in denen dies möglich wäre, ist der Berechnungsaufwand für die Standardisierung zu hoch. In einer formalen Theorie aber müssen (Beschreibungs-)Werte für Objekte in endlicher Zeit berechnet und nicht nur abstrakt erklärt werden können.

Technisch ist Lazy Evaluation der einzig sinnvolle Weg, dies zu realisieren. Über Auswertung werden die Beschreibungswerte bestimmt, die man nicht weiter vereinfachen kann. Auf diese Werte kann man dann zurückgreifen, wenn man auf formale Weise die Semantik von Termen und Urteilen – also den Bezug zwischen einer Beschreibung und der Realität – festlegen will. Diese Semantik wird zwangsläufig etwas komplizierter ausfallen, da sie unter anderem präzisieren muß, wann zwei Werte dasselbe Objekt bezeichnen sollen, und somit die syntaktischen Restriktionen der lässigen Auswertung wieder aufhebt.

Wie die Semantik wird auch das Inferenzsystem der Typentheorie *nicht* den Einschränkungen der lässigen Auswertung unterliegen, da es – im Gegensatz zur Auswertung von Termen – nicht durch eine feste Strategie gesteuert werden muß.

Algorithmus $\boxed{\texttt{EVAL}(t)}$: *(t: beliebiger typentheoretischer Term)*

- Falls t in kanonischer Form ist, gebe t als Wert aus.
- Falls t nichtkanonisch ist, bestimme $s_1{:=}\texttt{EVAL}(t_1), \ldots, s_n{:=}\texttt{EVAL}(t_n)$, wobei die t_i die Hauptargumente von t sind. Ersetze in t die Argumente t_i durch ihre Werte s_i und nenne das Resultat s
  - Falls s ein Redex ist und u das zugehörige Kontraktum, so gebe $\texttt{EVAL}(u)$ als Wert aus.
  - Andernfalls stoppe die Reduktion ohne Ergebnis: t besitzt keinen Wert.

Abbildung 3.3: Die Auswertungsprozedur der Typentheorie

Die Definition des Begriffs *Wert* hängt also ausschließlich von der äußeren Struktur eines Terms – d.h. von seinem Operatornamen – ab. Diese Konzeption hat den Vorteil, daß Werte sehr leicht, nämlich durch Nachschlagen des Operatornamens in der Operatorentabelle, also solche identifiziert werden können. Natürlich muß man hierzu die Operatorentabelle, die wir in Abbildung 3.2 gegeben haben, um eine entsprechende Unterteilung erweitern. Wir wollen dies tun, nachdem wir die Semantik nichtkanonischer Terme erklärt haben.

Definition 3.2.11 (Reduktion)

Die prinzipiellen Argumente (Hauptargumente) eines nichtkanonischen Terms t sind diejenigen Teilterme, die in der Operatorentabelle als solche gekennzeichnet wurden.

Ein Redex ist ein nichtkanonischer Term t, dessen Hauptargumente in kanonischer Form sind und der in der Redex–Kontrakta Tabelle als Redex erscheint. Das Kontraktum von t ist der entsprechend zugehörige Eintrag in derselben Tabelle.

Ein Term t heißt β-reduzierbar auf u – im Zeichen $t \xrightarrow{\beta} u$ –, wenn t ein Redex und u das zugehörige Kontraktum ist.

Definition 3.2.12 (Semantik typentheoretischer Terme)

Der Wert eines geschlossenen Terms t ist derjenige Wert, der sich durch Anwendung der Auswertungsprozedur EVAL *aus Abbildung 3.3 auf t ergibt.*

Ist u der Wert des Terms t, so schreiben wir auch $t \xrightarrow{!} u$.

Wir wollen die Auswertung von Termen an einigen Beispielen illustrieren.

Beispiel 3.2.13

Abbildung 3.4 zeigt die Einträge der Operatorentabelle für Variablen, Funktionenraum, Produktraum und Summe, die wir um eine Klassifizierung in kanonische und nichtkanonische Terme erweitert haben. Die Hauptargumente nichtkanonischer Terme haben wir dabei durch eine Umrahmung gekennzeichnet.

kanonisch (Typen)	**kanonisch** (Elemente)	**nichtkanonisch**
	$\mathsf{var}\{x{:}\mathtt{v}\}()$ x	
$\mathsf{fun}\{\}(S;\, x.T)$ $x{:}S{\to}T$	$\mathsf{lam}\{\}(x.t)$ $\lambda x.t$	$\mathsf{apply}\{\}(\boxed{f};t)$ $\boxed{f}\ t$
$\mathsf{prod}\{\}(S;\, x.T)$ $x{:}S{\times}T$	$\mathsf{pair}\{\}(s;t)$ $\langle s,t\rangle$	$\mathsf{spread}\{\}(\boxed{e};\, x,y.u)$ $\mathsf{let}\ \langle x,y\rangle = \boxed{e}\ \mathsf{in}\ u$
$\mathsf{union}\{\}(S;T)$ $S{+}T$	$\mathsf{inl}\{\}(s),\quad \mathsf{inr}\{\}(t)$ $\mathsf{inl}(s),\quad \mathsf{inr}(t)$	$\mathsf{decide}\{\}(\boxed{e};\, x.u;\, y.v)$ $\mathsf{case}\ \boxed{e}\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v$

Abbildung 3.4: Vollständige Operatorentabelle für Funktionenraum, Produktraum und Summe

Redex		Kontraktum
$(\lambda x\,.\,u)\ t$	$\xrightarrow{\beta}$	$u[t/x]$
$\mathsf{let}\ \langle x\,,y\rangle = \langle s\,,t\rangle\ \mathsf{in}\ u$	$\xrightarrow{\beta}$	$u[s,t\,/\,x,y]$
$\mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v$	$\xrightarrow{\beta}$	$u[s/x]$
$\mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v$	$\xrightarrow{\beta}$	$v[t/y]$

Abbildung 3.5: Redex–Kontrakta Tabelle für Funktionenraum, Produktraum und Summe

Der besseren Lesbarkeit wegen wurde ein Sortierung in Typen und Elemente vorgenommen, die aber nur informativen Charakter hat, und die Display Form unter die Abstraktionsform geschrieben. So ist zum Beispiel jedes Tupel $\langle s\,,t\rangle$ kanonisch und gehört zu einem (kanonischen) Typ $x\!:\!S\times T$. Die Operation $\mathsf{let}\ \langle x\,,y\rangle = e\ \mathsf{in}\ u$ ist die zugehörige nichtkanonische Form, deren Hauptargument e ist.

Die zugehörigen Einträge der Redex–Kontrakta Tabelle sind in Abbildung 3.5 wiedergegeben. Entsprechend dieser Einträge liefert `EVAL` für einige Beispielterme die folgenden Resultate.

`(λx.inl(x)) y`	$\xrightarrow{l}$	`inl(y)`	
`inl((λx.x) y)`	$\xrightarrow{l}$	`inl((λx.x) y)`	
`(λx.inl(x)) (λy.y) z`	$\xrightarrow{l}$	`inl((λy.y) z)`	
`(λx.inl(x)) ( (λy.y y) (λy.y y) )`	$\xrightarrow{l}$	`inl((λy.y y) (λy.y y))`	
`let ⟨f,b⟩ = ⟨λx.inl(x),y⟩ in f b`	$\xrightarrow{l}$	`inl(y)`	(via `(λx.inl(x)) y`)
`let ⟨f,b⟩ = inl(x)⟨λx.inl(x),y⟩ in f b`	$\xrightarrow{l}$		(kein Wert)

Man beachte, daß wegen der lässigen Auswertung der Wert eines Terms durchaus ein Term sein kann, welcher einen nichtreduzierenden Teilterm enthält.

Für Funktionen läßt sich die in der Prozedur `EVAL` enthaltene lässige Reduktionsrelation mit der in Definition 2.3.11 (Seite 51) gegebenen *strikten Reduktion* vergleichen. Der Unterschied besteht darin, daß zugunsten der Terminierungsgarantie auf ξ- und μ-Reduktion verzichtet wurde. Die anderen Vorschriften zur Auflösung der Termstruktur sind nach wie vor in Kraft.

Aufgrund der Verwendung lässiger Auswertung von Termen können wir die Gleichheit zweier Terme nicht mehr so einfach definieren wie in Definition 2.3.24 (Seite 58), in der es ausreichte, daß die beiden Terme zu demselben Term reduzieren. Lässige Auswertung hat zur Folge, daß die Terme `λx.5` und `λx.3+2` bereits einen Wert darstellen und *nicht* auf denselben Wert reduziert werden, obwohl sie offensichtlich gleich sind. Wir müssen daher die semantische Gleichheit von der Normalform entkoppeln und als Eigenschaft betrachten, die semantisch durch ein *Urteil* zu definieren ist.

3.2.2.2 Urteile

Urteile sind die semantischen Grundbausteine, mit denen Eigenschaften von Termen und die Beziehungen zwischen Termen formuliert werden. Sie legen fest, welche Grundeigenschaften innerhalb einer Theorie *gültig* sein sollen. In einem Beweis bilden sie die *Behauptung*, die als wahr nachzuweisen ist. Sie selbst sind aber kein Bestandteil der formalen Sprache, da *über* ein Urteil – also die Frage *ob* es gültig ist oder nicht – nicht geschlossen werden kann.[20] In der Prädikatenlogik war das Grundurteil eine Aussage, die durch Formeln beschrieben werden konnte. Im λ-Kalkül war es die Gleichheit von λ-Termen und in der einfachen Typentheorie die Typisierung und die Eigenschaft, Typausdruck zu sein. All diese Urteile müssen ein Gegenstück in den

[20] Allerdings werden Beweise und Inferenzregeln der Theorie so ausgelegt werden, daß sie die Bedeutung der Urteile respektieren. So wird bei einer groben Betrachtung kaum ein Unterschied zwischen Urteilen und Ausdrücken der formalen Sprache zu sehen sein. Der Unterschied ist subtil: eine logische Formel kann zu wahr oder zu falsch ausgewertet werden, da sie nur ein Ausdruck ist; ein Urteil dagegen legt fest, *wann* etwas wahr ist, und kann daher nicht ausgewertet werden, ohne die Theorie zu verlassen.

Urteilen der Typentheorie finden, wobei wir Formeln durch Typen ausdrücken, deren Elemente die Beweise beschreiben. Deshalb benötigen wir in der Typentheorie mehrere Arten von Urteilen.

Definition 3.2.14 (Urteile)

Ein Urteil in der Typentheorie hat eine der folgenden vier Gestalten.

- $\underline{T\ \mathrm{Typ}}$*: Der Term* T *ist ein Ausdruck für einen Typ (Menge).* *(Typ-Sein)*
- $\underline{S{=}T}$*: Die Terme* S *und* T *beschreiben denselben Typ.* *(Typgleichheit)*
- $\underline{t{\in}T}$*: Der Term* t *beschreibt ein Element des durch* T *charakterisierten Typs.* *(Typzugehörigkeit)*
- $\underline{s{=}t{\in}T}$*: Die Terme* s *und* t *beschreiben dasselbe Element des durch* T *charakterisierten Typs.* *(Elementgleichheit)*

Was soll nun die Bedeutung dieser Urteile sein? In unserer Diskussion zu Beginn von Abschnitt 2.4 hatten wir bereits festgelegt, daß ein Typ definiert ist durch seine kanonischen Elemente und die zulässigen Operationen auf seinen Elementen. Wir hatten ebenfalls angedeutet, daß die Gleichheit von Elementen von dem Typ abhängt, zu dem sie gehören. Somit muß die Eigenschaft, ein Typ zu sein, die Begriffe der Typzugehörigkeit und der typisierten Gleichheit mit enthalten. Zwei Typen können nur dann gleich sein, wenn sie dieselben Elemente enthalten und dieselbe Form von Gleichheit induzieren. Diese Bedingung ist allerdings nur notwendig und nicht unbedingt hinreichend, denn der strukturelle Aufbau spielt ebenfalls eine Rolle. Ebenso klar ist, daß die Gleichheitsurteile Äquivalenzrelationen auf Termen sein müssen und die Semantik von Ausdrücken zu respektieren haben: ein nichtkanonischer Term t muß genauso beurteilt werden wie der Wert, auf den er reduziert. Diese Überlegungen führen zu der folgenden allgemeinen Vorgehensweise, die bei einer präzisen Definition der Bedeutung von Urteilen befolgt werden sollte.

- $\underline{T\ \mathrm{Typ}}$: Der Begriff des Typ-Seins ist an kanonischen Termen festzumachen und für nichtkanonische Ausdrücke entsprechend an ihren Wert zu binden.

  Unter den kanonischen Ausdrücken werden bestimmte Ausdrücke als kanonische Typausdrücke gekennzeichnet, wobei einerseits ihre Syntax – also zum Beispiel $x{:}S{\to}T$ – eine Rolle spielt, andererseits aber auch gewisse semantische Vorbedingungen gestellt werden müssen – wie zum Beispiel, daß S und T selbst Typen sind und $T[s/x]$ und $T[s'/x]$ für gleiche Werte s, s' aus S auch denselben Typ bezeichnen.

  Ein Ausdruck T beschreibt nun genau dann einen Typ, wenn er auf einen kanonischen Typausdruck reduziert.

- $\underline{S{=}T}$: Auch die Typgleichheit muß zunächst relativ zu kanonischen Typen definiert werden. Dies erfordert allerdings eine komplexe induktive Definition, die von der Struktur der Typausdrücke abhängt.

  $S{=}T$ gilt, wenn S und T auf denselben kanonischen Typ reduzieren.

- $\underline{t{\in}T}$ gilt, wenn T ein Typ ist und t zu einem kanonischen Element von T reduziert, was wiederum induktiv zu definieren ist.

- $\underline{s{=}t{\in}T}$ gilt, wenn s und t zu demselben kanonischen Element von T reduzieren, was ebenfalls eine induktive Definition erfordert.

Es ist leicht zu sehen, daß die Typgleichheit das Typ-Sein subsumiert, da zwei Typen nur gleich sein können, wenn sie überhaupt Typen darstellen. Aus dem gleichen Grunde stellt die Elementgleichheit eine Verallgemeinerung der Typzugehörigkeit dar. Wir können die Bedeutung der Urteile also im wesentlichen dadurch erklären, daß wir die Gleichheit kanonischer Terme in der Form von Tabelleneinträgen fixieren und die anderen Konzepte hierauf abstützen. Dies macht natürlich nur Sinn für geschlossene Terme, da ansonsten kein Wert bestimmt werden kann.

Definition 3.2.15 (Bedeutung von Urteilen)

Die Semantik konkreter typentheoretischer Urteile ist für geschlossene Terme s, t, S und T induktiv wie folgt definiert.

- *T Typ gilt genau dann, wenn $T{=}T$ gilt.*
- *$S{=}T$ gilt genau dann, wenn es kanonische Terme S' und T' gibt, deren Typgleichheit $S'{=}T'$ aus einem der Einträge in der Typsemantiktabelle folgt und für die $S \xrightarrow{l} S'$ und $T \xrightarrow{l} T'$ gilt*
- *$t \in T$ gilt genau dann, wenn $t{=}t \in T$ gilt.*
- *$s{=}t \in T$ gilt genau dann, wenn es kanonische Terme s', t' und T' gibt, deren Elementgleichheit $s'{=}t' \in T'$ aus einem der Einträge in der Elementsemantiktabelle folgt und für die $s \xrightarrow{l} s'$, $t \xrightarrow{l} t'$ und $T{=}T'$ gilt.*

Man beachte hierbei, daß sich die einzelnen Definitionen der Urteile gegenseitig beeinflussen. Dies wird besonders deutlich in den Semantiktabellen, innerhalb derer eine Vielfalt von Querbezügen hergestellt wird. Die Bestimmung der Semantik ist somit ein ständiges Wechselspiel von Reduktion und der Konsultation von Tabelleneinträgen. Die Einträge in der Tabelle wiederum richten sich nach dem intuitiven Verständnis der entsprechenden Konzepte, das nur genügend präzisiert werden muß, wobei man der syntaktischen Struktur der Terme folgt. Wir wollen dies am Beispiel von Funktionenraum, Produktraum und Summe illustrieren.

Beispiel 3.2.16

Zwei Funktionenräume $S_1 \to T_1$ und $S_2 \to T_2$ sind gleich, wenn die Argumentebereiche S_1 und S_2 und die Wertebereiche T_1 und T_2 gleich sind. Verallgemeinert man dies auf abhängige Funktionenräume $x_1{:}S_1 \to T_1$ und $x_2{:}S_2 \to T_2$, so kommt zusätzlich hinzu, daß T_1 und T_2 auch dann gleich bleiben, wenn sie durch gleichwertige (aber syntaktisch verschiedene) Elemente von S_1 (bzw. S_2) beeinflußt werden. Abhängige und unabhängige Funktionenräume sind kompatibel in dem Sinne, daß $S_1 \to T_1$ identisch ist mit $x_1{:}S_1 \to T_1$, wobei x_1 eine beliebige Variable ist.[21] Für diese Definition, die in der Typsemantiktabelle in Abbildung 3.6 präzisiert wird, benötigt man die Typgleichheit der Teilterme und die Gleichheit von Elementen aus S_1, was zu einer rekursiven Verwendung von Definition 3.2.15 führt.

Zwei Funktionen $\lambda x_1.t_1$ und $\lambda x_2.t_2$ sind gleich in $x{:}S \to T$, falls $x{:}S \to T$ überhaupt ein Typ ist (was keineswegs sicher ist) und die Funktionskörper t_1 bzw. t_2 sich auf gleichen Eingaben aus S auch in T gleich verhalten.

Abbildung 3.6 stellt diese Urteile und die entsprechenden Urteile für Produkträume und disjunkte Vereinigungen zusammen.

Man beachte, daß die Semantik der Gleichheitsurteile – im Gegensatz zur Auswertung – *keine lässige Auswertung* zuläßt, sondern durch einen ständigen Wechsel von Reduktion und Tabellenverwendung einen Term bis in das letzte Detail analysiert. Aus der Definition 3.2.15 und den bisher angegebenen Semantiktabellen lassen sich folgende Eigenschaften der semantischen Urteile ableiten.

Lemma 3.2.17

1. *Typgleichheit $S{=}T$ ist transitiv und symmetrisch (aber nicht reflexiv)*
2. *Für alle Terme S und T gilt $S{=}T$ genau dann, wenn es einen Term S' gibt mit $S \xrightarrow{l} S'$ und $S'{=}T$.*
3. *Elementgleichheit $s{=}t \in T$ ist transitiv und symmetrisch (aber nicht reflexiv) in s und t.*
4. *Für alle Terme s, t und T gilt $s{=}t \in T$ genau dann, wenn es einen Term s' gibt mit $s \xrightarrow{l} s'$ und $s'{=}t \in T$.*
5. *Für alle Terme s, t und T folgt aus $s{=}t \in T$ immer, daß T ein Typ ($T{=}T$) ist.*
6. *Für alle Terme s, t, S und T folgt aus $s{=}t \in T$ und $S{=}T$, daß $s{=}t \in S$ gilt.*

[21] Da $S_1 \to T_1$ geschlossen sein muß, kann x_1 in T_1 nicht frei vorkommen und es entstehen keinerlei Probleme.

Typsemantik		
$x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2$	falls	$S_1{=}S_2$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2]$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\to}T_2$	falls	$T = x_2{:}S_2{\to}T_2$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\to}T_1 = T$	falls	$x_1{:}S_1{\to}T_1 = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2$	falls	$S_1{=}S_2$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2]$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\times}T_2$	falls	$T = x_2{:}S_2{\times}T_2$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\times}T_1 = T$	falls	$x_1{:}S_1{\times}T_1 = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$S_1{+}T_1 = S_2{+}T_2$	falls	$S_1{=}S_2$ und $T_1{=}T_2$.
Elementsemantik		
$\lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$	falls	$x{:}S{\to}T$ Typ und $t_1[s_1/x_1] = t_2[s_2/x_2] \in T[s_1/x]$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S$.
$\langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S{\times}T$	falls	$x{:}S{\times}T$ Typ und $s_1{=}s_2 \in S$ und $t_1{=}t_2 \in T[s_1/x]$.
$\mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S{+}T$	falls	$S{+}T$ Typ und $s_1{=}s_2 \in S$.
$\mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S{+}T$	falls	$S{+}T$ Typ und $t_1{=}t_2 \in T$.

Abbildung 3.6: Semantik der Urteile für Funktionenraum, Produktraum und Summe

Darüber hinaus kann man auch noch zeigen, daß die Semantik der Urteile die Reduktion auch dann respektiert, wenn sie sich nicht an lässige Auswertung hält. Wenn zwei Terme sich durch beliebige Reduktionen auf denselben Term reduzieren lassen (also *berechnungsmäßig äquivalent* sind), dann gilt Typgleichheit, falls einer von ihnen ein Typ ist, und Elementgleichheit in einem Typ T, falls einer von ihnen zu T gehört.

Diese Aussagen lassen sich unter Verwendung der konkreten Semantiktabellen beweisen und gelten somit zunächst nur für Funktionenraum, Produktraum und disjunkte Vereinigung. Die Erweiterungen der Semantiktabellen, die wir in den folgenden Abschnitten vornehmen werden, werden allerdings so ausgelegt, daß Lemma 3.2.17 und die obige Aussage weiterhin Gültigkeit behalten werden.

3.2.2.3 Hypothetische Urteile

Um ein Argumentieren unter vorgegebenen *Annahmen* zu ermöglichen, ist es nötig, das Konzept des Urteils auf *hypothetische Urteile* zu erweitern. Diese erlauben es, zu beschreiben, daß ein bestimmtes Urteil wahr ist, falls bestimmte Voraussetzungen erfüllt sind. In ihrer textlichen Beschreibungsform sind hypothetische Urteile sehr ähnlich zu Sequenzen:

$$A_1,\ldots,A_n \vdash J$$

Dies soll ausdrücken, daß das Urteil (judgment) J aus den Annahmen A_i folgt. Man beachte jedoch, daß im Unterschied zu einer Sequenz ein hypothetisches Urteil ein semantisches Konzept ist. Innerhalb der Typentheorie beschränken wir uns auf die vier in Definition 3.2.14 festgelegten Arten von Urteilen und auf Annahmen der Art $x_i{:}T_i$ – also auf Typdeklarationen. Das hypothetische Urteil

$$\underline{x_1{:}T_1,\ldots,x_n{:}T_n \vdash J}$$

ist also zu lesen als

Unter der Annahme, daß x_i Variablen vom Typ T_i sind, gilt das Urteil J.

Eine präzise Definition hypothetischer Urteile muß *Funktionalität* und *Extensionalität* garantieren. In erster Näherung ist dies die Eigenschaft, daß $J[t_i/x_i]$ für alle kanonischen Elemente $t_i \in T_i$ gelten muß, und daß für alle (auch nichtkanonischen) Terme t_i und t_i' folgt, daß $J[t_i/x_i]$ und $J[t_i'/x_i]$ gleich beurteilt werden, falls $t_i{=}t_i' \in T_i$ gilt.

Auf eine genaue Erklärung dieser Eigenschaften wollen wir an dieser Stelle verzichten. Interessierte können Details finden in [Constable *et.al.*, 1986, Seite 141] und [Martin-Löf, 1984, Seite 16ff].

3.2.3 Inferenzsystem

Durch Reduktion und Urteile ist die Semantik von Ausdrücken der Typentheorie eindeutig definiert. Wir wollen nun beschreiben, wie man diese Semantik durch rein syntaktische Mittel innerhalb eines Inferenzsystems für die Typentheorie so simulieren kann, daß ein formales logisches Schließen über alle Bestandteile der Theorie möglich ist. Wir müssen hierzu im wesentlichen eine syntaktische Repräsentation der (hypothetischen) Urteile angeben und Inferenzregeln entwerfen, die den Einträgen in der Semantiktabelle entsprechen.[22]

Entsprechend der textlichen Beschreibungsform hypothetischer Urteile sind Sequenzen das angemessenste Mittel einer syntaktischen Repräsentation innerhalb eines Kalküls.[23] Die bisherige Gestalt von Sequenzen war

$$x_1{:}T_1, \ldots, x_n{:}T_n \vdash C,$$

wobei die x_i Variablen und die T_i Ausdrücke sind und durch die Anforderungen an den Kalkül sichergestellt war, daß die T_i Typen darstellten. Welche Gestalt aber soll nun die Konklusion C – das Gegenstück zu den einfachen Urteilen – besitzen? Wir möchten hierfür nur ein einziges syntaktisches Konzept verwenden, müssen aber insgesamt vier Arten von Urteilen repräsentieren. Wir wollen daher vor einer formalen Definition von Sequenzen diskutieren, auf welche Art wir diese Anforderung erreichen können und welche Auswirkungen dies auf den Kalkül der Typentheorie hat.

3.2.3.1 Universen: syntaktische Repräsentation der Typ-Eigenschaft

Für die syntaktische Repräsentation der Typeigenschaft hatten wir bereits im Zusammenhang mit der einfachen Typentheorie ein *Typuniversum* eingeführt, das wir mit dem Symbol $\mathbb{U}$ gekennzeichnet hatten. Unter Verwendung dieses Symbols können wir das Urteil 'T *Typ*' durch '$T \in \mathbb{U}$' darstellen und '$S{=}T$' durch '$S{=}T \in \mathbb{U}$'. Damit hätten wir Typ-Sein und Typgleichheit auf die Urteile Typzugehörigkeit und Elementgleichheit zurückgeführt und brauchen uns nur noch um eine einheitliche Darstellung dieser beiden Urteile zu bemühen.

Welche Rolle aber spielt $\mathbb{U}$ in unserer formalen Theorie? Offensichtlich muß es ein Ausdruck der formalen Sprache sein und sein Verwendungszweck suggeriert, daß es einen Typausdruck kennzeichnet. Dies würde bedeuten, daß $\mathbb{U}$ als ein Typausdruck selbst zum Universum aller Typen gehören muß. Damit müßte das Urteil $\mathbb{U} \in \mathbb{U}$ gültig sein. Können wir dies nun vermeiden? Das folgende Beispiel zeigt, daß $\mathbb{U}$ in der Tat nicht nur aus formalen Gründen ein Typ sein muß.

Beispiel 3.2.18

In Beispiel 3.1.1 auf Seite 97 hatten wir die Notwendigkeit abhängiger Datentypen für eine hinreichend ausdrucksstarke Theorie begründet. Abhängigkeit bedeutet dabei zum Beispiel für einen Funktionenraum $x{:}S{\to}T$, daß der Datentyp T eine freie Variable x enthalten muß, die mit Werten aus S belegt werden darf. Die einzige Möglichkeit, dies syntaktisch korrekt zu beschreiben, ist die Verwendung einer Funktion $\hat{T}$, die bei Eingabe eines Wertes $s \in S$ genau auf den Wert $T[s/x]$ reduziert, was ein Typ – also ein Element von $\mathbb{U}$ – sein muß. $\hat{T}$ *muß* somit (mindestens) den Typ $S{\to}\mathbb{U}$ besitzen.

Der Ausdruck $\mathbb{U}$ wird also als Bestandteil von Typaudrücken vorkommen müssen und wenn wir kein neues Sonderkonzept zusätzlich zu Typen und Elementen[24] einführen wollen, dann müssen wir $\mathbb{U}$ selbst als Typaus-

[22] Für das intuitive Verständnis der Typentheorie ist dieser Teilabschnitt vielleicht der bedeutendste aber zugleich auch der schwierigste, da eine Reihe scheinbar willkürlicher "Entwurfsentscheidungen" getroffen werden müssen, deren Tragweite erst bei einer gewissen Erfahrung im Umgang mit dieser Theorie deutlich wird. Einen theoretisch zwingenden Grund, das Inferenzsystem in der hier vorgestellten Form auszulegen, gibt es nicht. Es ist eher das Resultat langjähriger Erfahrungen mit der Automatisierung einer mathematisch formalen Grundlagentheorie, bei denen die Vor- und Nachteile verschiedener Möglichkeiten sorgsam gegeneinander abgewägt werden mußten. Oft zeigten auch die Experimente, daß mancher Weg, der theoretisch so einfach erschien, zu unlösbaren praktischen Problemen führte. Daher werden manche Entscheidungen erst im Rückblick einsichtig.

[23] Die formale Ähnlichkeit ist in der Tat so groß, daß Sequenzen und hypothetische Urteile oft miteinander verwechselt werden. Der Unterschied besteht darin, daß die Bestandteile eines hypothetischen Urteils tatsächlich Annahmen (Deklarationen) und Urteile sind, während eine Sequenz eigentlich nur ein Stück Text ist.

[24] Diesen Weg hat Girard [Girard, 1971, Girard, 1986, Girard *et.al.*, 1989] in seinem System $\mathcal{F}$ eingeschlagen, um die ursprünglichen Paradoxien der frühen Martin-Löf'schen Ansätze [Martin-Löf, 1970] zu umgehen.

druck zulasssen. Das Russelsche Paradoxon der Mengentheorie wird hiervon nicht berührt, da es sich bei $\mathbb{U}$ um eine ganz spezielle "maximale" Klasse handelt, die ein Element von sich selbst ist, und diese Eigenschaft keineswegs für andere Mengen (Typen) erlaubt wird.

Die Wahl eines Typuniversums $\mathbb{U}$, das selbst als Typ verwandt werden darf, würde die Theorie übrigens auch sehr vereinfachen, da in diesem Fall viele der von uns gewünschten Typkonstrukte durch einen abhängigen Funktionenraum simuliert und damit als definitorische Abkürzung eingeführt werden könnten.[25] So könnten wir durch das Typuniversum $\mathbb{U}$ auf einfache Weise einen Kalkül höherer Ordnung einführen, der extrem ausdrucksstark ist. Leider ermöglicht die Kombination von abhängigen Typen und dem Axiom $\mathbb{U}\in\mathbb{U}$ jedoch ein anderes Paradoxon, das zwar sehr viel komplizierter ist als das Russelsche Paradoxon, aber dennoch die Widersprüchlichkeit einer solchen Theorie aufdeckt. Mit diesem Paradoxon (*Girard's Paradoxon*, entdeckt 1970) kann man den folgenden allgemeinen Satz nachweisen.

Satz 3.2.19

Jede Theorie, welche den abhängigen Funktionenraum $x{:}S{\to}T$ enthält und $\mathbb{U}\in\mathbb{U}$ zuläßt, ist inkonsistent.

Wir müssen also nach einem anderen Weg suchen, die Typeigenschaft des Typuniversums $\mathbb{U}$, um die wir aus den oben erwähnten Gründen nicht herumkommen, syntaktisch zu repräsentieren.

Die wohl einfachste Lösung für dieses Problem ergibt sich, wenn wir versuchen, die Natur des Universums $\mathbb{U}$ zu charakterisieren. Es ist gleichzeitig eine Menge von Typen und selbst ein Typ. Als Typ aber steht $\mathbb{U}$ in irgendeinem Sinne auf einer höheren Stufe als diejenigen Typen, die Elemente von $\mathbb{U}$ sind. Das Universum, zu dem $\mathbb{U}$ gehören muß, ist also nicht $\mathbb{U}$ selbst, sondern ein höher geartetes Universum, welches man zum Beispiel mit $\mathbb{U}_2$ bezeichnen kann. $\mathbb{U}$ selbst ist also die Menge aller "kleinen" Typen und selbst ein "großer" Typ. Diesen Gedanken muß man natürlich weiterführen, denn auch $\mathbb{U}_2$ muß wiederum ein Typ eines höheren Universums sein. So erhalten wir schließlich eine ganze Hierarchie von Universen $\mathbb{U}{=}\mathbb{U}_1$, $\mathbb{U}_2$, $\mathbb{U}_3$, ..., die das syntaktische Gegenstück zur Typeigenschaft bilden. Diese Universen dürfen allerdings nicht völlig getrennt voneinander erscheinen, wie das folgende Beispiel zeigt.

Beispiel 3.2.20

Im Beispiel 3.2.18 hatten wir erklärt, daß der Typ T im abhängigen Funktionenraum $x{:}S{\to}T$ im wesentlichen durch eine Funktion $\hat{T}$ gebildet werden kann, die vom Typ $S{\to}\mathbb{U}$ ist. Dabei ist S ein einfacher Typ aus $\mathbb{U}_1$ und $\mathbb{U}$ ein großer Typ aus $\mathbb{U}_2$.

Zu welchem Universum gehört nun $S{\to}\mathbb{U}$? Gemäß unseren bisherigen Regeln ist es das Universum, zu dem sowohl S als auch $\mathbb{U}$ gehören. Da S und $\mathbb{U}$ aber zu verschiedenen Universen gehören, müssen wir entweder die Funktioneraumbildung dahingehend abändern, daß wir die Universen der Teiltypen und des Gesamttyps als Indizes mitschleppen, oder zulassen, daß alle Elemente von $\mathbb{U}_1$ auch Elemente von $\mathbb{U}_2$ sind. Da die erste Alternative zu so komplizierten Konstrukten wie $S\overset{1,2}{\to}_2\mathbb{U}$ und einer Vielfalt nahezu identischer Regeln führen würde, erscheint die zweite Lösung sinnvoller.

Es ist also sinnvoll, die Hierarchie der Universen *kumulativ* zu gestalten: jedes Universum enthält das nächsttieferliegende Universum und alle seine Elemente.

$$\mathbb{U}_1 \overset{\in}{\subset} \mathbb{U}_2 \overset{\in}{\subset} \mathbb{U}_3 \overset{\in}{\subset} \ldots$$ [26]

[25] Als Beispiel seien die folgenden Typsimulationen genannt.

$x{:}S{\times}T$	$\equiv$ X:$\mathbb{U}$ → (x:S→(T→X)) → X	$A\wedge B$	$\equiv \forall$P:$\mathbb{U}$.(A⇒(B⇒P)) ⇒ P
$\forall x{:}T.P$	$\equiv x{:}T{\to}P$	$A\vee B$	$\equiv \forall$P:$\mathbb{U}$. ((A⇒P) ⇒ (B⇒P)) ⇒ P
$\exists x{:}T.P$	$\equiv x{:}T{\times}P$	Λ	$\equiv \forall$P:$\mathbb{U}$.P
$A\Rightarrow B$	$\equiv A{\to}B$	$\neg A$	$\equiv A\Rightarrow\Lambda$
		$s{=}t\in T$	$\equiv \forall$P:$T{\to}\mathbb{U}$. P(s)⇒P(t)

Man kann relativ leicht überprüfen, daß die logischen Gesetze der simulierten Operationen sich tatsächlich aus denen des abhängigen Funktionenraumes ergeben. Auch die Probleme der natürlichen Zahlen innerhalb der einfachen Typentheorie lassen sich durch abhängige Funktionenraüme durch eine leichte Erweiterung der Church-Numerals lösen.

$\overline{n}\equiv \lambda$X.$\lambda$f.$\lambda$x. f^n x	$\mathbb{N}$	$\equiv$ X:$\mathbb{U}$ → (X→X)→X→X
	PRs[*base*,*h*]	$\equiv \lambda$n. n($\mathbb{N}$) *h base*

Mit diesen Definitionen kann die primitive Rekursion problemlos und sinnvoll typisiert werden.

kanonisch		nichtkanonisch
(Typen)	(Elemente)	
$\mathbf{U}\{j:\mathtt{1}\}()$ U_j	(alle kanonischen Typen)	
$\mathbf{equal}\{\}(s;t;T)$ $s = t \in T$	$\mathbf{Axiom}\{\}()$ Axiom	

Abbildung 3.7: Operatorentabelle für Universen und Gleichheit

Gemessen an ihrer Ausdruckskraft ist diese unendliche kumulative Universenhierarchie sehr ähnlich zu der Eigenschaft $\mathtt{U}\in\mathtt{U}$. Sie kann jedoch ihre Probleme vermeiden und ist somit für formale Systeme besser geeignet.

Es sei an dieser Stelle angemerkt, daß für die Universenhierarchie ein sehr wichtiges *Reduktionsprinzip* gilt, welches besagt, daß jedes Objekt eines höheren Typs durch ein Objekt aus $\mathbb{U}$ simuliert werden kann. Prinzipiell könnte man also auf die höheren Universen verzichten und immer die Einbettung in $\mathbb{U}$ vornehmen. Dies würde die Theorie jedoch unnötig verkomplizieren. Wir behalten daher die Universenhierarchie und werden uns die Möglichkeit einer Einbettung für eventuelle Selbstreflektion aufheben.

Entwurfsprinzip 3.2.21

Die Typeigenschaft wird dargestellt durch eine kumulative Hierarchie von Universen.

Um dieses Prinzip zu realisieren, müssen wir also unsere Syntax durch einen weiteren Eintrag in der Operatorentabelle (siehe Abbildung 3.7) ergänzen. Zugunsten einer größeren Flexibilität beim formalen Schließen lassen wir als Indizes neben den konstanten natürlichen Zahlen – welche zu kanonischen Universen führen – auch einfache arithmetische Ausdrücke zu, die aus Zahlen, Zahlenvariablen, Addition und Maximumbildung zusammengesetzt sind. Diese *level-expressions* bilden die erlaubten Parameter des Operators $\mathbf{U}$ in Abbildung 3.7. Spezielle nichtkanonische Formen für Universen gibt es nicht (und somit auch keine Einträge in die Redex–Kontrakta Tabelle).

Die Semantik der Universen (siehe Abbildung 3.8) ist einfach. Zwei Universen U_i und U_j sind als Typen gleich, wenn i und j als Zahlen gleich sind. Die kanonischen Elemente eines Universums sind genau die kanonischen Typen und somit ist die Gleichheit von Elementen aus U_j nahezu identisch mit der Typgleichheit. Die Kumulativität wird semantisch dadurch wiedergespiegelt, daß die Grundtypen wie $\mathbb{Z}$, aus denen jeder kanonische Term im Endeffekt aufgebaut wird, zu jedem Universum gehören werden.

3.2.3.2 Gleichheit als Proposition

Mithilfe der Universenhierarchie haben wir die syntaktische Repräsentation von Typ-Sein und Typgleichheit auf die Urteile Typzugehörigkeit und Elementgleichheit zurückgeführt. Wir wollen nun zeigen, wie wir auch für diese beiden Urteile eine einheitliche Darstellung finden können. Unsere Definition 3.2.15 der Semantik von Urteilen hat die Typzugehörigkeit mehr oder weniger als Spezialfall der Elementgleichheit charakterisiert. Dennoch gibt es Gründe, nicht die Elementgleichheit, sondern die Typzugehörigkeit als formalen Oberbegriff zu wählen und die semantische Gleichheit – ähnlich wie die Typeigenschaft – *innerhalb der Objektsprache* zu simulieren, also einen Gleichheitstyp einzuführen, welcher die semantischen Eigenschaften der Gleichheit syntaktisch wiederspiegelt.

In unserer Diskussion der Urteile hatten wir auf die Notwendigkeit des Gleichheits*urteils* hingewiesen. Unter allen elementaren logischen Aussagen ist die Gleichheit die wichtigste, da Schließen über Gleichheit in nahezu allen Teilen der Mathematik und Programmierung eine essentielle Rolle spielt. Was uns bisher jedoch fehlt, ist eine Möglichkeit, innerhalb von formalen Beweisen über Gleichheit zu *schließen*, da wir in

[26] Diese Idee ist eigentlich schon in der frühen Mathematik entstanden. In '*Principia Mathematicae*', dem Grundlagenbuch der modernen Mathematik [Whitehead & Russell, 1925], findet man eine hierarchische Typstruktur, wenn auch eine recht komplizierte. Sie wurde später durch Quine [Quine, 1963] vereinfacht, was zu einer kumulativen Theorie der Typen führte.

Typsemantik		
$s_1 = t_1 \in T_1 \;=\; s_2 = t_2 \in T_2$	falls	$T_1{=}T_2$ und $s_1{=}s_2 \in T_1$ und $t_1{=}t_2 \in T_1$.
$\mathsf{U}_{j_1} = \mathsf{U}_{j_2}$	falls	$j_1{=}j_2$ *(als natürliche Zahl)*
Elementsemantik		
$\mathsf{Axiom} = \mathsf{Axiom} \in s = t \in T$	falls	$s{=}t \in T$
$x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2] \in \mathsf{U}_j$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\to}T_2 \in \mathsf{U}_j$	falls	$T = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\to}T_1 = T \in \mathsf{U}_j$	falls	$x_1{:}S_1{\to}T_1 = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2] \in \mathsf{U}_j$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\times}T_2 \in \mathsf{U}_j$	falls	$T = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\times}T_1 = T \in \mathsf{U}_j$	falls	$x_1{:}S_1{\times}T_1 = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$S_1{+}T_1 = S_2{+}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1{=}T_2 \in \mathsf{U}_j$.
$s_1 = t_1 \in T_1 \;=\; s_2 = t_2 \in T_2 \in \mathsf{U}_j$	falls	$T_1{=}T_2 \in \mathsf{U}_j$ und $s_1{=}s_2 \in T_1$ und $t_1{=}t_2 \in T_1$.
$\mathsf{U}_{j_1} = \mathsf{U}_{j_2} \in \mathsf{U}_j$	falls	$j_1{=}j_2{<}j$ *(als natürliche Zahl)*

Abbildung 3.8: Semantik der Urteile für Universen und Gleichheit

den Hypothesen einer Sequenz bisher nur (Typ-)Ausdrücke, aber keine Urteile zugelassen haben. Wir stehen daher vor der Wahl, entweder die Struktur der Sequenzen zu erweitern und Urteile und Typausdrücke simultan in den Hypothesen zu verwalten, oder einfach nur die formale Sprache um einen Gleichheitstyp zu ergänzen.

Die Entscheidung für den ersten Weg würde eine Menge formaler Komplikationen mit sich bringen. Der zweite Weg dagegen wirft philosophische Probleme auf: *Gleichheit ist eine logische Aussage und nicht etwa ein Typ*. Konzeptuell ist dieser Unterschied in der Tat von großer Bedeutung. Man sollte allerdings im Auge behalten, daß die Typentheorie das logische Schließen so formalisieren soll, daß es von Computern verarbeitet werden kann. Der Unterschied zwischen Typen und logischen Aussagen verschwindet also spätestens auf der Ebene der Verarbeitung der Symbole, die zur Repräsentation verwendet werden. Somit ist er für die *formale* Theorie von untergeordneter Bedeutung und da eine formale Trennung einen erheblichen Mehraufwand mit sich bringen würde, entscheiden wir uns dafür, logische Aussagen bereits *innerhalb der Theorie* durch Typen zu simulieren. Diese Vorgehensweise, deren Berechtigung durch die Curry-Howard Isomorphie gestützt wird, die wir in Abschnitt 3.3 genauer besprechen, wird in der Literatur als Prinzip der Propositionen als Typen (*propositions as types principle*) bezeichnet

Entwurfsprinzip 3.2.22 (Propositionen als Typen)
In der Typentheorie werden logische Aussagen dargestellt durch Typen, deren Elemente den Beweisen der Aussagen entsprechen.

Für die Darstellung der Gleichheit ergänzen wir unsere formale Sprache also um einen weiteren vordefinierten Term **equal**$\{\}(s;t;T)$ (siehe Abbildung 3.7), der das Gleichheitsurteil $s{=}t \in T$ syntaktisch simuliert. In seiner Display Form werden wir diesem Term daher dieselbe Gestalt $s = t \in T$ geben.[27] Das kanonische Element dieses Typs wird mit $\underline{\mathsf{Axiom}}$ bezeichnet. Alle Beweise (Elemente des Typs) werden im Endeffekt auf Axiom reduzieren und somit Evidenz liefern, daß das Urteil $s{=}t \in T$ tatsächlich Gültigkeit hat. In der Semantiktabelle 3.8 stellen wir genau dieser Zusammenhang zwischen kanonischen Elementen des *Typs* $s = t \in T$ und dem *Urteil* $s{=}t \in T$ her. Auch für die Gleichheit gibt es keine speziellen nichtkanonischen Ausdrücke und dementsprechend keine Einträge in der Redex–Kontrakta Tabelle.

Durch die Verwendung der Gleichheit als Datentyp sind wir nun in der Lage, alle vier Arten von Urteilen durch ein einziges Urteil, nämlich die Typzugehörigkeit, zu simulieren und Gleichheiten auch in den Hypothesen eines Beweises zu verwenden. Welche weiteren Vorteile die Wahl des Typzugehörigkeitsurteils als zentrales Beweiskonzept anstelle der Elementgleichheit mit sich bringt, zeigt der nun folgende Abschnitt.

[27] Man achte auf die feinen Unterschiede in den Zeichensätzen.

3.2.3.3 Entwicklung von Beweisen statt Überprüfung

In unserer Diskussion der Anforderungen an einen praktisch verwendbaren Kalkül im Abschnitt 3.1.3 hatten wir bereits darauf hingewiesen, daß wir Kalküle zum *Entwickeln* formaler Beweise einsetzen wollen und nicht etwa nur zum Überprüfen eines bereits gefundenen Beweises. Da wir nun gemäß dem Prinzip "Propositionen als Typen" logische Aussagen als Typen repräsentieren, deren Elemente genau die Beweise dieser Aussagen sind, ist eine *direkte* Darstellung des Typzugehörigkeitsurteils durch ein syntaktisches Konstrukt der Gestalt $t \in T$ (wobei t und T Ausdrücke sind) wenig geeignet, um dieses Ziel zu erreichen. Wir wollen dies an einem Beispiel erläutern.

Beispiel 3.2.23

Eine direkte syntaktische Darstellung des Urteils $s{=}t \in T$ müßte entsprechend der im vorigen Abschnitt beschriebenen Methodik die Gestalt

$$p \ \in \ s = t \in T$$

haben, wobei p ein Term ist, der im Endeffekt zu Axiom reduzieren muß. Dieser Term hängt aber von dem Beweis ab, der geführt werden muß, um diese konkrete Gleichheit nachzuweisen. Wir müßten also bereits *im voraus* wissen, wie $s{=}t \in T$ zu beweisen ist, um p angeben zu können. Dann aber ist der Kalkül von geringem Nutzen, da wir nur noch überprüfen können, ob unsere Beweisführung stimmt.

Das Problem wird noch größer, wenn wir bedenken, wie viele logische Regeln wir beim Beweis einer logischen Aussage verwenden können. Wir wären gezwungen, den Beweis zunächst von Hand zu führen, aus dem vollständigen Beweis einen Beweisterm zusammenzusetzen und danach den Beweis zu überprüfen, indem wir ihn anhand des Beweisterms noch einmal durchgehen. Die Rechnerunterstützung bei der Beweisführung würde dadurch ihren Sinn verlieren, da sie keinerlei Vorteile gegenüber der Handarbeit mit sich brächte.

Der Sinn einer automatisierbaren formalen Logik ist jedoch nicht, die Korrektheit *bestimmter* Beweise nachzuweisen, sondern ein Hilfsmittel dafür zu bieten, die Gültigkeit einer *Aussage* zu beweisen. Mit anderen Worten: wir wollen den Beweisterm aufbauen *während* wir den Beweis entwickeln und nicht etwa vorher. Um dies zu erreichen, müssen wir den formalen Kalkül, mit dem wir Beweise für Urteile syntaktisch simulieren, so gestalten, daß er die *Entwicklung* eines Terms, der zu einem vorgegebenen Typ gehören soll, unterstützt und nicht nur die Korrektheit eines gegebenen Typzugehörigkeitsurteils nachzuweisen hilft. Dies verlangt eine andere Form der Darstellung eines Urteils. Anstelle von $p \ \in \ s = t \in T$ müssen wir eine Form finden, die es erlaubt, p erst dann niederzuschreiben, wenn der Beweis beendet ist und den Term in unvollständigen Beweisen einfach offenzulassen. Formal werden wir dies durch die Notation

$$s = t \in T \quad [\textbf{ext}\ p]$$

kennzeichnen, die inhaltlich dasselbe Urteil repräsentiert wie zuvor, aber durch die Notation $[\textbf{ext}\ p]$ andeutet, daß wir den Beweisterm p zu verstecken gedenken und aus einem vollständigen Beweis *extrahieren* können. Dies hat keinerlei Einfluß auf die Semantik der Theorie und die theoretische Bedeutung der Inferenzregeln, bewirkt aber einen gravierenden praktischen Unterschied für das Arbeiten mit diesen Regeln innerhalb eines interaktiven Beweissystems für die Typentheorie. Die Anwendung einer Regel scheint nämlich zunächst einmal nur den Typ selbst zu betreffen, dessen Element wir innerhalb des Beweises konstruieren wollen. Daß diese Regel implizit natürlich auch sagt, was der Zusammenhang zwischen den "*Extrakt-Termen*" der Teilziele und dem des ursprünglichen Beweisziels ist, spielt zunächst erst einmal keine Rolle. Erst, wenn der Beweis beendet ist, spielt diese Information eine Rolle, denn sie kann dazu verwendet werden, die jeweiligen Extrakt-Terme automatisch zu konstruieren. Die Entscheidung, einen Kalkül zur interaktiven Entwicklung von Beweisen zu entwerfen, führt also zu dem folgenden Gestaltungsmerkmal der intuitionistischen Typentheorie von NuPRL.

Entwurfsprinzip 3.2.24 (Implizite Darstellung von Elementen)

Urteile der Typentheorie werden dargestellt durch Konklusionen der Form $\underline{T \ [\textbf{ext}\ p]}$.

Es sei angemerkt, daß wir durch diese Entwurfsentscheidung, die den Kalkül von NuPRL deutlich von den meisten anderen Formulierungen der Typentheorie abhebt, keineswegs die Möglichkeit zu einem expliziten Schließen über Elemente eines gegebenen Typs verlieren. Der Datentyp der typabhängigen Gleichheit, den wir oben eingeführt haben, kann genau für diesen Zweck genutzt werden. Somit bietet uns der Kalkül insgesamt sechs grundsätzliche Arten des formalen Schließens innerhalb eines einheitlichen Formalismus.

- Explizites Schließen (Überprüfung eines Urteils):
  - *Typ-Sein* "T Typ": Zeige die Existenz eines Terms p und eines Levels j mit $p \in T = T \in \mathsf{U}_j$.
  - *Typgleichheit* "S=T": Zeige die Existenz eines Terms p und eines Levels j mit $p \in S = T \in \mathsf{U}_j$.
  - *Typzugehörigkeit* "$t \in T$": Zeige die Existenz eines Terms p mit $p \in t = t \in T$.
  - *Elementgleichheit* "s=$t \in T$": Zeige die Existenz eines Terms p mit $p \in s = t \in T$.

  Die zugehörigen Beweisziele lauten $\underline{\vdash T = T \in \mathsf{U}_j}$, $\underline{\vdash S = T \in \mathsf{U}_j}$, $\underline{\vdash t = t \in T}$ bzw. $\underline{\vdash s = t \in T}$.
- Implizites Schließen (Konstruktion von Termen, die ein Urteil erfüllen):
  - *Konstruktion von Datentypen*: Zeige die Existenz eines Terms T mit $T \in \mathsf{U}_j$.
  - *Konstruktion von Elementen* eines Typs T: Zeige die Existenz eines die Terms t mit $t \in T$.
    Diese Form wird benutzt um logische Beweise zu führen (Konstruktion von Beweistermen) und vor allem auch, um Algorithmen zu konstruieren, von denen man nur die Spezifikation gegeben hat.

  Die entsprechenden Beweisziele sind $\underline{\vdash \mathsf{U}_j}$ bzw. $\underline{\vdash T}$.

3.2.3.4 Sequenzen, Regeln und formale Beweise

Nach all diesen Vorüberlegungen sind wir nun in der Lage, präzisen Definition der im Beweiskalkül von NuPRL relevanten Konzepte zu geben. Als syntaktische Repräsentation von hypothetischen Urteilen bilden *Sequenzen* die Grundkonstrukte formaler Beweise. Ihre Gestalt ergibt sich aus dem Entwurfsprinzip 3.2.24.

Definition 3.2.25 (Sequenzen)

1. *Eine* Deklaration *hat die Gestalt* $x{:}T$, *wobei* x *eine Variable und* T *ein Term ist.*
2. *Eine* Hypothesenliste *ist eine Liste* $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$ *von durch Komma getrennten Deklarationen.*
3. *Eine* Konklusion *ist ein Term im Sinne von Definition 3.2.5.*
4. *Eine* Sequenz *(oder* Beweisziel*) hat die Gestalt* $\Gamma \vdash C$, *wobei* Γ *eine Hypothesenliste und* C *eine Konklusion ist.*
5. *Eine Sequenz* $Z = x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$ [28] *ist* geschlossen, *wenn jede in* C *oder einem der* T_i *frei vorkommende Variable* x *zuvor durch* $x{:}T_j$ $(j < i)$ *deklariert wurde.*

   Eine Sequenz ist rein, *wenn sie geschlossen ist und jede Variable nur einmal deklariert wurde.*
6. *Eine (reine) Sequenz* $\Gamma \vdash C$ *ist* gültig, *wenn es einen Term* t *gibt, für den* $\Gamma \vdash t \in C$ *ein hypothetisches Urteil ist. Ist der Term* t *bekannt, der die Sequenz gültig macht, so schreiben wir* $\underline{\Gamma \vdash C \ \lfloor \mathbf{ext}\ t \rfloor}$.

Eine Sequenz hat also insgesamt die Form $\underline{x_1{:}T_1, \ldots, x_n{:}T_n \vdash C}$, wobei die x_i Variablen sind und C sowie die T_i Terme. Eine solche Sequenz ist zu lesen als die *Behauptung*

> *Unter der Annahme, daß* x_i *Variablen vom Typ* T_i *sind, kann ein Element des Typs* C *konstruiert werden.*

[28] Um Verwechslungen mit Typen zu vermeden, die wir mit dem Symbol S kennzeichnen, verwenden wir für Sequenzen das Symbol Z, welches die Anwendung als Beweisziel heraushebt.

In der textlichen Erscheinungsform sind Sequenzen also sehr ähnlich zu den hypothetischen Urteilen, die zur Erklärung der Gültigkeit von Sequenzen herangezogen werden (und nun einmal durch irgendeine Form von Text aufgeschrieben werden müssen). Man beachte jedoch, daß hypothetische Urteile semantische Konzepte sind, die gemäß Definition 3.2.15 einen Sachverhalt als wahr "beurteilen". Im Gegensatz dazu ist eine Sequenz nur die syntaktische Repräsentation einer Behauptung, die auch ungültig – insbesondere also auch unbeweisbar – sein kann.

Es sei angemerkt, daß man auch bei der Definitionen von Sequenzen die Sprache der Terme aus Definition 3.2.5 verwenden und zum Beispiel declaration{}($x.T$) oder sequent{}($\Gamma;C$) schreiben könnte. Dies würde die Möglichkeit eröffnen, Sequenzen durch Terme der Objektsprache auszudrücken und somit innerhalb der Typentheorie über den eigenen Beweismechanismus formal zu schließen. Eine solche *Selbstreflektion* ist durchaus wünschenswert, würde zum gegenwärtigen Zeitpunkt jedoch zu weit führen. Wir beschränken uns daher auf eine textliche Definition der Beweiskonzepte und werden das Thema der Reflektion erst in späteren Kapiteln wieder aufgreifen.

Die äußere Form, die wir für Sequenzen gewählt haben, hat auch Auswirkungen auf die Struktur der Regeln. Während eine Inferenzregel bisher nur zur Zerlegung von Beweiszielen in Unterziele verwandt wurde, macht die implizite Verwaltung von Extrakt-Termen es nötig, daß die Regeln nun zwei Aufgaben übernehmen: Sie zerlegen Ziele in Teilziele und müssen gleichzeitig angeben, wie Extrakt-Terme der Teilziele – also die Evidenzen für ihre Gültigkeit – zu einem Extrakt-Term des ursprünglichen Ziels zusammengesetzt werden können. Diese beiden Teilaufgaben nennt man *Dekomposition* und *Validierung*.

Definition 3.2.26 (Regeln)

1. *Eine Dekomposition ist eine Abbildung, welche eine Sequenz – das Beweisziel – in eine endliche Liste (möglicherweise leere) von Sequenzen – die Unter- oder Teilziele – abbildet.*

2. *Eine Validierung ist eine Abbildung, welche eine endliche Liste von (Paaren von) Sequenzen und Termen in einen Term*[29] *abbildet.*

3. *Eine Inferenzregel hat die Gestalt (dec,val), wobei dec eine Dekomposition und val eine Validierung ist.*

4. *Eine Inferenzregel r = (dec,val) ist korrekt, wenn für jede reine Sequenz $Z = \Gamma \vdash C$ gilt:*
   - *Alle durch Anwendung von dec erzeugten Teilziele $Z_1 = \Gamma_1 \vdash C_1$, ..., $Z_n = \Gamma_n \vdash C_n$ sind rein.*
   - *Aus der Gültigkeit der Teilziele $Z_1, \ldots Z_n$ folgt die Gültigkeit des Hauptzieles Z.*
   - *Gilt $\Gamma_i \vdash C_i$ ⌊**ext** t_i⌋ für alle Teilziele Z_i, so folgt $\Gamma \vdash C$ ⌊**ext** t⌋, wobei t derjenige Term ist, der durch Anwendung von val auf $[(Z_1,t_1),\ldots,(Z_nt_n)]$ entsteht.*

Eine korrekte Inferenzregel zerlegt also ein Beweisziel so, daß seine Korrektheit aus der Korrektheit der Teilziele folgt, wobei die Validierungsfunktion die Evidenzen für die Gültigkeit der Ziele entsprechend zusammensetzen kann. Damit ist es möglich, in einem Beweis eine Initialsequenz durch Anwendung von Inferenzregeln schrittweise zu verfeinern und nach Abschluß des Beweises den Term, welcher die Sequenz gültig macht, automatisch – nämlich durch Zusammensetzen der Validierungen – aus dem Beweis zu extrahieren. Aus diesem Grunde wird dieser Term auch als Extrakt-Term des Beweises bzw. der Sequenz bezeichnet.

Formale Beweise sind Bäume, deren Knoten mit Sequenzen und Regeln markiert sind, wobei die Sequenzen der Nachfolger eines Knotens genau die Teilziele sind, welche sich durch Anwendung der Regel auf die Knotensequenz ergeben. Unvollständige Beweise enthalten Blätter, die nur mit einer Sequenz markiert – also *unverfeinert* – sind. Die Blätter eines vollständigen Beweises müssen notwendigerweise Regeln enthalten, welche keine Teilziele erzeugen.

[29] Um einen Extrakt-Term einer Sequenz zu bilden benötigt eine Validierung also eine neben den Extrakt-Termen der Teilziele auch die Teilziele selbst, da in deren Annahmen zuweilen notwendige Komponenten enthalten sind. Da die Sequenz zusammen mit ihrem Extrakt-Term alle Informationen des eigentlichen Beweises enthält, werden im NuPRL-System Validierungen der Einfachheit als Abbildungen dargestellt, die Listen von Beweisen in Beweise abbilden können (siehe Abschnitt 4.1.2).

Definition 3.2.27 (Beweise und Extrakt-Terme)

1. *Beweise sind induktiv wie folgt definiert*
   - *Jede Sequenz $Z = \Gamma \vdash C$ ist ein unvollständiger Beweis mit Wurzel Z*
   - *Ist $Z = \Gamma \vdash C$ eine Sequenz, r = (dec,val) eine korrekte Inferenzregel und sind $\pi_1,\ldots,\pi_n$ ($n \geq 0$) Beweise, deren Wurzeln die durch Anwendung von dec auf Z erzeugten Teilziele sind, so ist das Tupel $(Z, r, [\pi_1,\ldots,\pi_n])$ ein Beweis mit Wurzel Z.*
2. *Ein Beweis ist vollständig, wenn er keine unvollständigen Teilbeweise enthält.*
3. *Der Extrakt-Term* $\texttt{EXT}(\pi)$ *eines vollständigen Beweises $\pi = (Z, (dec,val), [\pi_1,\ldots,\pi_n])$ rekursiv definiert als $val([(Z_1,\texttt{EXT}(\pi_1)),\ldots,(Z_n,\texttt{EXT}(\pi_n))])$, wobei die Z_i die Wurzeln der Beweise π_i sind.*
4. *Eine Initialsequenz ist eine geschlossene Sequenz, deren Hypothesenliste leer ist.*
5. *Ein Theorem ist ein vollständiger Beweis, dessen Wurzel eine Initialsequenz ist.*

Die rekursive Definition von Extrakt-Termen schließt den Fall mit ein, daß ein Beweis nach Anwendung der Regeln keine weiteren Unterziele mehr enthält und die Validierungsfunktion entsprechend einen Extrakt-Term aus einer leeren Liste erzeugt. Daher ist eine Fallunterscheidung in der Definition überflüssig. Da die Verwendung abhängiger Datentypen eine (vollständige) automatische Typüberprüfung unmöglich macht, müssen die Initialsequenzen von Beweisen eine leere Hypothesenliste besitzen. Auf diese Art kann durch das Regelsystem sichergestellt werden, daß in den Unterzielen jede Deklaration auf der rechten Seite nur Typen enthält.

Das folgende Lemma zeigt, daß die obigen Definitionen tatsächlich zu Beweisen führen, welche die durch Urteile gegebene Semantik respektieren. Damit ist sichergestellt, daß jeder Beweis im Sinne der Semantik korrekt ist und dem intuitiven Verständnis der dargestellten Sequenzen entspricht. Es sei jedoch angemerkt, daß wegen der reichhaltigen Ausdruckskraft der Theorie nicht alles, was semantisch gültig ist, auch formal beweisbar sein kann.

Lemma 3.2.28

Es sei π ein Theorem mit Wurzel $\vdash T$. Dann gilt

1. *T ist ein Typ (d.h. es gilt das Urteil 'T Typ').*
2. *Für $t :=$ $\texttt{EXT}(\pi)$ ist $t \in T$ ein Urteil*
3. *Für jeden Teilbeweis π' von π mit Wurzel $Z = x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$ gilt:*
   - *Z ist eine reine Sequenz.*
   - *C und alle T_i sind Typen.*
   - *$\Gamma \vdash$ $\texttt{EXT}(\pi') \in C$ ist ein hypothetisches Urteil.*
   - *Z ist eine gültige Sequenz.*

Auch bei der Definitionen von Beweisen, Extrakttermen und Regeln könnte man die Sprache der Terme aus Definition 3.2.5 verwenden und zum Beispiel $\mathsf{unrefined}\{\}(Z)$ und $\mathsf{refined}\{\}(Z,\ r,\ [\pi_1,\ldots,\pi_n])$ als Notation verwenden. Damit ist eine Selbstreflektion innerhalb der Typentheorie tatsächlich durchführbar.[30]

3.2.3.5 Das Regelsystem im Detail

Die Beweisregeln für die einzelnen Konstrukte der formalen Theorie sind so anzulegen, daß sie die durch Urteile gegebene Semantik – also die Einträge der Typsemantiktabelle und der Elementsemantiktabelle – respektieren. Wir werden sie im folgenden gruppenweise entsprechend dem Typkonstruktor auflisten. Jede dieser Gruppen wird normalerweise vier Arten von Regeln enthalten, die wir am Beispiel der Regeln für den Funktionenraum (siehe Abbildung 3.9 auf Seite 122) erläutern wollen.

[30] Die gegenwärtige Konzeption des NuPRL Systems unterstützt diese Selbstreflektion, da Regeln und viele andere Metakonzepte – wie zum Beispiel der Editor – durch explizite Objekte der Theorie in der Syntax der Terme beschrieben werden.

1. *Formationsregeln* legen fest, wie ein Typ aus anderen Typen zusammengesetzt wird. Sie unterstützen das Schließen über Typgleichheit und die Eigenschaft, ein Typ zu sein, und könnten als Regeln über die kanonischen Elemente der Universen betrachtet werden. Die hier betrachteten Beweisziele haben die Form $\Gamma \vdash S = T \in \mathsf{U}_j$, wobei S und T kanonische Typausdrücke sind (, die gleich sind, wenn über Typ-Sein geschlossen werden soll) und der Regelname folgt dem Muster *typ*`Eq`, wobei *typ* der Name des Typs ist.[31] Für den Funktionenraum ist dies zum Beispiel die folgende Regel, die genau der in Beispiel 3.2.16 auf Seite 111 erklärten Typgleichheit von Funktionenräumen entspricht.

   $\Gamma \vdash x_1{:}S_1 \to T_1 = x_2{:}S_2 \to T_2 \in \mathsf{U}_j$ [Ax]
   by `functionEq`
   $\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ [Ax]
   $\Gamma, x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathsf{U}_j$ [Ax]

2. *Kanonische* Regeln erklären, wie die kanonischen Elemente eines Typs zusammengesetzt werden und wann sie gleich sind. Die Beweisziele haben die Gestalt $\Gamma \vdash s = t \in T$ bzw. $\Gamma \vdash T$ [**ext** t], wobei T ein kanonischer Typausdruck ist und s bzw. t die zugehörigen kanonischen Elementausdrücke. Die Regeln haben üblicherweise den Namen *element*`Eq` bzw. *element*`I`, wobei *element* der Name des kanonischen Elements ist. Als Beispiel seien hier die kanonischen Regeln des Funktionenraums genannt.

   $\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S \to T$ [Ax]
   by `lambdaEq` j
   $\Gamma, x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ [Ax]
   $\Gamma \vdash S \in \mathsf{U}_j$ [Ax]

   $\Gamma \vdash x{:}S \to T$ [**ext** $\lambda x'.t$]
   by `lambdaI` j
   $\Gamma, x'{:}S \vdash T[x'/x]$ [**ext** t]
   $\Gamma \vdash S \in \mathsf{U}_j$ [Ax]

3. *Nichtkanonische* Regeln erklären, wann ein nichtkanonischer Term zu einem vorgegebenen Typ gehört. Die hier betrachteten Beweisziele haben die Gestalt $\Gamma \vdash s = t \in T$ bzw. $\Gamma, x{:}S, \Delta \vdash T$ [**ext** t], wobei T ein beliebiger Typausdruck ist, S ein kanonischer Typausdruck, x eine Variable und s bzw. t nichtkanonische Terme sind. Im ersten Falle erklärt die Regel (*nichtkanonisch*`Eq`), wie die nichtkanonischen Ausdrücke zu zerlegen sind, während im zweiten Fall (*typ*`E`) die in der Hypothesenliste deklarierte Variable x des zugehörigen kanonischen Typs S 'eliminiert' wird, um den nichtkanonischen Extrakt-Term zu generieren. Die nichtkanonischen Regeln des Funktionenraums lauten

   $\Gamma \vdash f_1\, t_1 = f_2\, t_2 \in T[t_1/x]$ [Ax]
   by `applyEq` $x{:}S \to T$
   $\Gamma \vdash f_1 = f_2 \in x{:}S \to T$ [Ax]
   $\Gamma \vdash t_1 = t_2 \in S$ [Ax]

   $\Gamma, f{:}\, x{:}S \to T, \Delta \vdash C$ [**ext** $t[f\, s, \mathsf{Axiom}/y, z]$]
   by `functionE` i s
   $\Gamma, f{:}\, x{:}S \to T, \Delta \vdash s \in S$ [Ax]
   $\Gamma, f{:}\, x{:}S \to T, y{:}T[s/x],$
   $z{:}\, y = f\, s \in T[s/x], \Delta \vdash C$ [**ext** t]

   Die Eliminationsregel `functionE` muß $y = f\, s \in T[s/x]$ zu den Hypothesen hinzunehmen, um einen Bezug zwischen der neuen Variablen y und möglichen Vorkommen von f und s in Δ und C herzustellen.

4. Die bisherigen Regeln erlauben nur die Untersuchung einer *strukturellen Gleichheit* von Termen, bei der ausschließlich die äußere Form von Bedeutung ist. Nicht möglich ist dagegen ein Vergleich von Termen, die strukturell verschieden aber semantisch gleich sind. Aus diesem Grunde ist es nötig, Regeln mit aufzunehmen, welche den Auswertungsmechanismus widerspiegeln.

   Berechnungsregeln (*nichtkanonisch*`Red`) stellen den Bezug zwischen kanonischen und nichtkanonischen Ausdrücken eines Datentyps her. Sie sind anwendbar auf Beweisziele der Form $\Gamma \vdash s = t \in T$, wobei T ein beliebiger Typausdruck ist und einer der beiden Terme s und t ein Redex ist, das durch Anwendung der Regel zu seinem Kontraktum reduziert wird.[32] Für den Funktionenraum ist dies die folgende Regel:

[31] Es gibt auch entsprechende Regeln zu den impliziten Beweiszielen der Form $\Gamma \vdash \mathsf{U}_j$ [**ext** T]. Diese werden aber praktisch überhaupt nicht mehr benutzt und daher im folgenden nicht aufgeführt.

[32] Diese typspezifischen Reduktionsregeln sind innerhalb von NuPRL eigentlich redundant, da zugunsten einer einheitlicheren Beweisführung sogenannte '*direct computation rules*' eingeführt wurden, die es ermöglichen, den allgemeinen Auswertungsalgorithmus aus Abbildung 3.3 (Seite 108) auf jeden beliebigen Teilterm eines Ausdrucks bei gleichzeitiger Kontrolle der Anzahl der Reduktionsschritte anzuwenden. Die Details dieser typunabhängigen Berechnungsregeln werden wir in Abschnitt 3.6 besprechen.

$\Gamma \vdash x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$ ⌊Ax⌋
by functionEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$ ⌊Ax⌋
by lambdaEq j
$\Gamma,\ x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ ⌊Ax⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash x{:}S{\to}T$ ⌊**ext** $\lambda x'.t$⌋
by lambdaI j
$\Gamma,\ x'{:}S \vdash T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash f_1\,t_1 = f_2\,t_2 \in T[t_1/x]$ ⌊Ax⌋
by applyEq $x{:}S{\to}T$
$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in S$ ⌊Ax⌋

$\Gamma,\ f{:}\ x{:}S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s, \mathsf{Axiom}\,/\,y, v]$⌋
by functionE i s
$\Gamma,\ f{:}\ x{:}S{\to}T,\ \Delta \vdash s \in S$ ⌊Ax⌋
$\Gamma,\ f{:}\ x{:}S{\to}T,\ y{:}T[s/x],$
$v{:}\ y = f\,s \in T[s/x],\ \Delta \vdash C$ ⌊**ext** t⌋

$\Gamma \vdash (\lambda x.t)\,s = t_2 \in T$ ⌊Ax⌋
by applyRed
$\Gamma \vdash t[s/x] = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊**ext** t⌋
by functionExt j $x_1{:}S_1{\to}T_1$ $x_2{:}S_2{\to}T_2$
$\Gamma,\ x'{:}S \vdash f_1\,x' = f_2\,x' \in T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma \vdash f_1 \in x_1{:}S_1{\to}T_1$ ⌊Ax⌋
$\Gamma \vdash f_2 \in x_2{:}S_2{\to}T_2$ ⌊Ax⌋

$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s, /y]$⌋
by functionE_indep i
$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash S$ ⌊**ext** s⌋
$\Gamma,\ f{:}\ S{\to}T,\ y{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋

Abbildung 3.9: Regeln für Funktionenraum

$\Gamma \vdash (\lambda x.t)\,s = t_2 \in T$ ⌊Ax⌋
by applyRed
$\Gamma \vdash t[s/x] = t_2 \in T$ ⌊Ax⌋

– In manchen Gruppen kommen noch *besondere Regeln* zu diesen vier Regelarten hinzu. So gilt im Falle des Funktionenraumes die Gleichheit zweier Funktionen nicht nur, wenn die Regel `lambdaEq` anwendbar ist, sondern auch dann, wenn die Funktionen sich auf allen Argumenten gleich verhalten. Diese *Extensionalität* ist eine besondere Eigenschaft von Funktionen, die deutlich macht, daß nicht die innere Struktur, sondern das äußere Verhalten das wesentliche Charakteristikum einer Funktion ist. Die formale Regel berücksichtigt auch, daß die beiden genannten Funktionen zu völlig verschiedenen Funktionenräumen gehören können, die eine "Obermenge" des genannten Funktionenraums $x{:}S{\to}T$ sind.

$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊**ext** t⌋
by functionExt j $x_1{:}S_1{\to}T_1$ $x_2{:}S_2{\to}T_2$
$\Gamma,\ x'{:}S \vdash f_1\,x' = f_2\,x' \in T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma \vdash f_1 \in x_1{:}S_1{\to}T_1$ ⌊Ax⌋
$\Gamma \vdash f_2 \in x_2{:}S_2{\to}T_2$ ⌊Ax⌋

$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s, /y]$⌋
by functionE_indep i
$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash S$ ⌊**ext** s⌋
$\Gamma,\ f{:}\ S{\to}T,\ y{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋

Hinzu kommt eine besondere Behandlung des unabhängigen Funktionenraums $S{\to}T$, der als Spezialfall eines abhängigen Funktionenraums gesehen werden kann, bei dem die Bindungsvariable keine Rolle spielt. Die meisten Regeln des unabhängigen Funktionenraums sind direkte Spezialisierungen der obengenannten Regeln. Bei der Eliminationsregel ergibt sich allerdings eine so starke Vereinfachung, daß hierfür eine gesonderte Regel existiert, bei der ein Anwender einen Parameter weniger angeben muß.

$\Gamma \vdash x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$ ⌊Ax⌋
by `productEq`
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma,\ x'{:}S_1 \vdash T_1[x'/x_1] = T_2[x'/x_2] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in x{:}S{\times}T$ ⌊Ax⌋
by `pairEq` j
$\Gamma \vdash s_1 = s_2 \in S$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in T[s_1/x]$ ⌊Ax⌋
$\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash x{:}S{\times}T$ ⌊**ext** $\langle s,t\rangle$⌋
by `pairI` j s
$\Gamma \vdash s \in S$ ⌊Ax⌋
$\Gamma \vdash T[s/x]$ ⌊**ext** t⌋
$\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash$ **let** $\langle x_1,y_1\rangle = e_1$ **in** t_1
$\quad = $ **let** $\langle x_2,y_2\rangle = e_2$ **in** $t_2 \in C[e_1/z]$ ⌊Ax⌋
by `spreadEq` z C $x{:}S{\times}T$
$\Gamma \vdash e_1 = e_2 \in x{:}S{\times}T$ ⌊Ax⌋
$\Gamma,\ s{:}S,\ t{:}T[s/x],\ y{:}\ e_1{=}\langle s,t\rangle \in x{:}S{\times}T$
$\quad \vdash t_1[s,t/x_1,y_1] = t_2[s,t/x_2,y_2] \in C[\langle s,t\rangle/z]$ ⌊Ax⌋

$\Gamma,\ z{:}\ x{:}S{\times}T,\ \Delta \vdash C$ ⌊**ext** **let** $\langle s,t\rangle = z$ **in** u⌋
by `productE` i
$\Gamma,\ z{:}\ x{:}S{\times}T,\ s{:}S,\ t{:}T[s/x],\ \Delta[\langle s,t\rangle/z]$
$\quad \vdash C[\langle s,t\rangle/z]$ ⌊**ext** u⌋

$\Gamma \vdash$ **let** $\langle x,y\rangle = \langle s,t\rangle$ **in** $u = t_2 \in T$ ⌊Ax⌋
by `spreadRed`
$\Gamma \vdash u[s,t/x,y] = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T$ ⌊Ax⌋
by `pairEq_indep`
$\Gamma \vdash s_1 = s_2 \in S$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash S{\times}T$ ⌊**ext** $\langle s,t\rangle$⌋
by `pairI_indep`
$\Gamma \vdash S$ ⌊**ext** s⌋
$\Gamma \vdash T$ ⌊**ext** t⌋

Abbildung 3.10: Regeln für Produktraum

Wir haben diese Regeln wie immer in ihrer allgemeinsten Form durch Regelschemata beschrieben, welche auf die konkrete Situation angepaßt werden muß. Dabei haben wir im wesentlichen die Dekomposition – also die Form, die ein Anwender der Theorie beim Arbeiten mit NuPRL sehen wird – hervorgehoben während das Validierungsschema in eckigen Klammern (⌊**ext** t⌋) neben den entsprechenden Sequenzenschemata steht. Zur Vereinfachung der textlichen Präsentation haben wir Extrakt-Terme der Gestalt ⌊**ext** Axiom⌋ durch ⌊Ax⌋ und Teilziele der Gestalt $s{=}s \in T$ durch $s \in T$ abgekürzt.[33]

Die neuen Variablen in den Unterzielen werden automatisch generiert, falls Umbenennungen erforderlich sind. Es gibt jedoch noch eine Reihe weiterer Parameter, die von manchen Regeln zu Steuerungszwecken benötigt werden und vom Benutzer anzugeben sind. Dies sind

1. Die Position i einer zu elimierenden Variablen innerhalb der Hypothesenliste wie zum Beispiel bei der Regel `functionE_indep` i.

2. Ein Term, der als einzusetzender Wert für die weitere Verarbeitung benötigt wird, wie zum Beispiel s in der Regel `functionE` i s.

3. Das Level j des Universums, was innerhalb eines Unterziels benötigt wird, um nachzuweisen, daß ein bestimmter Teilterm ein Typ ist. Dies muß zum Beispiel in `lambdaEq` j angegeben werden, während es im Falle der Regel `functionEq` bereits bekannt ist.

4. Der Typ T eines Teilterms des zu analysierenden Beweiszieles, welcher in einem Unterziel isoliert auftritt. Dies wird vor allem beim Schließen über nichtkanonische Formen benötigt, in denen der Typ des Hauptargumentes nicht eindeutig aus dem Kontext hervorgeht. So muß zum Beispiel bei `applyEq` $x{:}S{\to}T$ der Typ der Funktion f_1 genannt werden, die als Hauptargument der Applikation auftritt.

[33] Diese Abkürzungen sind jedoch eine reine Notationsform für die Regeln und nicht etwa ein Teil der Typentheorie selbst. In den Regelschemata von NuPRL steht ⌊**ext** Axiom⌋, wo in diesem Skript ⌊Ax⌋ steht und $s{=}s \in T$, wo $s \in T$ steht.

$\Gamma \vdash S_1{+}T_1 \;=\; S_2{+}T_2 \;\in\; \mathsf{U}_j$ [Ax]
by unionEq
$\Gamma \vdash S_1 \;=\; S_2 \;\in\; \mathsf{U}_j$ [Ax]
$\Gamma \vdash T_1 \;=\; T_2 \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{inl}(s_1) \;=\; \mathsf{inl}(s_2) \;\in\; S{+}T$ [Ax]
by inlEq j
$\Gamma \vdash s_1 \;=\; s_2 \;\in\; S$ [Ax]
$\Gamma \vdash T \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash S{+}T$ [**ext** $\mathsf{inl}(s)$]
by inlI j
$\Gamma \vdash S$ [**ext** s]
$\Gamma \vdash T \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{inr}(t_1) \;=\; \mathsf{inr}(t_2) \;\in\; S{+}T$ [Ax]
by inrEq j
$\Gamma \vdash t_1 \;=\; t_2 \;\in\; T$ [Ax]
$\Gamma \vdash S \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash S{+}T$ [**ext** $\mathsf{inr}(t)$]
by inrI j
$\Gamma \vdash T$ [**ext** t]
$\Gamma \vdash S \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}(x_1) \mapsto u_1 \mid \mathsf{inr}(y_1) \mapsto v_1$
$\quad = \mathsf{case}\ e_2\ \mathsf{of}\ \mathsf{inl}(x_2) \mapsto u_2 \mid \mathsf{inr}(y_2) \mapsto v_2$
$\quad \in C[e_1/z]$ [Ax]
by decideEq z C $S{+}T$
$\Gamma \vdash e_1 \;=\; e_2 \;\in\; S{+}T$ [Ax]
$\Gamma,\ s{:}S,\ y{:}\ e_1{=}\mathsf{inl}(s) \in S{+}T$
$\quad \vdash u_1[s/x_1] = u_2[s/x_2] \;\in\; C[\mathsf{inl}(s)/z]$ [Ax]
$\Gamma,\ t{:}T,\ y{:}\ e_1{=}\mathsf{inr}(t) \in S{+}T$
$\quad \vdash v_1[t/y_1] = v_2[t/y_2] \;\in\; C[\mathsf{inr}(t)/z]$ [Ax]

$\Gamma,\ z{:}S{+}T,\ \Delta \vdash C$
$\quad$ [**ext** $\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$]
by unionE i
$\Gamma,\ z{:}S{+}T,\ x{:}S,\ \Delta[\mathsf{inl}(x)/z]$
$\quad \vdash C[\mathsf{inl}(x)/z]$ [**ext** u]
$\Gamma,\ z{:}S{+}T,\ y{:}T,\ \Delta[\mathsf{inr}(y)/z]$
$\quad \vdash C[\mathsf{inr}(y)/z]$ [**ext** v]

$\Gamma \vdash \mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$
$\quad = t_2 \;\in\; T$ [Ax]
by decideRedL
$\Gamma \vdash u[s/x] \;=\; t_2 \;\in\; T$ [Ax]

$\Gamma \vdash \mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$
$\quad = t_2 \;\in\; T$ [Ax]
by decideRedR
$\Gamma \vdash v[t/y] \;=\; t_2 \;\in\; T$ [Ax]

Abbildung 3.11: Regeln für disjunkte Vereinigung (Summe)

5. Ein Term C und eine Variable z, die in C frei vorkommt. Dies ist in manchen Fällen nötig, um die Abhängigkeit eines Teilterms der Konklusion von einem anderen Teilterm deutlich zu machen. Dies wird zum Beipiel in der nichtkanonischen Regel `spreadEq` $z\ C\ x{:}S{\times}T$ des Produktraumes benötigt, welche in Abbildung 3.10 aufgeführt ist.

   Im Typ der Gleichheit $\mathsf{let}\ \langle x_1, y_1\rangle = e_1\ \mathsf{in}\ t_1 \;=\; \mathsf{let}\ \langle x_2, y_2\rangle = e_2\ \mathsf{in}\ t_2 \;\in\; C[e_1/z]$ kann es sein, daß bestimmte Vorkommen des Teilterms e_1 in C mit dem e_1 des let-Konstruktes in Verbindung zu bringen sind und andere nicht. Ist zum Beispiel C der Term $e_1{=}e_1 \in T$, so könnte dies zum Beispiel eine Instanz von $e_1{=}z \in T$ sein oder eine von $z{=}z \in T$.

   Da die Konklusion C üblicherweise in instantiierter Form vorliegt, muß die intendierte Abhängigkeit zwischen C und dem Teilterm e_1 explizit angegeben werden. Dies kann dadurch geschehen, daß man die Konlusion C dadurch verallgemeinert, daß man e_1 an den gewünschten Stellen durch eine neue Variable z ersetzt (also die Konklusion als Instanz $C[e_1/z]$ des allgemeineren Terms auffaßt) und sowohl z als auch die verallgemeinerte Konklusion als Steuerungsparameter für die Regel mit angibt. Damit wird zum Beispiel bei Anwendung von `spreadEq` festgelegt, e_1 ersetzt werden und welche bestehen bleiben, also ob im zweiten von `spreadEq` erzeugten Teilziel $e_1{=}\langle s, t\rangle \in T$ oder $\langle s, t\rangle{=}\langle s, t\rangle \in T$ steht.

In den meisten Fällen können die letzten drei Parameter automatisch berechnet werden, da sie aus dem Beweiskontext eindeutig hervorgehen. Da dies aber nicht in allen Beweisen der Fall ist, müssen sie als separate Bestandteile der Regeln mit aufgeführt werden. Wir wollen im folgenden nun alle Beweisregeln der Typentheorie tabellarisch zusammenstellen, die bis zu diesem Zeitpunkt relevant sind und die Besonderheiten, die sich nicht unmittelbar aus den Semantiktabellen ergeben, kurz erläutern.

$\Gamma \vdash \mathsf{U}_j \in \mathsf{U}_k$ [Ax] **by** univEq *	$\Gamma \vdash T \in \mathsf{U}_k$ [Ax] **by** cumulativity j * $\Gamma \vdash T \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash s_1 = t_1 \in T_1 = s_2 = t_2 \in T_2 \in \mathsf{U}_j$ [Ax] **by** equalityEq $\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax] $\Gamma \vdash s_1 = s_2 \in T_1$ [Ax] $\Gamma \vdash t_1 = t_2 \in T_1$ [Ax]	$\Gamma \vdash \mathsf{Axiom} \in s = t \in T$ [Ax] **by** axiomEq $\Gamma \vdash s = t \in T$ [Ax] $\Gamma, z\colon s = t \in T, \Delta \vdash C$ [**ext** u] **by** equalityE i $\Gamma, z\colon s = t \in T, \Delta[\mathsf{Axiom}/z]$ $\vdash C[\mathsf{Axiom}/z]$ [**ext** u]
$\Gamma, x\colon T, \Delta \vdash x \in T$ [Ax] **by** hypEq i	$\Gamma \vdash s = t \in T$ [Ax] **by** symmetry $\Gamma \vdash t = s \in T$ [Ax]
$\Gamma \vdash s = t \in T$ [Ax] **by** transitivity t' $\Gamma \vdash s = t' \in T$ [Ax] $\Gamma \vdash t' = t \in T$ [Ax]	$\Gamma \vdash C[s/x]$ [**ext** u] **by** subst j $s = t \in T$ x C $\Gamma \vdash s = t \in T$ [Ax] $\Gamma \vdash C[t/x]$ [**ext** u] $\Gamma, x\colon T \vdash C \in \mathsf{U}_j$ [Ax]

*: *Die Regel ist nur anwendbar, wenn $j < k$ ist.*

Abbildung 3.12: Regeln für Universen und Gleichheit

Die Regeln für den *Produktraum* in Abbildung 3.10 sind strukturell sehr ähnlich zu denen des Funktionenraumes. Um die Gleichheit zweier Tupel nachzuweisen, muß man sie komponentenweise untersuchen und – im Falle abhängiger Produkte $x\colon S \times T$ – zusätzlich die allgemeine Typ-Eigenschaft von T nachweisen. Um ein Element von $x\colon S \times T$ einführen zu können, muß man im abhängigen Fall die erste Komponente explizit angeben, da der Typ der zweiten davon abhängt. Bei unabhängigen Produkten sind die entsprechenden Regeln erheblich einfacher und daher separat aufgeführt. Die Gleichheit der nichtkanonischen Formen (`spreadEq`) wird ebenfalls durch strukturelle Dekomposition in Teilausdrücke untersucht, wobei die obenerwähnten Komplikationen zu berücksichtigen sind. Die Elimination einer Produktvariablen z führt zur Einführung zweier Variablen für die Einzelkomponenten, die als Tupel an die Stelle von z treten.

Die meisten Regeln für die *disjunkte Vereinigung* in Abbildung 3.11 sind naheliegend. Bei den kanonischen Regeln für inl und inr muß jedoch sichergestellt werden, daß der *gesamte* Ausdruck $S+T$ einen Typ darstellt. Da jeweils nur noch einer der beiden Teiltypen weiter betrachtet wird, müssen die Teilziele $T \in \mathsf{U}_j$ bzw. $S \in \mathsf{U}_j$ in diese Regeln explizit hineingenommen werden. Die Regel `decideEq` entspricht der Fallanalyse.

Wie in Abschnitt 3.2.3.1 besprochen, dient eine kumulative Hierarchie von *Universen* der syntaktischen Repräsentation der Typeigenschaft. Alle Typkonstruktoren – einschließlich U_j für $j < k$ – erzeugen somit kanonische Elemente eines Universums U_k. Da die entsprechenden kanonischen Regeln für Universen bereits als Formationsregeln des entsprechenden Typs aufgeführt wurden, gibt es keine speziellen kanonischen Regeln für Universen. Nichtkanonische Ausdrücke für Universen gibt es ebenfalls nicht und damit entfällt auch die Berechnungsregel. Bedeutend ist jedoch eine Regel zur Darstellung der Kumulativität, die es ermöglicht, Ziele wie $T \to \mathsf{U}_1 \in \mathsf{U}_2$ zu beweisen, wenn T nur ein Element von U_1 ist.

Der Typ der *Gleichheit* besitzt nur ein kanonisches Element Axiom und keine nichtkanonischen Ausdrücke. Aus diesem Grunde erzeugt jede Regel, deren Beweisziel eine Gleichheit ist, den Term Axiom als Extrakt-Term. Die Regel `equalityE` wird relativ selten eingesetzt, da sie wenig praktische Auswirkungen hat. Die einzig bedeutenden Regeln sind die bekannten Regeln zum Schließen über Gleichheit. Dabei ist `hypEq` eine Art bedingte Reflexivitätsregel. Allgemeine Reflexivität (`x = x ∈ T`) gilt nicht, da Typzugehörigkeit geprüft

$\Gamma,\ x{:}T,\ \Delta \vdash T$ ⌊**ext** x⌋ **by** `hypothesis` i	$\Gamma \vdash T$ ⌊**ext** t⌋ **by** `intro` t $\Gamma \vdash t \in T$ ⌊Ax⌋
$\Gamma,\Delta \vdash C$ ⌊**ext** $(\lambda x.t)\ s$⌋ **by** `cut` $i\ T$ $\Gamma,\Delta \vdash T$ ⌊**ext** s⌋ $\Gamma,x{:}T,\Delta \vdash C$ ⌊**ext** t⌋	$\Gamma,\ x{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋ **by** `thin` i $\Gamma,\Delta \vdash C$ ⌊**ext** t⌋

Abbildung 3.13: Strukturelle Regeln

werden muß. Die Regeln `symmetry` und `transitivity` sind – wie bereits in Beispiel 2.2.26 auf Seite 42 – redundant, da sie durch die Substitutionsregel simuliert werden können.[34] Die Regeln für Universen und Gleichheit sind in Abbildung 3.12 zusammengestellt.

Neben den Regeln, welche die Semantik einzelner Ausdrücke der Typentheorie wiederspiegeln, benötigen wir noch eine Reihe *struktureller Regeln*, von denen wir einige bereits aus Abschnitt 2.2.4 kennen.

- Mit der *Hypothesenregel* `hypothesis` können wir Konklusionen beweisen, die bereits als Typausdruck in der Hypothesenliste erscheinen. Extrakt-Term ist in diesem Falle die an dieser Stelle deklarierte Variable.

- Eine *explizite Einführungsregel* `intro` ermöglicht es, den Extrakt-Term für eine Konklusion direkt anzugeben. In diesem Falle muß natürlich noch seine Korrektheit überprüft werden.

- Die *Schnittregel* `cut` erlaubt das Einfügen von Zwischenbehauptungen. Diese sind in einem Teilziel zu beweisen und dürfen in einem anderen zum Beweis weiterverwendet werden. Die gewünschte Position der Zwischenbehauptung in der Hypothesenliste kann angegeben werden, wobei aber darauf zu achten ist, daß alle freien Variablen der Zwischenbehauptung in früheren Hypothesen deklariert sein müssen.

- Mit der *Ausdünnungsregel* `thin` werden überflüssige Annahmen aus der Hypothesenliste entfernt, um den Beweis überschaubarer zu machen. Deklarationen von noch benutzten freien Variablen können jedoch nicht ausgedünnt werden.

3.2.4 Grundprinzipien des systematischen Aufbaus

In diesem Abschnitt haben wir die allgemeinen Prinzipien und Entwurfsentscheidungen vorgestellt, die beim formalen Aufbau der Typentheorie eine zentrale Rolle spielen und am Beispiel dreier Typkonstrukte erläutert. Der wesentliche Grundsatz war dabei, Definitionen bereitzustellen, welche die formale Struktur von Syntax, Semantik und Inferenzsystem *unabhängig von der konkreten Theorie* eindeutig festlegen und es dann erlauben, die eigentliche Theorie kurz und einfach durch Einträge in diversen Tabellen zu definieren, welche die Anforderungen der allgemeinen Definitionen respektieren. Dies erleichtert sowohl ein Verständnis als auch eine Implementierung der formalen Theorie.

Zugunsten einer einheitlichen Syntax wurde entschieden, die *Abstraktionsform* eines Terms und seine *Darstellungsform* getrennt voneinander zu behandeln. Dies ermöglicht, eine einfache Definition für Terme, Variablenbindung und Substitution zu geben und in einer *Operatorentabelle* die konkrete Syntax der Theorie anzugeben. Hierdurch gewinnen wir maximale Flexibilität bei der formalen Repräsentation von Konstrukten aus Mathematik und Programmierung innerhalb eines computerisierten Beweissystems.

[34] In NuPRL gibt es anstelle der beiden Regeln `symmetry` und `transitivity` eine komplexere `equality` Regel. Diese Regel ist als Entscheidungsprozedur zum allgemeinen Schließen über Gleichheit implementiert und kann eine vielfache Anwendung von (bedingter) Reflexivität, Symmetrie und Transitivität in einem Schritt durchführen. Wir werden diese Regel und den zugrundeliegenden Algorithmus bei der Besprechung der Automatisierungsmöglichkeiten im nächsten Kapitel ausführlicher vorstellen.

Die Semantik der Typentheorie basiert auf *Lazy Evaluation*. Bestimmte Terme werden anhand ihres Operatornamens als kanonisch deklariert und nichtkanonische Terme werden nur soweit ausgewertet, bis ihre äußere Form kanonisch ist. Neben dem allgemeinen Auswertungsalgorithmus benötigt die konkrete Ausprägung der Theorie hierfür nur eine *Tabelle von Redizes und Kontrakta*. Die eigentliche Semantik wird nun durch *Urteile* über Terme festgelegt, wobei es nur vier Formen von Grundurteilen gibt. In einer allgemeinen Definition wird fixiert, wie für gegebene Terme die entsprechenden Urteile zu bestimmen sind, und für eine konkrete Ausprägung dieser Urteile werden wieder Einträge in einer Semantiktabelle vorgenommen.

Die syntaktische Repräsentationsform der semantischen Urteile innerhalb des Inferenzsystems bilden die *Sequenzen*. Zugunsten einer einheitlichen Darstellung wurde ein Gleichheitstyp und der Typ der Universen eingeführt. Hierbei wurde entschieden, logische Aussagen durch Typen darzustellen, deren Elemente ihren Beweisen entsprechen (*Propositionen als Datentypen*), und eine *kumulative Hierarchie von Universen* zu verwenden, um Widersprüche zu vermeiden. Der Kalkül wurde so ausgelegt, daß eine interaktive *Entwicklung* von Beweisen und Programmen unterstützt wird und nicht nur ihre Überprüfung. Dementsprechend müssen Inferenzregeln nicht nur Beweisziele in Teilziele zerlegen, sondern auch implizit *Extrakt-Terme* verwalten.

Auf dieser Basis ist bei (nicht-konservativer) Einführung eines neuen Konstrukts wie folgt vorzugehen.

- Die Syntax und die Darstellungsform der neuen Terme ist in die Operatortabelle einzutragen.
- Kanonische und nichtkanonische Terme sind zu trennen. Die Redex-Kontrakta Tabelle ist entsprechend zu erweitern.
- Die Semantik (Typgleichheit und Elementegleichheit) ist durch Einträge in der Typ- bzw. Elementsemantiktabelle zu fixieren. Hierbei ist darauf zu achten, daß keine Widersprüchlichkeiten entstehen.
- Es sind Inferenzregeln aufzustellen, welche diese Semantik weitestgehend wiederspiegeln.

Bei einer konservativen Erweiterung durch einen Benutzer der Theorie geht man im Prinzip genauso vor. Allerdings wird hierbei die Abstraktionsform als Abkürzung für einen bereits bestehenden komplexeren Term definiert. Daher ergibt sich die (widerspruchsfreie!) Semantik und die zugehörigen Inferenzregeln mehr oder weniger automatisch aus der Definition.

3.3 Logik in der Typentheorie

Mit den bisher vorgestellten Konstrukten können wir die elementaren Grundkonzepte der Programmierung innerhalb der Typentheorie repräsentieren. Wir haben dabei allerdings schon öfter angedeutet, daß wir die Prädikatenlogik über das Prinzip “Propositionen als Datentypen” einbetten wollen. Dies wollen wir nun weiter ausarbeiten und die Konsequenzen der Einbettung untersuchen.

3.3.1 Die Curry-Howard Isomorphie

In Abschnitt 2.4.7 (Seite 87) hatten wir die *Curry-Howard Isomorphie* zwischen der einfachen Typentheorie und dem Implikationsfragment der Prädikatenlogik aufgedeckt. Wir wollen nun zeigen, daß auch die anderen logischen Operationen ein isomorphes Gegenstück innerhalb der Typentheorie besitzen. Wir bedienen uns hierzu einer konstruktiven Interpretation der logischen Symbole, die davon ausgeht, daß eine logische Aussage wahr ist, wenn wir einen Beweis – also ein Element des entsprechenden Datentyps – finden können.

Konjunktion: Um $A \wedge B$ zu beweisen, müssen wir in der Lage sein, A und B unabhängig voneinander zu beweisen. Ist also a ein Beweis für A und b einer für B, dann liefern a und b zusammen – also zum Beispiel das Paar $\langle a, b \rangle$ – alle Evidenzen, die wir benötigen, um die Gültigkeit von $A \wedge B$ nachzuweisen. Damit verhält sich die Konjunktion im wesentlichen wie ein *(unabhängiger) Produktraum*. Dies wird

besonders deutlich, wenn man die entsprechenden Inferenzregeln für die Konjunktion mit den impliziten Regeln des unabhängigen Produkts vergleicht.[35]

$\Gamma \vdash A \wedge B$
by `and_i`
$\Gamma \vdash A$
$\Gamma \vdash B$

$\Gamma \vdash A \times B$ ⌊**ext** $\langle a, b\rangle$⌋
by `pairI_indep`
$\Gamma \vdash A$ ⌊**ext** a⌋
$\Gamma \vdash B$ ⌊**ext** b⌋

Die Einführungsregel für die Konjunktion ist nichts anderes als ein Spezialfall der Regel `pairI_indep`, bei dem man die Validierung unterdrückt hat. Ähnlich sieht es bei den Eliminationsregeln aus.

$\Gamma, A \wedge B, \Delta \vdash C$
by `and_e` i
$\Gamma, A, B, \Delta \vdash C$

$\Gamma,\ z\colon A \times B,\ \Delta \vdash C$ ⌊**ext** **let** $\langle a, b\rangle = z$ **in** u⌋
by `productE` i
$\Gamma,\ z\colon A \times B,\ a\colon A,\ b\colon B,\ \Delta[\langle a, b\rangle / z]$
$\vdash C[\langle a, b\rangle / z]$ ⌊**ext** u⌋

Der Unterschied besteht hier darin, daß innerhalb der Typentheorie alle Formeln (Typen) in den Hypothesen mit Variablen in Verbindung gebracht werden und mögliche Abhängigkeiten der Hypothesen und der Konklusion von diesen Variablen ebenfalls berücksichtigt sind. Streicht man diese Abhängigkeiten aus der Regel wieder heraus, weil sie innerhalb der Prädikatenlogik nicht auftauchen können, und dünnt danach[36]die redundante Hypothese $z\colon A \times B$ aus, so erhält man folgende Spezialisierung von `productE` i, die den Zusammenhang zu `and_e` i deutlich macht.

$\Gamma,\ z\colon A \times B,\ \Delta \vdash C$ ⌊**ext** **let** $\langle a, b\rangle = z$ **in** u⌋
by `productE` i `THEN thin` i
$\Gamma,\ a\colon A,\ b\colon B,\ \Delta \vdash C$ ⌊**ext** u⌋

Aus dieser Regel wird auch deutlich, welche Evidenzen sich ergeben, wenn man eine Konjunktion eliminiert: ist bekannt, wie aus Beweisen a für A und b für B ein Beweis u für C aufgebaut werden kann, so beschreibt **let** $\langle a, b\rangle = z$ **in** u, wie der Beweis für C aus einem Beweis z für $A \wedge B$ zu konstruieren ist.

Disjunktion: Um $A \vee B$ zu beweisen, müssen wir entweder A oder B beweisen können. Die Menge aller Beweise für $A \vee B$ ist damit die *disjunkte Vereinigung* der Beweise für A und der Beweise für B. Auch hier zeigt ein Vergleich der entsprechenden Beweisregeln die genauen Zusammenhänge auf.

$\Gamma \vdash A \vee B$
by `or_i1`
$\Gamma \vdash A$

$\Gamma \vdash A + B$ ⌊**ext** **inl**(a)⌋
by `inlI` j
$\Gamma \vdash A$ ⌊**ext** a⌋
$\Gamma \vdash B \in \mathsf{U}_j$ ⌊Ax⌋

Der Unterschied zwischen diesen beiden Regeln besteht nur darin, daß bei der disjunkten Vereinigung das Level des Universums von B angegeben werden muß. Innerhalb der Prädikatenlogik *erster Stufe* steht eigentlich fest, daß immer das Level 1 zu wählen ist, da Typen (Propositionen) höherer Universen (Stufen) nicht erlaubt sind. Mit dem zweiten Ziel ist also nur nachzuweisen, daß B tatsächlich eine Formel ist. Wegen der wesentlich einfacheren Syntax der Prädikatenlogik, die eine Mischung von Formeln und Termen verbietet (siehe Definition 2.2.4 auf Seite 24), ist diese Überprüfung jedoch automatisierbar und könnte ganz entfallen. Wir werden daher in den folgenden Vergleichen derartige *Wohlgeformtheitsziele* unterdrücken[37] und erhalten als zweite Einführungsregel für die Disjunktion

$\Gamma \vdash A \vee B$
by `or_i2`
$\Gamma \vdash B$

$\Gamma \vdash A + B$ ⌊**ext** **inr**(b)⌋
by `inrI 1 THEN ...`
$\Gamma \vdash B$ ⌊**ext** b⌋

Die spezialisierte Eliminationsregel `unionE` i erweitert die Regel `or_e` i ebenfalls wieder nur um die Konstruktion der Evidenzen. Hierdurch wird auch deutlicher, daß die Elimination einer Disjunktion im wesentlichen eine Fallunterscheidung ist.

[35]Zum Zwecke der Vergleichbarkeit haben wir in den typentheoretischen Regelschemata S durch A und T durch B ersetzt.

[36]Das *Tactical* `THEN` (siehe Abschnitt 4.2.4) erlaubt es, mehrere Regeln direkt miteinander zu verbinden.

[37]In einer Logik höherer Stufe, in der auch über beliebig zusammengesetzte Formeln geschlossen werden darf, muß das Wohlgeformtheitsziel dagegen bestehen bleiben, da hier eine vollständige Automatisierung nicht möglich ist.

$\Gamma, A \vee B, \Delta \vdash C$
by `or_e` i
$\Gamma, A, \Delta \vdash C$
$\Gamma, B, \Delta \vdash C$

$\Gamma,\ z{:}A{+}B,\ \Delta \vdash C$
$\lfloor$**ext** case z of inl$(a) \mapsto u$ | inr$(b) \mapsto v$ $\rfloor$
by `unionE` i `THEN thin` i
$\Gamma,\ a{:}A,\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$
$\Gamma,\ b{:}B,\ \Delta \vdash C$ $\lfloor$**ext** $v\rfloor$

Auch zeigt die Regel `unionE`, welche Evidenzen sich bei der Elimination ergeben: ist bekannt, wie aus einem Beweis a für A bzw. aus b für B ein Beweis u (bzw. v) für C aufgebaut werden kann, so beschreibt case z of inl$(a) \mapsto u$ | inr$(b) \mapsto v$ einen Beweis für C auf der Grundlage eines Beweises z für $A \vee B$.

Implikation: Der grundsätzliche Zusammenhang zwischen der Implikation und dem *unabhängigen Funktioneraum* ist bereits aus Abschnitt 2.4.7 bekannt. Auch hier machen die spezialisierten Regeln (mit Unterdrückung der Wohlgeformtheitsziele) deutlich, wie die Beweise zusammengesetzt werden.

$\Gamma \vdash A \Rightarrow B$
by `imp_i`
$\Gamma, A \vdash B$

$\Gamma \vdash A \to B$ $\lfloor$**ext** $\lambda x.b\rfloor$
by `lambdaI 1 THEN ...`
$\Gamma,\ x{:}A \vdash B$ $\lfloor$**ext** $b\rfloor$

$\Gamma, A \Rightarrow B, \Delta \vdash C$
by `imp_e` i
$\Gamma, A \Rightarrow B, \Delta \vdash A$
$\Gamma, \Delta, B \vdash C$

$\Gamma,\ pf{:}\ A \to B,\ \Delta \vdash C$ $\lfloor$**ext** $b[pf\,a,/y]\rfloor$
by `functionE_indep` i `THEN ...`
$\Gamma,\ pf{:}\ A \to B,\ \Delta \vdash A$ $\lfloor$**ext** $a\rfloor$
$\Gamma,\ y{:}B,\ \Delta \vdash C$ $\lfloor$**ext** $b\rfloor$

In der spezialisierten Regel wird die Hypothese $pf{:}\ A \to B$ nur im zweiten Teilziel ausgedünnt.

Negation: $\neg A$ kann gemäß unserer Definiton auf Seite 34 als Abkürzung für $A \Rightarrow \Lambda$ angesehen werden. Damit ergibt sich die Repräsentation der Negation aus dem unabhängigen Funktionenraum und der Darstellung der Falschheit.

Falschheit: Λ ist eine logische Aussage, für die es keine Beweise geben darf. Der zugehörige Datentyp muß also ein Typ ohne Elemente – ein Gegenstück zur leeren Menge sein. Diesen Datentyp void werden wir im folgenden Abschnitt genauer besprechen.

Universelle Quantifizierung: Um $\forall x{:}T.A$ zu beweisen, müssen wir eine allgemeine Methode angeben, wie wir $A[x]$ für ein beliebiges $x \in T$ beweisen, ohne weitere Informationen über x zu besitzen. Damit ist diese Methode im wesentlichen eine Funktion, welche für alle Eingaben $x \in T$ einen Beweis von $A[x]$ liefert. Der Ausgabetyp dieser Funktion – die Proposition $A[x]$ – hängt allerdings von x ab und deshalb ist das typentheoretische Gegenstück zur universellen Quantifizierung ein *abhängiger Funktionenraum*. Auch hier zeigen die spezialisierten Regeln die genauen Zusammenhänge.

$\Gamma \vdash \forall x{:}T.A$
by `all_i` *
$\Gamma, x'{:}T \vdash A[x'/x]$

$\Gamma \vdash x{:}T \to A$ $\lfloor$**ext** $\lambda x'.a\rfloor$
by `lambdaI 1 THEN ...`
$\Gamma,\ x'{:}T \vdash A[x'/x]$ $\lfloor$**ext** $a\rfloor$

Man beachte, daß die Eigenvariablenbedingung * in `all_i` "*Die Umbenennung* $[x'/x]$ *erfolgt, wenn* x *in* Γ *frei vorkommt*" innerhalb der typentheoretischen Regeln durch ein wesentlich einfacheres Kriterium dargestellt wird. Um die Reinheit der Sequenz zu erhalten, erfolgt eine Umbenennung $[x'/x]$, wenn x in Γ bereits deklariert wurde.

$\Gamma, \forall x{:}T.A, \Delta \vdash C$
by `all_e` i t
$\Gamma, \forall x{:}T.A, \Delta, A[t/x] \vdash C$

$\Gamma,\ pf{:}\ x{:}T \to A,\ \Delta \vdash C$ $\lfloor$**ext** $u[pf\,t\,/y]\rfloor$
by `functionE` i t
$\Gamma,\ pf{:}\ x{:}T \to A,\ \Delta \vdash t \in T$ $\lfloor$Ax$\rfloor$
$\Gamma,\ pf{:}\ x{:}T \to A,\ y{:}A[t/x],\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$

Das zusätzliche erste Teilziel von `functionE` i t stellt sicher, daß es sich bei dem angegebenen Term t tatsächlich um einen Term vom Typ T handelt. Innerhalb der einfachen Prädikatenlogik kann dieser Zusammenhang nicht überprüft werden.

Existentielle Quantifizierung: Um $\exists x{:}T\,.\,A$ zu beweisen, müssen wir in der Lage sein, ein Element $t \in T$ anzugeben und für dieses Element einen Beweis a für $A[t/x]$ zu konstruieren. Damit besteht der gesamte Beweis im wesentlichen aus dem Paar $\langle t\,,a\rangle$, wobei $t \in T$ und $a \in A[t/x]$ gilt. Das bedeutet, daß zur Darstellung der existentiellen Quantifizierung ein *abhängiger Produktraum* genau das geeignete Konstrukt ist. Auch hier zeigen die Regeln wieder die genauen Zusammenhänge, wobei für die Typzugehörigkeit im ersten Teilziel von `pairI` dasselbe gilt wie im Falle von `functionE` und anstelle der Eigenvariablenbedingung in `ex_e` i wiederum das einfachere Kriterium bei der typentheoretischen Regel `productE` i verwendet werden kann.

$\Gamma \vdash \exists x{:}T\,.\,A$
by `ex_i` t
$\Gamma \vdash A[t/x]$

$\Gamma, \exists x{:}T\,.\,A, \Delta \vdash C$
by `ex_e` i **
$\Gamma, x'{:}T, A[x'/x], \Delta \vdash C$

$\Gamma \vdash x{:}T\times A$ ⌊**ext** $\langle t,a\rangle$⌋
by `pairI 1` t `THEN ...`
$\Gamma \vdash t \in T$ ⌊Ax⌋
$\Gamma \vdash A[t/x]$ ⌊**ext** a⌋

$\Gamma,\ z{:}\ x{:}T\times A,\ \Delta \vdash C$ ⌊**ext** let $\langle x',a\rangle = z$ in u⌋
by `productE` i `THEN thin` i
$\Gamma,\ x'{:}T,\ a{:}A[x'/x],\ \Delta \vdash C$ ⌊**ext** u⌋

Damit lassen sich alle logischen Operatoren mit Ausnahme der Falschheit Λ durch die bereits bekannten Konstrukte Funktionenraum, Produkt und Summe innerhalb der Typentheorie repräsentieren. Die so erweiterte Curry-Howard Isomorphie ist eine weitere praktische Rechtfertigung des Prinzips, Propositionen innerhalb der Typentheorie als Datentypen anzusehen. Dieses ermöglicht zudem eine einheitliche Behandlung von Logik und Berechenbarkeit in der intuitionistischen Typentheorie. Wir müssen also nur noch eine geeignete Beschreibung eines leeren Datentyps finden und erhalten dann den Kalkül der Prädikatenlogik mehr oder weniger umsonst.

3.3.2 Der leere Datentyp

Um logische Falschheit darzustellen, benötigen wir, wie soeben angedeutet, einen Datentyp, der im wesentlichen der leeren Menge entspricht. Prinzipiell gibt uns der Gleichheitstyp die Möglichkeit, diesen Datentyp, den wir mit dem Namen void bezeichnen, als konservative Erweiterung der bisherigen Theorie einzuführen, indem wir void auf einem Datentyp abstützen, von dem wir *wissen*, daß er leer sein muß. So könnte man zum Beispiel definieren

$$\mathsf{void} \;\equiv\; \mathtt{X}{:}\mathsf{U}_1 \;\rightarrow\; \mathtt{X} = (\mathtt{X}\rightarrow\mathtt{X}) \in \mathsf{U}_1\,.$$

Dennoch gibt es eine Reihe von Gründen, die dafür sprechen, void explizit einzuführen.

- Das Konzept der logischen Falschheit bzw. des leeren Datentyps ist *primitiv* – also ein grundlegendes mathematisches Konzept – und sollte daher als ein einfacher Typ des Universums U_1 dargestellt werden. Jede Simulation mit den bisherigen Typkonstrukten[38] gehört jedoch mindestens zum Universum U_2.

- Die Besonderheiten des leeren Datentyps, die bei einer Simulation eine untergeordnete Rolle spielen, sollten durch eine explizite Definition von Semantik und Inferenzregeln hervorgehoben werden.

- Der *Gleichheitstyp* besitzt keine nichtkanonischen Ausdrücke, die es erlauben, Gleichheiten zu analysieren. Insbesondere gibt es keine Möglichkeit, in formalen Beweisen die Aussage zu verwenden, daß eine Gleichheit *nicht* gilt. Dies zu dem Gleichheitstyp hinzuzunehmen wäre aber in etwa genauso aufwendig wie eine explizite Definition des Typs void[39] und wesentlich unnatürlicher, denn die Tatsache, daß void keine Elemente enthält, ist das wesentliche Charakteristikum dieses Typs.

[38] Bei Verwendung des Datentyps $\mathbb{Z}$ oder $\mathbb{B}$ wäre natürlich auch eine Simulation innerhalb des ersten Universums möglich, etwa durch $0 = 1 \in \mathbb{Z}$ oder $\mathsf{T} = \mathsf{F} \in \mathbb{B}$.

[39] Die umgekehrte Richtung, Ungleichheit durch Verwendung der logischen Falschheit $\Lambda \equiv \mathsf{void}$ zu definieren, also zum Beispiel durch $s \neq t \in T \;\equiv\; s = t \in T \;\rightarrow\; \mathsf{void}$, kommt dem allgemeinen mathematischen Konsens erheblich näher.

Eine Simulation von void mit dem Typ der Gleichheit ist also möglich, aber aus praktischen Gründen nicht sinnvoll. Es sei jedoch hervorgehoben, daß alle Eigenschaften der Typentheorie, die sich aus der Hinzunahme von void ergeben – wie zum Beispiel der Verlust der starken Normalisierbarkeit – im Prinzip auf dem Gleichheitstyp beruhen bzw. auf der Tatsache, daß logische Propositionen innerhalb der Typentheorie dargestellt werden müssen. Es besteht daher keine Möglichkeit, diese Eigenschaften zu vermeiden.

So zwingt uns zum Beispiel die Hinzunahme eines leeren Datentyps, auch mit Konstrukten wie $\mathsf{void}\times T$, $\mathsf{void}+T$, $T\to\mathsf{void}$ und $\mathsf{void}\to T$ umzugehen. Dabei sind die ersten beiden Typen relativ leicht zu interpretieren. Da void keine Elemente besitzt, muß $\mathsf{void}\times T$ ebenfalls leer und $\mathsf{void}+T$ isomorph zu T sein. Was aber ist mit den anderen beiden Konstrukten?

- $T\to\mathsf{void}$ beschreibt die Menge aller totalen Funktionen von T in den leeren Datentyp void. Wenn nun T nicht leer ist – also ein Element t besitzt – dann würde jede Funktion f aus $T\to\mathsf{void}$ ein Element $f\ t\ \in\ \mathsf{void}$ beschreiben, dessen Existenz aber nun einmal ausgeschlossen sein soll. Folglich darf $T\to\mathsf{void}$ in diesem Fall keine Elemente besitzen. $T\to\mathsf{void}$ muß also leer sein, wann immer T Elemente besitzt.[40] Diese Erkenntnis deckt sich auch mit unserem Verständnis der logischen Falschheit. Eine Aussage der Form $A\Rightarrow\Lambda$ kann nicht beweisbar sein, wenn es einen Beweis für A gibt.

- Weitaus komplizierter ist die Interpretation von Typen der Art $\mathsf{void}\to T$. Solche Typen besitzen in jedem Fall mindestens ein Element – selbst dann, wenn T ein leerer Datentyp ist – denn die Semantik von Funktionen (siehe Abbildung 3.6 auf Seite 112) besagt nun einmal, daß ein Ausdruck $\lambda x.t$ genau dann ein Element von $\mathsf{void}\to T$ ist, wenn das Urteil $t[s/x]\in T$ für alle konkreten Elemente s von void gilt. Da void aber keine konkreten Elemente besitzt, die anstelle der Variablen x eingesetzt werden könnten, ist diese Bedingung in jedem Fall erfüllt. Auch diese Erkenntnis deckt sich mit unserem Verständnis der logischen Falschheit. Eine Aussage der Form $\Lambda\Rightarrow A$ ist immer wahr, denn wenn wir erst einmal gezeigt haben, daß ein Widerspruch (d.h. logische Falschheit) gilt, dann ist jede beliebige Aussage gültig.[41]

  Wie stellen wir dies nun syntaktisch dar? Aus dem oben Gesagten folgt, daß ein Beweisziel der Form

  $$\Gamma,\ z\colon\mathsf{void},\ \Delta\ \vdash\ T$$

  mit einer Regel `voidE` beweisbar sein muß, die – wie die äquivalente logische Regel `false_e` – keine weiteren Teilziele mehr erzeugt.

  Welchen Extrakt-Term aber soll diese Regel generieren? Es ist ein Term, der einerseits von z abhängen sollte und andererseits ein Element eines jeden beliebigen Datentyps T liefert, sofern z ein Element von void ist. Auch wenn wir wissen, daß wir für z niemals einen Term einsetzen können, müssen wir eine syntaktische Beschreibung angeben, welche den obigen Sachverhalt wiederspiegelt. Wir werden hierfür die Bezeichnung $\underline{\mathsf{any}(z)}$ verwenden, um zu charakterisieren, daß hier ein Element eines jeden (englisch "*any*") Typs generiert würde, sofern es gelänge, für z ein Element von void einzusetzen. $\mathsf{any}(z)$ ist folglich ein nichtkanonischer Term, der zum Datentyp void gehört.

Auf den ersten Blick erscheint die Einführung eines nichtkanonischen Terms, der ein Element eines beliebigen Datentyps sein soll, etwas Widersinniges zu sein. Bedenkt man jedoch, daß – entsprechend unserem Beispiel 2.3.25 auf Seite 59 – $\mathsf{T}=\mathsf{F}\in\mathbb{B}$ eine gute Simulation für den Typ void ist, so kann man leicht einsehen, daß die gesamte Typenhierarchie ohnehin kollabieren würde, wenn es gelingen würde, ein Element von void zu finden: alle Typen wären gleich und jeder Term wäre auch ein Element eines jeden Typs. Die Forderung "$\mathsf{any}(z)\in T$ falls $z\in\mathsf{void}$" ist also etwas sehr Sinnvolles.

Die Hinzunahme des Terms $\mathsf{any}(z)$ – so seltsame Eigenschaften auch damit verbunden sind – ändert also nicht im Geringsten etwas an unserer bisherigen Theorie. Sie macht nur deutlich, welche Konsequenzen mit

[40] Ist T dagegen selbst ein leerer Datentyp, so ist es durchaus möglich, daß $T\to\mathsf{void}$ Elemente besitzt. So ist zum Beispiel $\lambda\mathtt{x.x}$ ein zulässiges Element von $\mathsf{void}\to\mathsf{void}$.

[41] Vergleiche hierzu unsere Diskussion auf Seite 35 und das Beispiel 2.3.25 auf Seite 59, in dem wir aus der Gleichheit $\mathsf{F}=\mathsf{T}$ die Gleichheit aller λ-Terme gefolgert haben.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
void{}()		**any**{}(e)
void		any(e)

Zusätzliche Einträge in die Operatorentabelle

Redex	Kontraktum
— *entfällt* —	

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
void = void		
Elementsemantik		
$s = t \in$ void		*gilt niemals !*
void = void $\in \mathsf{U}_j$	falls	$j \geq 1$ *(als natürliche Zahl)*

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash$ void $\in \mathsf{U}_j$ [Ax]
by voidEq

$\Gamma \vdash$ any(s) = any(t) $\in T$ [Ax]
by anyEq
$\Gamma \vdash s = t \in$ void [Ax]

Γ, z: void, $\Delta \vdash C$ [**ext** any(z)]
by voidE i

Inferenzregeln

Abbildung 3.14: Syntax, Semantik und Inferenzregeln des Typs void

dem mathematischen Konzept des Widerspruchs verbunden sind. Es ist klar, daß der Term any(z) in der *Semantik* keine Rolle spielen darf, denn diese drückt ja nur aus, was gültig ist, und enthält in sich keine Widersprüche. Da void keine kanonischen Elemente enthält, wird any(z) auch niemals reduzierbar sein und somit überhaupt nicht zur Semantik eines Ausdrucks beitragen: *wenn Unsinn eingegeben wird, kann auch nichts Sinnvolles herauskommen.*

Insgesamt scheint die angesprochene Interpretation des Typs void ein angemessenes typentheoretisches Gegenstück sowohl zum logischen Konzept der Falschheit als auch zur leeren Menge zu liefern. Wir erweitern daher unsere bisherige Theorie wie folgt (siehe Abbildung 3.14).

- Die Operatorentabelle um zwei neue Operatoren **void**{}() und **any**{}(z) erweitert.
- Die Redex–Kontrakta Tabelle bleibt unverändert, da any(z) nicht reduzierbar ist. Entsprechend gibt es auch keine Reduktionsregel.
- In den Semantiktabellen wird festgelegt, daß void ein Typ (jedes Universums) ohne Elemente ist.[42]
- Die Tatsache daß void keine Elemente hat, wird dadurch widergespiegelt, daß es keine kanonischen Regeln gibt und daß die nichtkanonischen Regeln festlegen, daß any(z) zu jedem Typ gehört, wenn $z \in$ void gilt.

Durch die Hinzunahme des leeren Datentyps wird deutlich, daß die Einbettung von Logik in eine Typentheorie und speziell die Einbettung der Gleichheit – die man ja zur Simulation von void einsetzen könnte – zu einigen Komplikationen in der Theorie führt, derer man sich bewußt sein sollte. So führt die Verwendung von Widersprüchen in den Annahmen (also void oder Λ) zu gültigen Beweisen einiger sehr seltsamer Aussagen.

[42] Der Eintrag 'void = void' bedeutet, daß void ein Typ ist und ein anderer Typ nur dann gleich ist zu void, wenn er sich mittels lässiger Auswertung direkt zu dem Term void reduzieren läßt.

Beispiel 3.3.1

Für $u \equiv \lambda X.\lambda x.\lambda y.y$ können wir $\mathsf{void}\to u \in \mathsf{U}_1$ beweisen, obwohl u selbst überhaupt kein Typ ist. Der formale Beweis arbeitet wie folgt

```
⊢ void→(λX.λx.λy.y) ∈ U1
by functionEq
|\
| ⊢ void ∈ U1
| by voidEq
|  \
|   x:void ⊢ λX.λx.λy.y ∈ U1
|   by voidE 1
```

Als Konsequenz davon ist, daß in vollständigen und korrekten Beweisen durchaus Sequenzen mit unsinnigen Hypothesenlisten auftauchen können wie zum Beispiel `x:void, z:` $\lambda X.\lambda x.\lambda y.y$`, ...`.

Es gibt keinen Weg, derartigen Unsinn in Beweisen auszuschließen. Es sei jedoch angemerkt, daß eine Deklaration der Art `x:void` immer auftreten *muß*, um so etwas zu erzeugen. Dies bedeutet, daß in einem solchen Fall die gesamte Hypothesenmenge ohnehin widersprüchlich ist, und es ist nicht ganz so schmerzhaft, unsinnige Deklarationen als Folge von Annahmen hinzunehmen, die niemals gültig sein können.

Es gibt jedoch noch eine zweite, wesentlich gravierendere Art von Problemen, die Anlaß dafür waren, lässige Auswertung als Reduktionsmechanismus zu verwenden. Es ist nämlich möglich, typisierbare Terme aufzustellen, die einen nichtterminierenden Teilterm besitzen. Ein Beispiel hierfür ist der Term

$$\lambda T.\lambda z.\,(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$$

der sich, wie in Abbildung 3.15 gezeigt, als Element des Typs $T:\mathsf{U}_1\to\mathsf{void}\to T$ nachweisen läßt. Die Konsequenz hiervon ist, daß im Gegensatz zur einfachen Typentheorie eine starke Normalisierbarkeit – im Sinne von "jede beliebige Folge von Reduktionen in beliebigen Teiltermen eines Terms terminiert" – im Allgemeinfall nicht mehr gilt. Es sei aber noch einmal angemerkt, daß dies nicht ein Fehler bei der Beschreibung des Typs void ist, sondern unausweichlich mit der Möglichkeit verbunden ist, Datentypen wie $X:\mathsf{U}_1 \to X = X\to X \in \mathsf{U}_1$ zu formulieren, die keine Elemente besitzen. [43]

Korollar 3.3.2

Eine Typentheorie, in welche die Prädikatenlogik eingebettet werden kann, kann keine stark normalisierbare Reduktionsrelation besitzen.

Diese Tatsache führte im Endeffekt dazu, lässige Auswertung als grundlegende Reduktionsrelation für die (Martin-Löf'sche) Typentheorie zu verwenden. Der Term

$$\lambda z.\,(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$$

ist bereits in kanonischer Form und braucht daher nicht weiter reduziert zu werden. Auch eine Applikation, welche zur Freisetzung des Teilterms $(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$ führen würde, ist nicht durchführbar, da hierfür ein Term $t \in \mathsf{void}$ anstelle von z eingesetzt werden müßte. Einen solchen Term aber kann es nicht geben.

Durch den leeren Datentyp void wird somit der Unterschied zwischen Variablen eines Typs und Termen desselben Typs besonders deutlich. Die Beschreibung einer Funktion $f \equiv \lambda x.b \in S\to T$ suggeriert zwar, daß f Werte $b[x]$ vom Typ T produziert, aber dies kann auch ein Trugschluß sein. Ist nämlich die Eingabevariable vom Typ $S \equiv \mathsf{void}$, so gibt es keine Möglichkeit, für x einen Wert einzusetzen.[44]

[43] Terme wie $\lambda T.\lambda z.\,(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$ tauchen allerdings nicht von selbst in einem Beweis auf. Es ist notwendig, sie explizit als Teil des Beweisziels oder mit der Regel `intro` in den Beweis hineinzunehmen. Wenn wir unsere Theorie auf Terme einschränken, welche als Extrakt-Terme in Beweisen *generiert* werden können, so haben wir nach wie vor die starke Normalisierbarkeit.

[44] Dieser Unterschied erklärt auch die zum Teil sehr vorsichtige und kompliziert erscheinende Beweisführung zu den Eigenschaften der einfachen Typentheorie im Abschnitt 2.4.5. Will man diese Beweise auf allgemeinere Konstrukte übertragen, so muß man berücksichtigen, daß Variablen auch zu einem leeren Typ gehören können.

```
⊢ λT.λz. (λx. x x) (λx. x x) ∈ T:U1→void→T
by lambdaEq 2
|\
| T:U1 ⊢ λz. (λx. x x) (λx. x x) ∈ void→T
| by lambdaEq 1
| |\
| | T:U1, z:void ⊢ (λx. x x) (λx. x x) ∈ T
| | by voidE 2
| |\
|  T:U1 ⊢ void→T ∈ U1
|   by functionEq
|   |\
|   | T:U1 ⊢ void ∈ U2
|   | by voidEq
|   |\
|    T:U1, x:void ⊢ T ∈ U1
|    by hypEq 1
|\
 T:U1→void→T ∈ U2
 by functionEq
 |\
 | ⊢ U1 ∈ U2
 | by univEq
 |\
  T:U1 ⊢ void→T ∈ U2
  by cumulativity 1
  |\
   T:U1 ⊢ void→T ∈ U1
   by functionEq
   |\
   | T:U1 ⊢ void ∈ U2
   | by voidEq
   |\
    T:U1, x:void ⊢ T ∈ U1
    by hypEq 1
```

Abbildung 3.15: Typisierungsbeweis für λ`T.λz. (λx. x x) (λx. x x)`

3.3.3 Konstruktive Logik

Die Hinzunahme des Datentyps void liefert uns das letzte noch fehlende Konstrukt das wir für eine Einbettung der Prädikatenlogik in die Typentheorie mittels des Prinzips "Propositionen als Datentypen" benötigen. Die Regel `voidE` deckt sich mit der entsprechenden Regel `false_e` für die logische Falschheit und damit können alle Regeln des analytischen Sequenzenkalküls für die Prädikatenlogik (siehe Abbildung 2.6 auf Seite 43) als Spezialfälle typentheoretischer Regeln betrachtet werden. Deshalb sind wir in der Lage, die Prädikatenlogik als *konservative Erweiterung* (siehe Abschnitt 2.1.5) zu der bisher definierten Theorie hinzuzunehmen und können auf eine explizite Definition von Semantik und Inferenzregeln verzichten. Zu diesem Zweck führen wir die folgenden definitorischen Abkürzungen ein.[45]

Definition 3.3.3 (Logikoperatoren)

$A \wedge B$	$\equiv$ **and**$\{\}(A;B)$	$\equiv A \times B$
$A \vee B$	$\equiv$ **or**$\{\}(A;B)$	$\equiv A + B$
$A \Rightarrow B$	$\equiv$ **implies**$\{\}(A;B)$	$\equiv A \rightarrow B$
$\neg A$	$\equiv$ **not**$\{\}(A)$	$\equiv A \rightarrow$ void
Λ	$\equiv$ **false**$\{\}()$	$\equiv$ void
$\forall x{:}T\,.\,A$	$\equiv$ **all**$\{\}(T;\, x\,.\,A)$	$\equiv x{:}T \rightarrow A$
$\exists x{:}T\,.\,A$	$\equiv$ **exists**$\{\}(T;\, x\,.\,A)$	$\equiv x{:}T \times A$
$\mathbb{P}_i$	$\equiv \mathbb{P}i$	$\equiv \mathsf{U}_i$

[45] Im Prinzip sind derartige Abkürzungen nichts anderes als textliche Ersetzungen. Innerhalb des NuPRL Systems sind sie allerdings gegen ungewollte Auflösung geschützt. Man muß eine Definition explizit "auffalten" mit der Regel `Unfold ‘name‘ i`, wobei *name* der name der Definition ist (meist der Operatorname) und i die Hypothese, in der die Definition aufgelöst werden soll. Die Konklusion wird als Hypothese 0 angesehen. Eine eventuell nötige Rückfaltung wird mit `Fold ‘name‘ i` erzeugt. Die logischen Regeln aus Abbildung 2.6 wurden mit diesen zwei Regeln und den entsprechenden typentheoretischen Regeln simuliert.

Wir wollen uns nun die logischen Konsequenzen dieser Definitionen etwas genauer ansehen. Als erstes ist festzustellen, daß das natürliche Gegenstück der typentheoretischen Konstrukte die *intuitionistische* Prädikatenlogik ist. Alle Regeln des intuitionistischen Sequenzenkalküls haben ein natürliches Gegenstück in der Typentheorie, aber die klassische Negationsregel `classical_contradiction` gilt im Allgemeinfall nicht, da die Formel $\vdash \neg(\neg A) \Rightarrow A$ nicht für alle Formeln A beweisbar ist. Ein Beweisversuch für das typentheoretische Gegenstück – also $\vdash ((A \to \mathsf{void}) \to \mathsf{void}) \to A$ – würde erfolglos stecken bleiben.

> Der einzig mögliche erste Schritt wäre `lambdaI 1`, was zu $\texttt{f:}(A \to \mathsf{void}) \to \mathsf{void} \vdash A$ führt. Da A beliebig sein soll, können wir kein Element von A direkt angeben. Wir müssen also `functionE 1` s anwenden und hierzu als Argument für f einen Term $s \in A \to \mathsf{void}$ angeben. Dies aber ist nicht möglich, da $A \to \mathsf{void}$ – wie oben argumentiert – leer sein muß, wenn A Elemente haben (also wahr sein) soll. Damit bleibt uns keine Möglichkeit, den Beweis ohne spezielle Kenntnisse über A zu Ende zu führen.

Die Curry-Howard Isomorphie und das Prinzip "Propositionen als Datentypen" führen also zu einer konstruktiven Logik, obwohl wir uns bei der Erklärung der Semantik der Typentheorie nicht auf eine intuitionistische Denkweise abgestützt haben. Die Tatsache, daß wir zur Semantikdefinition nur Elemente benutzt haben, die man explizit angeben kann, reicht aus, der Typentheorie einen konstruktiven Aspekt zu verleihen.

Satz 3.3.4

Die Logik, welche durch das Prinzip "Propositionen als Datentypen" definiert wird, ist die intuitionistische Prädikatenlogik ($\underline{\mathcal{J}}$).

Aufgrund von Theorem 3.3.4 sind wir in der Lage, die Eigenschaften der intuitionistischen Prädikatenlogik $\mathcal{J}$ und des intuitionistischen Sequenzenkalküls $\mathcal{LJ}$ mit den Mitteln der Typentheorie genauer zu untersuchen. Einige sehr interessante Metatheoreme über $\mathcal{LJ}$, die ursprünglich recht aufwendig zu beweisen waren, erhalten dadurch erstaunlich einfache Beweise. Wir werden hierfür einige Beispiele geben.

Die Schnittregel `cut` ist im Sequenzenkalkül ein sehr wichtiges Hilfsmittel zur Strukturierung von Beweisen durch Einfügung von Zwischenbehauptungen (Lemmata). Für eine interaktive Beweiskonstruktion ist diese Regel unumgänglich. Für eine automatische Suche nach Beweisen stellt sie jedoch ein großes Hindernis dar. Außerdem taucht diese Regel im Kalkül des natürlichen Schließens nicht auf, was den Beweis der Äquivalenz der beiden Kalküle erheblich erschwert. Es ist jedoch möglich, Anwendungen der Schnittregel aus Beweisen zu eliminieren, wobei man allerdings ein eventuell exponentielles Wachstum der Beweise in Kauf nehmen muß. Dieser in der Logik sehr aufwendig zu beweisende Satz (Genztens *Hauptsatz* in [Gentzen, 1935]) läßt sich in der Typentheorie durch eine einfache Verwendung der Reduktion beweisen.

Satz 3.3.5 (Schnittelimination)

Wenn $\Gamma \vdash C$ innerhalb von $\mathcal{LJ}$ bewiesen werden kann, dann gibt es auch einen Beweis ohne Verwendung der Schnittregel.

Beweis: Wenn wir C als Typ betrachten und innerhalb der Typentheorie mit den zu $\mathcal{LJ}$ korrespondierenden Regeln beweisen, dann können wir aus dem Beweis einen Term t extrahieren, der genau beschreibt, welche Regeln zum Beweis von C in welcher Reihenfolge angewandt werden müssen. Da für extrahierte Terme innerhalb der Typentheorie starke Normalisierbarkeit gilt (siehe Fußnote Seite 133), können wir t so lange reduzieren, bis keine Redizes mehr vorhanden sind. Der resultierende Term t' liefert ebenfalls einen Beweis für C, der aber keine Schnittregel `cut` mehr enthalten kann, da diese im Extrakt-Term Teilterme der Art $(\lambda x.u)\ s$ erzeugt. □

Das Schnitteliminationstheorem erleichtert auch einen Beweis für die *Konsistenz* des Kalküls $\mathcal{LJ}$, also die Tatsache, daß nicht gleichzeitig eine Aussage und ihr Gegenteil beweisen werden kann.[46]

Satz 3.3.6

$\mathcal{LJ}$ ist konsistent.

[46] Die logische Konsistenz eines Kalküls kann unabhängig von einer konkreten Semantik überprüft werden und ist daher etwas schwächer als *Korrektheit* (*soundness*). Ist der Kalkül allerdings auch *vollständig*, so fallen die beiden Begriffe zusammen.

Beweis: Wenn es auch nur eine einzige Formel A gäbe, für die in $\mathcal{LJ}$ sowohl ein Beweis für $\vdash A$ als auch für $\vdash \neg A$ geführt werden könnte, dann könnte man $\vdash \Lambda$ wie folgt beweisen:

```
⊢ Λ
by cut 1 A
|\
| ⊢ A
| by  – fester Beweis –
  \
   x:A ⊢ Λ
   by cut 2 ¬A
   |\
   | ⊢ ¬A
   | by  – fester Beweis –
     \
      x:A, y:¬A ⊢ Λ
      by functionE_indep 2          (Unfold 'not' 2)
      |\
      | x:A, y:¬A ⊢ A
      | by hypEq 1
        \
         x:A, y:¬A, z:void ⊢ Λ
         by voidE 3                 (Unfold 'false' 0)
```

Nach Theorem 3.3.5 gäbe es dann aber auch einen Beweis für $\vdash \Lambda$, der ohne Schnitte auskommt. Da jedoch alle anderen Regeln auf dieses Ziel nicht anwendbar sind (es gibt keine Regel `falseI`), kann dies nicht stimmen. Also kann es keine einander widersprechende Aussagen geben, die beide in $\mathcal{LJ}$ beweisbar sind. □

Die Isomorphie zwischen der Typentheorie und der konstruktiven Logik ist übrigens nicht beschränkt auf Logik erster Stufe. Die Kombination von Universenhierarchie und abhängigem Funktionenraum ermöglicht die formale Darstellung aller Quantoren höherer Stufe. Insbesondere erhalten wir

- Einen Funktionalkalkül höherer Ordnung, in dem über beliebige Funktionen quantifiziert werden darf wie zum Beispiel in $\forall \mathtt{f}:S{\to}T\,.\,t[\mathtt{f}]$. (Stufe ω)
- Die Prädikatenlogik beliebiger fester Stufe `i+1`, in der über beliebige Prädikatsymbole der Stufe `i` quantifiziert werden darf wie zum Beispiel in $\forall \mathtt{P}:\mathbb{P}_{\mathtt{i}}\,.\,t[\mathtt{P}]$.[47]

3.3.4 Klassische Logik

Soweit die intuitionistische Logik. Wie aber sieht es mit der klassischen Logik aus, in der Theoreme bewiesen werden können, die nicht unbedingt konstruktiv gelten? Können wir diese ebenfalls in der Typentheorie darstellen? Die Lösung hierfür ist einfach, da die klassische Logik sich in die intuitionistische einbetten läßt, indem wir einfach jede Formel durch doppelte Negation ihres konstruktiven Anteils berauben. Wir definieren hierzu die klassischen logischen Operatoren wie folgt.

Definition 3.3.7 (Klassische Logikoperatoren)

Λ	$\equiv$ **false**{}()	$\equiv$ void
$\neg A$	$\equiv$ **not**{}(A)	$\equiv A{\to}$void
$A \wedge B$	$\equiv$ **and**{}(A;B)	$\equiv A{\times}B$
$\forall x{:}T\,.\,A$	$\equiv$ **all**{}(T; $x\,.\,A$)	$\equiv x{:}T{\to}A$
$A \vee_c B$	$\equiv$ **orc**{}(A;B)	$\equiv \neg(\neg A \wedge \neg B)$
$A \Rightarrow_c B$	$\equiv$ **impc**{}(A;B)	$\equiv \neg A \vee_c B$
$\exists_c x{:}T\,.\,A$	$\equiv$ **exc**{}(T; $x\,.\,A$)	$\equiv \neg(\forall x{:}T\,.\,\neg A)$

[47] Es gibt Versuche, dies auf Stufe ω auszuweiten, in der die Stufe des Prädikats nicht mehr verwaltet werden muß. Diese bergen jedoch einige Schwierigkeite in sich.

Wie man sieht, bleiben Negation, Konjunktion und Allquantor unverändert während die anderen drei Operatoren implizit schon die nichtkonstruktive Sicht wiederspiegeln. Mit dieser Übersetzung der klassischen Operatoren in die intuitionistischen, deren homomorphe Fortsetzung auf Formeln und Sequenzen wir mit $^\circ$ bezeichnen (die sogenannte *Gödel-Transformation*), läßt sich folgendes Einbettungstheorem beweisen.

Satz 3.3.8

Es sei $'$ die Transformation, welche in einer Formel bzw. Sequenz jede atomare Teilformel A durch $\neg\neg A$ ersetzt. Wenn innerhalb des klassischen Sequenzenkalküls die Sequenz $\Gamma \vdash C$ bewiesen werden kann, dann gibt es für $\Gamma^{\circ\prime} \vdash C^{\circ\prime}$ einen Beweis im intuitionistischen Sequenzenkalkül $\mathcal{LJ}$.

Statt eines Beweises von Theorem 3.3.8 wollen wir die Gültigkeit der (transformierten) klassischen Regel `classical_contradiction` nachweisen, welche die Grundlage aller Widerspruchsbeweise ist.

$\Gamma \vdash C^{\circ\prime}$ **by** `classical_contradiction`
$\Gamma,\ \neg C^{\circ\prime} \vdash \Lambda$

Wir skizzieren einen $\mathcal{LJ}$-Beweis für den Fall, daß C atomar – also $C^{\circ\prime} \equiv \neg\neg C$ – ist.

```
Γ ⊢ C°'
by not_i                          (mit Unfold)
 \
  Γ, ¬C ⊢ Λ
  by cut -1 ¬C°'
  |\
  | Γ, ¬C ⊢ ¬C°'
  | by not_i                      (mit Unfold)
  |  \
  |   Γ, ¬C, ¬¬C ⊢ Λ
  |   by not_e -1
  |   |\
  |   | Γ, ¬C, ¬¬C ⊢ ¬C
  |   | by hypothesis -2
  |    \
  |     Γ, ¬C, Λ ⊢ Λ
  |     by false_e -1
   \
    Γ, ¬C, ¬C°'  ⊢ Λ
    by thin -1
    Γ, ¬C°'  ⊢ Λ
```

Auf ähnliche Weise ließe sich übrigens das Gesetz vom Ausgeschlossenen Dritten $\vdash C \vee_c \neg C$ für beliebige Formeln C direkt nachweisen, was die Transformation $'$ in den meisten Fällen überflüssig macht.

Damit kann auch die klassische Logik in der Typentheorie behandelt werden. Man muß sich bei der Wahl der logischen Operatoren allerdings festlegen, ob man ein klassisches oder ein intuitionistisches Verständnis der Formel anlegen möchte. Im Gegensatz zu anderen Kalkülen sind Mischformen ebenfalls möglich.

3.4 Programmierung in der Typentheorie

Bei unseren bisherigen Betrachtungen der Typentheorie haben stand vor allem die Ausdruckskraft im Vordergrund. Mit den vorgestellten Konstrukten können wir einen Großteil der Mathematik und Programmierung formal repräsentieren und Eigenschaften von Programmen und mathematischen Konstrukten formal nachweisen. Wir wollen nun beginnen, Konzepte zu diskutieren, welche eine effiziente *Anwendung* der Typentheorie unterstützen. Hierzu gehört insbesondere der Aspekt der *Programmentwicklung*, den wir in diesem Abschnitt besprechen wollen.

$T \in \mathsf{U}_j$	$t \in T$	T [**ext** t]
T ist ein Typ (eine Menge)	t ist ein Element der Menge T	T ist nicht leer *(inhabited)*
T ist eine Proposition	t ist ein(e) Beweis(konstruktion) für T	T ist wahr (beweisbar)
T ist eine Intention	t ist eine Methode, um T zu erfüllen	T ist erfüllbar (realisierbar)
T ist ein Problem (Aufgabe)	t ist eine Methode, um T zu lösen	T ist lösbar

Abbildung 3.16: Interpretationen der typentheoretischen Urteile

3.4.1 Beweise als Programme

Für die typentheoretischen Urteile bzw. ihre Darstellung durch Sequenzen kann man – je nachdem, welche Problemstellung man mit formalen Theorien angehen möchte – eine Reihe verschiedener semantischer Interpretationen geben. Wir haben die wichtigsten Sichtweisen in Abbildung 3.16 zusammengestellt. Die erste Zeile beschreibt die Lesart, die wir bei der Einführung der Typentheorie als Motivation verwandt haben: Typen sind Mengen mit Struktur und Elementen. Die zweite Zeile entspricht einer *logischen Sicht*, die – unterstützt durch die Curry-Howard Isomorphie – zu dem Prinzip "Propositionen als Datentypen" geführt hatte. Die dritte, eher philosophische Sichtweise geht zurück auf Heyting [Heyting, 1931] und soll hier nicht vertieft werden.

Die letzte Interpretation wurde von Kolmogorov [Kolmogorov, 1932] vorgeschlagen, der die Mathematik als eine Sammlung von Problemstellungen verstand, für die es *Lösungsmethoden* zu konstruieren galt. In der Denkweise der Programmierung würde man heute den Begriff "*Spezifikation*" anstelle von "Problem" und das Wort "*Programm*" anstelle von "Methode" verwenden. Damit würde '$t \in T$' interpretiert als "t ist ein Programm, welches die Spezifikation T löst" und das Beweisziel '$\vdash T$ [**ext** t]' könnte verstanden werden als die *Aufgabe, ein Programm zu konstruieren*, welches eine gegebene Spezifikation T erfüllt. Der Kalkül der Typentheorie läßt sich daher prinzipiell zur Konstruktion von Programmen aus ihren Spezifikationen verwenden. Wir wollen nun untersuchen, wie dies geschehen kann, und betrachten hierzu ein einfaches Beispiel.

Beispiel 3.4.1

Wir nehmen einmal an, wir hätten innerhalb der Typentheorie ein Theorem beweisen, welches besagt, daß jede natürliche Zahl eine ganzzahlige Quadratwurzel besitzt.[48]

$$\vdash\ \forall \mathtt{x}:\mathbb{N}.\exists \mathtt{y}:\mathbb{N}.\ \mathtt{y}^2 \leq \mathtt{x}\ \wedge\ \mathtt{x} < (\mathtt{y}+1)^2$$

Dann liefert der Beweis dieses Theorems einen Extrakt-Term p, den wir als Evidenz für seine Gültigkeit verstehen können. Setzen wir nun anstelle der logischen Operatoren die entsprechenden Typkonstruktoren ein, durch die sie definiert wurden (vergleiche Definition 3.3.3), so sehen wir, daß p den Typ

$$\mathtt{x}:\mathbb{N} \rightarrow \mathtt{y}:\mathbb{N} \times \mathtt{y}^2 \leq \mathtt{x} \times \mathtt{x} < (\mathtt{y}+1)^2$$

hat – wobei wir für $\leq$ und $<$ vorerst keine Interpretation geben. Damit ist p also eine Funktion, die eine natürliche Zahl n in ein Tupel $\langle y, \langle pf_1, pf_2\rangle\rangle$ abbildet, wobei y die Integerquadratwurzel von n ist und pf_1 sowie pf_2 Beweise für $y^2 \leq n$ bzw. $n < (y+1)^2$ sind. p enthält also sowohl den Berechnungsmechanismus für die Integerquadratwurzel als auch einen allgemeinen Korrektheitsbeweis für diesen Mechanismus. Beide Anteile kann man voneinander durch Anwendung des spread-Operators trennen und erhält somit ein allgemeines Programm zur Berechnung der Integerquadratwurzel durch

$$\mathtt{sqrt} \equiv \lambda \mathtt{n}.\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = p\,\mathtt{n}\ \mathsf{in}\ \mathtt{y}.$$

Die Korrektheit dieses Programms folgt unmittelbar aus dem obigen Theorem.

Dies bedeutet also, daß uns ein Beweis für eine mathematische Aussage der Form $\forall x:T.\ \exists y:S.\ spec[x,y]$ – ein sogenanntes *Spezifikationstheorem* – automatisch ein Programm liefert, welches die Spezifikation *spec* erfüllt. Wir können die Typentheorie daher nicht nur für die mathematische Beweisführung sondern auch für die Entwicklung von Programmen mit garantierter Korrektheit verwenden. Diese Tatsache läßt sich sogar *innerhalb* der Typentheorie nachweisen: wenn ein Spezifikationstheorem der Form $\forall x:T.\ \exists y:S.\ spec[x,y]$ beweisbar ist, dann gibt es auch eine Funktion $f:T \rightarrow S$, welche die Spezifikarion *spec* erfüllt.

[48] Die hierzu nötigen Konstrukte werden wir im Abschnitt 3.4.2 und einen Beweis im Beispiel 3.4.9 auf Seite 154 vorstellen.

Satz 3.4.2 (Konstruktives Auswahlaxiom / Beweise als Programme)

Für alle Universen U_j *kann das folgende Theorem in der Typentheorie beweisen werden.*

$\vdash\ \forall \mathtt{T}:\mathsf{U}_j\,.\,\forall \mathtt{S}:\mathsf{U}_j\,.\,\forall \mathtt{spec}:\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j\,.\ (\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)\ \Rightarrow\ \exists \mathtt{f}:\mathtt{T}\to\mathtt{S}.\forall \mathtt{x}:\mathtt{T}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{f}\,\mathtt{x}\rangle$

Beweis: Der formale Beweis spiegelt genau das Argument aus Beispiel 3.4.1 wieder.
Der Übersichtlichkeit halber werden wir ihn nur skizzieren und dabei Wohlgeformtheitsziele unterdrücken.

$\vdash\ \forall \mathtt{T}:\mathsf{U}_j\,.\,\forall \mathtt{S}:\mathsf{U}_j\,.\,\forall \mathtt{spec}:\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j\,.\ (\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)\ \Rightarrow\ \exists \mathtt{f}:\mathtt{T}\to\mathtt{S}.\forall \mathtt{x}:\mathtt{T}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{f}\,\mathtt{x}\rangle$
by `Repeat all_i` *(d.h.* `function_i --` *3 mal)*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$ $\vdash$ $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)\ \Rightarrow\ \exists \mathtt{f}:\mathtt{T}\to\mathtt{S}.\forall \mathtt{x}:\mathtt{T}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{f}\,\mathtt{x}\rangle$
by `imp_i` *(d.h.* `function_i` *mit Variable)*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$ $\vdash$ $\exists \mathtt{f}:\mathtt{T}\to\mathtt{S}.\forall \mathtt{x}:\mathtt{T}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{f}\,\mathtt{x}\rangle$
by `ex_i` $\lambda \mathtt{x}.\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{y}$ *(d.h.* `pair_i` *)*

| `T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$ $\vdash$ $\lambda \mathtt{x}.\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{y}\ \in\ \mathtt{T}\to\mathtt{S}$
| **by** *– Wohlgeformtheitsbeweis –*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$
$\vdash$ $\forall \mathtt{x}:\mathtt{T}.\ \mathtt{spec}\,\langle \mathtt{x},(\lambda \mathtt{x}.\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{y})\,\mathtt{x}\rangle$
by *– explizite Reduktion, siehe Abschnitt 3.6*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$ $\vdash$ $\forall \mathtt{x}:\mathtt{T}.\ \mathtt{spec}\,\langle \mathtt{x},\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{y}\rangle$
by `all_i`

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$, `x`:`T` $\vdash$ $\mathtt{spec}\,\langle \mathtt{x},\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{y}\rangle$
by `intro` $\mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{pf}$ *(Siehe Abbildung 3.13, Seite 126)*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$, `x`:`T`
$\vdash$ $\mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{pf}\ \in\ \mathtt{spec}\,\langle \mathtt{x},\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{p}\,\mathtt{x}\ \mathsf{in}\ \mathtt{y}\rangle$
by `spreadEq z` $\mathtt{spec}\,\langle \mathtt{x},\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \mathtt{z}\ \mathsf{in}\ \mathtt{y}\rangle$ $\mathtt{y}:\mathtt{S}\times\mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle$

| `T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$, `x`:`T` $\vdash$ $\mathtt{p}\,\mathtt{x}\ \in\ \mathtt{y}:\mathtt{S}\times\mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle$
| **by** *– Wohlgeformtheitsbeweis mit Hypothese 4 –*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$, `x`:`T`, `s`:`S`,
`t`:$\mathtt{spec}\,\langle \mathtt{x},\mathtt{s}\rangle$, `z`: $\mathtt{p}\,\mathtt{x}=\langle \mathtt{s},\mathtt{t}\rangle\ \in\ \mathtt{y}:\mathtt{S}\times\mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle$ $\vdash$ $\mathtt{t}\ \in\ \mathtt{spec}\,\langle \mathtt{x},\ \mathsf{let}\ \langle \mathtt{y},\mathtt{pf}\rangle = \langle \mathtt{s},\mathtt{t}\rangle\ \mathsf{in}\ \mathtt{y}\rangle$
by *– explizite Reduktion, siehe Abschnitt 3.6*

`T`:U_j, `S`:U_j, `spec`: $\mathtt{T}\times\mathtt{S}\to\mathbb{P}_j$, `p`: $(\forall \mathtt{x}:\mathtt{T}.\ \exists \mathtt{y}:\mathtt{S}.\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle)$, `x`:`T`, `s`:`S`,
`t`:$\mathtt{spec}\,\langle \mathtt{x},\mathtt{s}\rangle$, `z`: $\mathtt{p}\,\mathtt{x}=\langle \mathtt{s},\mathtt{t}\rangle\ \in\ \mathtt{y}:\mathtt{S}\times\mathtt{spec}\,\langle \mathtt{x},\mathtt{y}\rangle$ $\vdash$ $\mathtt{t}\ \in\ \mathtt{spec}\,\langle \mathtt{x},\mathtt{s}\rangle$
by `hypEq 7`

□

In der Mengentheorie ist die Aussage des Theorems 3.4.2 als eine Variante des *Auswahlaxioms* bekannt geworden. In keiner Formulierung der (nichtkonstruktiven) Mengentheorie (z.B. Zermelo-Fraenkel) ist es möglich, diese Aussage zu beweisen, und man müß sie deshalb immer als Axiom zur Theorie hinzunehmen. Die Rechtfertigung dieses Axioms ist allerdings nicht sehr einfach. In der Typentheorie ist die Aussage dagegen ein Theorem, welches unmittelbar aus der Darstellung von Propositionen als Datentypen folgt und uns eine generelle Methode zur Entwicklung von Programmen aus gegebenen Spezifikationen liefert.

Entwurfsprinzip 3.4.3 (Beweise als Programme)

In der Typentheorie ist Programmentwicklung dasselbe wie Beweisführung

Dieses Prinzip, welches unter dem Namen "*proofs as programs*" [Bates & Constable, 1985] bekannt geworden ist, ermöglicht es, auf systematische Weise Programme zu entwickeln, die *garantiert korrekt sind*: man muß einfach nur einen Beweis für ein Spezifikationstheorem führen und hieraus ein Programm extrahieren. Auf den ersten Blick scheint diese Methode unnatürlich und sehr ineffizient zu sein – was hat Programmierung mit dem Nachweis der Erfüllbarkeit einer Spezifikation zu tun? Es hat sich jedoch herausgestellt, daß nahezu

alle Programmiermethoden, die von Systematikern wie Hoare [Hoare, 1972], Dijkstra [Dijkstra, 1976] und anderen aufgestellt wurde, im wesentlichen nichts anderes sind als spezielle Neuformulierungen von bekannten Beweismethoden. Daher ist es durchaus legitim, Programmierung als eine Art Spezialfall von Beweisführung anzusehen und Entwurfstechniken, die ursprünglich von Methodikern wie Polya [Polya, 1945] als mathematische Methoden für Beweise vorgeschlagen wurden, auf die Programmentwicklung zu übertragen.[49] Hierfür ist es allerdings notwendig, Konstrukte in die Typentheorie hineinzunehmen, die notwendig sind, um "echte" Mathematik und Programmierung zu beschreiben. Dies wollen wir nun in den folgenden Abschnitten tun.

3.4.2 Zahlen und Induktion

In der praktischen Mathematik und den meisten "realen" Programmen spielen arithmetische Konstrukte eine zentrale Rolle. Aus diesem Grunde ist es notwendig, das Konzept der Zahl und die elementare Arithmetik – einschließlich der primitiven Rekursion und der Möglichkeit einer induktiven Beweisführung – in die Typentheorie mit aufzunehmen. Wir hatten in Abschnitt 2.3.3 auf Seite 54 bereits gezeigt, daß natürliche Zahlen prinzipiell durch Church Numerals simuliert werden können. Eine solche Simulation würde aber bereits die einfachsten Konstrukte extrem aufwendig werden lassen und ein formales Schließen über Arithmetik praktisch unmöglich machen. Zugunsten einer effizienten Beweisführung ist es daher notwendig, Zahlen und die wichtigsten Grundoperationen sowie die zugehörigen Inferenzregeln explizit einzuführen.

Wir werden dies in zwei Phasen tun. Zunächst werden wir eine eher spartanische Variante der Arithmetik – die *primitiv rekursive Arithmetik* PRA – betrachten, die nur aus den natürlichen Zahlen, der Null, der Nachfolgerfunktion und der primitiven Rekursion besteht. Dies ist aus theoretischer Sicht der einfachste Weg, die Typentheorie um eine Arithmetik zu erweitern, und ermöglicht es, die Grundeigenschaften einer solchen Erweiterung gezielt zu erklären. Aus praktischen Gesichtspunkten ist es jedoch sinnvoller, die Typentheorie um eine erheblich umfangreichere Theorie der Arithmetik – die konstruktive Peano Arithmetik oder *Heyting Arithmetik* HA – zu erweitern. Diese ist bezüglich ihrer Ausdruckskraft genauso mächtig wie die primitiv rekursive Arithmetik. Jedoch werden ganze Zahlen anstelle der natürlichen Zahlen betrachtet, die Zahlen wie `0, 1, 2, ...` und eine Fülle von elementaren arithmetischen Operationen wie `+, -, *, ÷, rem, <` und arithmetische Fallunterscheidungen sind bereits vordefiniert und das Regelsystem ist entsprechend umfangreich. Die Hinzunahme von Zahlen in die Typentheorie von NuPRL gestaltet sich daher verhältnismäßig aufwendig.

3.4.2.1 Die primitiv rekursive Arithmetik

Die primitiv rekursive Arithmetik besteht aus den minimalen Bestandteilen, die zum Aufbau einer arithmetischen Theorie notwendig sind. Als Typkonstruktor wird der Typ $\underline{\mathbb{N}}$[50] der natürlichen Zahlen eingeführt, der als kanonische Elemente die Null ($\underline{\mathsf{0}}$) besitzt und alle Elemente, die durch Anwendung der Nachfolgerfunktion $\underline{\mathsf{s}}$:$\mathbb{N}\to\mathbb{N}$ auf kanonische Elemente entstehen. Die zugehörige nichtkanonische Form analysiert den induktiven Aufbau einer natürlichen Zahl aus `0` und `s` und baut hieraus rekursiv einen Term zusammen: der Term $\underline{\mathsf{ind}(n;\, base;\, m, f_m.t)}$ beschreibt die Werte einer Funktion f, die bei Eingabe der Zahl n=`0` das Resultat *base* und bei Eingabe von n=`s`(m) den Term $t[m, f_m]$ liefert, wobei f_m ein Platzhalter für $f(m)$ ist. Dieser nichtkanonische Term ist also ein typentheoretisches Gegenstück zur primitiven Rekursion.

Die Semantik der obengenannten Terme ist relativ leicht zu formalisieren. Entsprechend der intuitiven Erklärung gibt es zwei Arten, den Induktionsterm zu reduzieren.

$$\begin{array}{lcl} \mathsf{ind}(\mathsf{0};\, base;\, m, f_m.t) & \xrightarrow{\beta} & base \\ \mathsf{ind}(\mathsf{s}(n);\, base;\, m, f_m.t) & \xrightarrow{\beta} & t[n, \mathsf{ind}(n;\, base;\, m, f_m.t)\ /\ m,\ f_m] \end{array}$$

[49] Dieser Zusammenhang wurde von David Gries [Gries, 1981] herausgestellt. Ein Beispiel für die Vorteile dieser Sichtweise haben wir bereits in Abschnitt 1.1 vorgestellt.

[50] Auf die Darstellung in der einheitlichen Syntax wollen wir an dieser Stelle verzichten, da wir anstelle der natürlichen Zahlen die ganzen Zahlen als Grundkonstrukt zur Typentheorie hinzunehmen. Die natürlichen Zahlen können hieraus mit dem Teilmengenkonstruktor, den wir im Abschnitt 3.4.4 vorstellen werden, als definitorische Erweiterung gebildet werden.

Unter den kanonischen Termen ist $\mathbb{N}$ ein Typ, der nur identisch ist mit sich selbst (d.h. es gilt das Urteil '$\mathbb{N}$=$\mathbb{N}$') und 0 ein Element von $\mathbb{N}$, das nur mit sich selbst identisch ist. Die Nachfolger zweier natürlicher Zahlen n und m sind genau dann gleich, wenn n und m semantisch gleich sind. In den formalen Semantiktabellen wären also die folgenden Einträge zu ergänzen.

Typsemantik
$\mathbb{N} = \mathbb{N}$

Elementsemantik	
$0 = 0 \in \mathbb{N}$	
$\mathsf{s}(n) = \mathsf{s}(m) \in \mathbb{N}$	falls $n = m \in \mathbb{N}$
$\mathbb{N} = \mathbb{N} \in \mathsf{U}_j$	

Man beachte, daß hieraus folgt, daß $\mathsf{s}(n)=0 \in \mathbb{N}$ *nicht* gelten kann, da es keinen Tabelleneintrag gibt, der dises Urteil unterstützt. Ebenso gilt, daß aus $\mathsf{s}(n)=\mathsf{s}(m) \in \mathbb{N}$ auch $n=m \in \mathbb{N}$ folgen muß, weil es keinen anderen Weg gibt, die Gleichheit von $\mathsf{s}(n)$ und $\mathsf{s}(m)$ semantisch zu begründen. Diese beiden Grundeigenschaften der natürlichen Zahlen müssen daher nicht explizit in den Tabellen aufgeführt werden.
Die zugehörigen Inferenzregeln müssen die soeben gegebene Semantik respektieren.

- Als Formationsregel erhalten wir somit, daß $\mathbb{N}$ ein Typ jeden Universums ist, da an die Typeigenschaft von $\mathbb{N}$ ja keine Anforderungen gestellt wurden.

  $\Gamma \vdash \mathbb{N} \in \mathsf{U}_j$ [Ax]
  by `natEq`

- Die kanonischen Gleichheitsregeln spiegeln genau die obigen Gleichheitsurteile wieder. Die entsprechenden impliziten Varianten zur Erzeugung kanonischer Terme ergeben sich direkt aus diesen.

  $\Gamma \vdash 0 \in \mathbb{N}$ [Ax]
  by `zeroEq`

  $\Gamma \vdash \mathbb{N}$ [**ext** 0]
  by `zeroI`

  $\Gamma \vdash \mathsf{s}(t_1) = \mathsf{s}(t_2) \in \mathbb{N}$ [Ax]
  by `sucEq`
  $\Gamma, \vdash t_1 = t_2 \in \mathbb{N}$ [Ax]

  $\Gamma \vdash \mathbb{N}$ [**ext** $\mathsf{s}(t)$]
  by `sucI`
  $\Gamma \vdash \mathbb{N}$ [**ext** t]

- In den nichtkanonischen Regeln legen wir fest, daß zwei Induktionsterme nur gleich sein können, wenn die Hauptargumente gleich sind, die Basisfälle übereinstimmen und die rekursive Abarbeitung sich gleich verhält. Dies deckt natürlich nicht alle Fälle ab, denn die Gleichheit kann ja auch aus anderen Gründen gelten – so wie zum Beispiel $4+7 = 2+9$ ist. Derartige Gleichheiten hängen jedoch von den konkreten Werten ab und können nur durch Auswertung der Argumente und Reduktion (siehe unten) nachgeweisen werden. Die implizite Variante ergibt sich entsprechend.

  $\Gamma \vdash \mathsf{ind}(n_1; base_1; m_1, f_1.t_1) = \mathsf{ind}(n_2; base_2; m_2, f_2.t_2) \in T[n_1/z]$ [Ax]
  by `indEq` z T
  $\Gamma \vdash n_1 = n_2 \in \mathbb{N}$ [Ax]
  $\Gamma \vdash base_1 = base_2 \in T[0/z]$ [Ax]
  $\Gamma, m{:}\mathbb{N}, f{:}T[m/z] \vdash t_1[m, f\,/\,m_1, f_1] = t_2[m, f\,/\,m_2, f_2] \in T[\mathsf{s}(m)/z]$ [Ax]

  $\Gamma, n{:}\mathbb{N}, \Delta \vdash C$ [**ext** $\mathsf{ind}(n; base; m, f.t)$]
  by `natE` i
  $\Gamma, n{:}\mathbb{N}, \Delta \vdash C[0/n]$ [**ext** $base$]
  $\Gamma, n{:}\mathbb{N}, \Delta, m{:}\mathbb{N}, f{:}C[m/n] \vdash C[\mathsf{s}(m)/n]$ [**ext** t]

- Zwei Reduktionsregeln werden benötigt, um die oben angesprochenen Berechnungen zu repräsentieren.

  $\Gamma \vdash \mathsf{ind}(0; base; m, f.t) = t_2 \in T$ [Ax]
  by `indRedBase`
  $\Gamma \vdash base = t_2 \in T$ [Ax]

  $\Gamma \vdash \mathsf{ind}(\mathsf{s}(n); base; m, f.t) = t_2 \in T$ [Ax]
  by `indRedUp`
  $\Gamma \vdash t[n, \mathsf{ind}(n; base; m, f.t)\ /\ m,\ f] = t_2 \in T$ [Ax]

Soweit die Standardregeln des Datentyps $\mathbb{N}$. Was bisher zu fehlen scheint, ist eine Regel, welche die Beweisführung durch Induktion wiederspiegelt: "*Um einer Eigenschaft P für alle natürlichen Zahlen nachzuweisen, reicht es aus, $P(0)$ zu beweisen und zu zeigen, daß aus $P(n)$ immer auch $P(n{+}1)$ folgt*". Dies wäre also eine Regel der folgenden Art.

```
Γ, n:IN, Δ ⊢ P(n)
  by induction n
    Γ, n:IN, Δ ⊢ P(0)
    Γ, n:IN, Δ, P(n) ⊢ P(n+1)
```

Welche Evidenzen werden nun durch diese Regel konstruiert? Nehmen wir einmal an, wir hätten einen Beweisterm b für $P(0)$ im ersten Teilziel. In den Hypothesen des zweiten Teilziels sei x ein Bezeichner für die Annahme $P(n)$. Dann wird ein Beweisterm t für $P(n+1)$ bestenfalls von x und n abhängen können. Für ein beliebiges $m \in \mathbb{N}$ wird also der Beweisterm für $P(m)$ schrittweise aus b und t aufgebaut werden: für $m=0$ ist er b, für $m=1$ ist er $t[0, b\,/\,n, x]$, für $m=2$ ist er $t[1, t[0, b\,/\,n, x]\,/\,n, x]$ usw. Dies aber ist genau der Term, der sich durch $\mathsf{ind}(m; b; n, x.t)$ beschreiben läßt. Damit ist aber die Induktionsregel nichts anderes als ein Spezialfall der Eliminationsregel `natE` für natürliche Zahlen und es ergibt sich folgendes Prinzip:

Entwurfsprinzip 3.4.4 (Induktion)

In der Typentheorie ist induktive Beweisführung dasselbe wie die Konstruktion rekursiver Programme.

Dieses Prinzip vervollständigt das Prinzip "Beweise als Programme" um das Konzept der Rekursion, welches in der Curry-Howard-Isomorphie nicht enthalten war.

Auf der Basis der primitiv rekursiven Arithmetik kann nun im Prinzip die gesamte Arithmetik und Zahlentheorie entwickelt werden. So könnte man zum Beispiel Addition, Vorgängerfunktion (**p**), Subtraktion, Multiplikation, Quadrat und eine Fallunterscheidung mit Test auf Null durch die üblichen aus der Theorie der primitiv rekursiven Funktionen bekannten Formen definieren.

$$
\begin{array}{ll}
n\mathtt{+}m & \equiv \mathsf{ind}(m; n; \mathtt{k,n}_{+k}.\mathbf{s}(\mathtt{n}_{+k})) \\
\mathbf{p}(n) & \equiv \mathsf{ind}(n; 0; \mathtt{k,\_.k}) \\
n\mathtt{-}m & \equiv \mathsf{ind}(m; n; \mathtt{k,n}_{-k}.\mathbf{p}(\mathtt{n}_{-k})) \\
n\mathtt{*}m & \equiv \mathsf{ind}(m; 0; \mathtt{k,n}_{*k}\,.\,\mathtt{n}_{*k}\mathtt{+}n) \\
n^2 & \equiv n\mathtt{*}n \\
\mathsf{if}\ n\mathtt{=}0\ \mathsf{then}\ s\ \mathsf{else}\ t & \equiv \mathsf{ind}(n; s; \_,\_.t)
\end{array}
$$

Die definitorischen Erweiterungen der primitiv rekursiven Arithmetik um die wichtigsten arithmetischen Operationen sind also noch relativ einfach durchzuführen. Es ist jedoch verhältnismäßig mühsam, die bekannten Grundeigenschaften dieser Operationen nachzuweisen. Ein Beweis für die Kommutativität der Addition würde zum Beispiel wie folgt beginnen

```
⊢ ∀n:IN.∀m:IN. n+m = m+n ∈ IN
by all_i THEN all_i
 \
 n:IN, m:IN ⊢ n+m = m+n ∈ IN
 by natE 2
 |\
 | n:IN, m:IN ⊢ n+0 = 0+n ∈ IN
 | by indRedBase
 |  \
 |  n:IN, m:IN ⊢ n = 0+n ∈ IN
 |  by natE 1
 |  |\
 |  | n:IN, m:IN ⊢ 0 = 0+0 ∈ IN
 |  | by symmetry THEN indRedBase
 |  |  \
 |  |  n:IN, m:IN ⊢ 0 = 0 ∈ IN
 |  |  by zeroEq
 |   \
 |    n:IN, m:IN, k:IN, f_k:k=0+k ∈ IN ⊢ s(k) = 0+s(k) ∈ IN
 |    by symmetry THEN indRedUp THEN symmetry
 |     \
 |      n:IN, m:IN, k:IN, f_k:k=0+k ∈ IN ⊢ s(k) = s(0+k) ∈ IN
 |      by sucEq THEN hypothesis 4
  \
  n:IN, m:IN, k:IN, f_k:n+k=k+n ∈ IN ⊢ n+s(k) = s(k)+n ∈ IN
  by  ⋮
```

Wie man sieht, ist schon der Beweis einer einfachen Eigenschaft in der primitiv rekursiven Arithmetik so aufwendig, daß diese Theorie für praktische Zwecke einfach unbrauchbar ist. Wir müssen daher die Erweiterung der Typentheorie um das Konzept der Zahlen auf eine tragfähigere Grundlage stellen.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
int{}()	**natnum**{n:n}() **minus**{}(**natnum**{n:n}())	**ind**{}($\boxed{u}$; $x, f_x.s$; $base$; $y, f_y.t$) **minus**{}($\boxed{u}$), **add**{}($\boxed{u}$;$\boxed{v}$), **sub**{}($\boxed{u}$;$\boxed{v}$) **mul**{}($\boxed{u}$;$\boxed{v}$), **div**{}($\boxed{u}$;$\boxed{v}$), **rem**{}($\boxed{u}$;$\boxed{v}$) **int_eq**($\boxed{u}$; $\boxed{v}$; s; t), **less**($\boxed{u}$; $\boxed{v}$; s; t)
$\mathbb{Z}$	n $-n$	ind($\boxed{u}$; $x, f_x.s$; $base$; $y, f_y.t$) -$\boxed{u}$, $\boxed{u}$+$\boxed{v}$, $\boxed{u}$-$\boxed{v}$ $\boxed{u}$*$\boxed{v}$, $\boxed{u}$÷$\boxed{v}$, $\boxed{u}$ rem $\boxed{v}$ if $\boxed{u}$=$\boxed{v}$ then s else t, if $\boxed{u}$<$\boxed{v}$ then s else t
lt{}(u;v)	**Axiom**{}() Axiom	

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
ind(0; $x, f_x.s$; $base$; $y, f_y.t$)	$\xrightarrow{\beta}$	$base$
ind(n; $x, f_x.s$; $base$; $y, f_y.t$)	$\xrightarrow{\beta}$	$t[n$,ind(n-1; $x, f_x.s$; $base$; $y, f_y.t$) $/y, f_y]$, $(n>0)$
ind(-n; $x, f_x.s$; $base$; $y, f_y.t$)	$\xrightarrow{\beta}$	$s[-n$,ind(-n+1; $x, f_x.s$; $base$; $y, f_y.t$) $/x, f_x]$, $(n>0)$
-i	$\xrightarrow{\beta}$	*Die Negation von i (als Zahl)*
i+j	$\xrightarrow{\beta}$	*Die Summe von i und j*
i-j	$\xrightarrow{\beta}$	*Die Differenz von i und j*
i*j	$\xrightarrow{\beta}$	*Das Produkt von i und j*
i÷j	$\xrightarrow{\beta}$	0, *falls j=0; ansonsten die Integer-Division von i und j*
i rem j	$\xrightarrow{\beta}$	0, *falls j=0; ansonsten der Rest der Division von i und j*
if i=j then s else t	$\xrightarrow{\beta}$	s, *falls $i = j$; ansonsten t*
if i<j then s else t	$\xrightarrow{\beta}$	s, *falls $i < j$; ansonsten t*

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$\mathbb{Z} = \mathbb{Z}$		
i_1<$j_1 = i_2$<j_2	falls	$i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$
Elementsemantik		
$i = i \in \mathbb{Z}$		
Axiom = Axiom $\in$ s<t	falls	es ganze Zahlen i und j gibt, wobei i kleiner als j ist, für die gilt $s \xrightarrow{l} i$ und $t \xrightarrow{l} j$
$\mathbb{Z} = \mathbb{Z} \in \mathsf{U}_j$		
i_1<$j_1 = i_2$<$j_2 \in \mathsf{U}_j$	falls	$i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$

Zusätzliche Einträge in den Semantiktabellen

Abbildung 3.17: Syntax und Semantik des Typs $\mathbb{Z}$

3.4.2.2 Der NuPRL-Typ der ganzen Zahlen

Beim Entwurf der Typentheorie von NuPRL hat man sich dafür entschieden, anstelle der natürlichen Zahlen die *ganzen* Zahlen ($\underline{\mathbb{Z}}$) als grundlegenden Datentyp zu verwenden. Der Grund hierfür ist, daß das formale Schließen über ganze Zahlen nur unwesentlich komplexer ist als Schließen in $\mathbb{N}$ während umgekehrt eine Simulation von $\mathbb{Z}$ – etwa durch $\mathbb{N}+\mathbb{N}$ – das Umgehen mit negativen Zahlen erheblich verkomplizieren würde. Ebenso wurde aus praktischen Gesichtspunkten festgelegt, einen Großteil der elementaren arithmetischen Operationen als vordefinierte Terme in die Theorie mit aufzunehmen.

Die kanonischen Elemente von $\mathbb{Z}$ sind die ganzen Zahlen `0, 1, -1, 2, -2, ...` in der üblichen Dezimaldarstellung.[51] Die oben beschriebene Induktion, der zentrale nichtkanonische Term der ganzen Zahlen, wird erweitert auf die negativen Zahlen, was zu dem Term ind$(u; x, f_x.s; base; y, f_y.t)$ führt. Standardoperationen wie das einstellige `-` (Negation) and die binären Operation `+` (*Addition*), `-` (*Subtraktion*), `*` (*Multiplikation*), ÷ (*Integerdivision*, d.h. Division abgerundet auf die nächstkleinere ganze Zahl), `rem` (*Divisionsrest / remainder*) und das Prädikat `<` (*"kleiner"-Relation*) können in der vertrauten Infix-Notation benutzt werden.

Zur konkreten Analyse von Zahlen werden zwei arithmetische Fallunterscheidungen if u=v then s else t und if u<v then s else t eingeführt. Man beachte jedoch, daß dies nichtkanonische Terme zum Typ $\mathbb{Z}$ sind, die berechenbare Enscheidungsprozeduren darstellen, während das Gleichheits*prädikat* $u = v \in \mathbb{Z}$ und die Relation $u<v$ *Typen* sind, die elementare Propositionen (mit `Axiom` als kanonischem Element) repräsentieren.

Bei der präzisen Beschreibung der Semantik wird davon ausgegangen, daß die Bedeutung arithmetischer Grundoperationen auf ganzen Zahlen nicht weiter erklärt werden muß, da jeder Mensch ein intuitives Verständnis davon besitzt. Deshalb stützt sich die Reduktion von Redizes mit kanonischen Elementen von $\mathbb{Z}$ weniger auf Termersetzung als auf eine Auswertung arithmetischer Berechnungen auf den zugrundeliegenden Parametern des Operators **natnum**. So liefert zum Beispiel eine Reduktion des Terms `4+5`, dessen Abstraktionsform **add**{}(**natnum**{4:n}(); **natnum**{5:n}()) ist, den Term **natnum**{9:n}(), weil die Zahl *neun* genau die Summe der Zahlen *vier* und *fünf* ist. In der Display Form sieht dies genauso aus, wie man es üblicherweise hinschreiben würde: `4+5` $\xrightarrow{l}$ `9`.[52] Die vollständige Syntax und Semantik der formalen Repräsentation ganzer Zahlen und ihrer Operationen innerhalb von NuPRL's Typentheorie ist in Abbildung 3.17 zusammengefaßt.

Die große Anzahl der vordefinierten Operationen hat natürlich auch eine umfangreiche Tabelle von zugehörigen Inferenzregeln zur Folge. Neben erweiterten Versionen der im Abschnitt 3.4.2.1 angesprochenen Regeln – also Typformationsregeln, kanonische Regeln[53] sowie nichtkanonische und Reduktionsregeln für den Induktionsterm – kommen jetzt Einführungsregeln für die strukturelle Analyse der arithmetischen Operationen und Reduktionsregeln für die arithmetischen Fallunterscheidungen hinzu. Zusätzlich müssen Regeln für den Vergleichstyp **lt** eingeführt werden.

Da Reduktion nicht auf Termersetzung sondern auf arithmetischer Berechnung beruht, können die Reduktionsregeln von $\mathbb{Z}$ im Gegensatz zu denen der anderen Typen auch auf nichtkanonische Elemente angewandt

[51] Hier gibt es allerdings eine geringe Inkonsistenz in der einheitlichen Abstraktionsform. Während nichtnegative ganze Zahlen intern durch natnum{n:n}() beschrieben werden, hat die abstrakte Darstellung der negativen Zahlen die Form minus{}(natnum{n:n}()). Dabei wird der Operator minus allerdings als *nichtkanonisch* deklariert, wenn sein Argument *nicht* die Form natnum{n:n}() besitzt. Der Grund für diese Doppelrolle des Operators minus ist, daß einerseits nur Zahlen ohne Vorzeichen als Parametertyp verwendet werden sollten andererseits aber alle ganzen Zahlen als kanonische Elemente von $\mathbb{Z}$ zu gelten haben. Die Einführung eines Zusatzoperators negnum{n:n}() mit Darstellungsform '-n', die aus Gründen einer klareren Trennung vorsuziehen wäre, wurde unterlassen, da der Operator minus ohnehin benötigt wird.

[52] Etwas trickreicher ist der Umgang mit negativen kanonischen Elementen von $\mathbb{Z}$, da ein Term der Gestalt $-n$ intern durch minus{}(natnum{n:n}()) beschrieben wird. Um diese Form aufrecht zu erhalten, muß bei der Auswertung von Operationen, in denen negative Zahlen vorkommen, eine Fallunterscheidung vorgenommen werden. So liefert zum Beispiel die Negation einer positiven ganzen Zahl n intern überhaupt keine Veränderung, da der Term minus{}(natnum{n:n}()) bereits in kanonischer Form ist. Die Negation einer negativen Zahl dagegen muß zwei minus-Operatoren entfernen. Es gilt also

minus{}(natnum{n:n}()) $\xrightarrow{\beta}$ minus{}(natnum{n:n}()) minus{}(minus{}(natnum{n:n}())) $\xrightarrow{\beta}$ natnum{n:n}()

In den anderen Fällen ist Reduktion intern etwas komplizierter zu beschreiben, ist aber ebenfalls durch eine simple Fallunterscheidung zu programmieren. Aus theoretischer Hinsicht ist diese Komplikation allerdings unbedeutend.

[53] Man beachte, daß der Term n in den kanonischen Regeln eine *natürliche* Zahl sein muß, also die interne Form natnum{n:n}() haben muß. Bei negativen kanonischen Termen ist zuvor die Regel `minusEq` bzw. `minusI` anzuwenden.

werden. Dabei entstehen dann als Teilziele Behauptungen, die den Voraussetzungen für die Reduktion in der Redex–Kontrakta Tabelle entsprechen. Zur Erhöhung der Lesbarkeit dieser Regeln machen wir Gebrauch von den folgenden einfachen (nichtinduktiven) definitorischen Abkürzungen.

Definition 3.4.5 (Arithmetische Vergleiche)

$i{=}j \equiv \mathbf{eqi}\{\}(i;j) \equiv i = j \in \mathbb{Z}$
$i{\neq}j \equiv \mathbf{neqi}\{\}(i;j) \equiv \neg(i{=}j)$
$i{\leq}j \equiv \mathbf{le}\{\}(i;j) \equiv \neg(j{<}i)$
$i{\geq}j \equiv \mathbf{ge}\{\}(i;j) \equiv j{\leq}i$
$i{>}j \equiv \mathbf{gt}\{\}(i;j) \equiv j{<}i$

$\Gamma \vdash \mathbb{Z} \in \mathsf{U}_j$ ⌊Ax⌋
by intEq

$\Gamma \vdash n \in \mathbb{Z}$ ⌊Ax⌋
by natnumEq

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** n⌋
by natnumI n

$\Gamma \vdash -s_1 = -s_2$ ⌊Ax⌋
by minusEq
$\Gamma \vdash s_1 = s_2$ ⌊Ax⌋

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** $-s$⌋
by minusI
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** s⌋

$\Gamma \vdash s_1{+}t_1 = s_2{+}t_2$ ⌊Ax⌋
by addEq
$\Gamma \vdash s_1 = s_2$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2$ ⌊Ax⌋

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** $s{+}t$⌋
by addI
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** s⌋
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** t⌋

$\Gamma \vdash s_1{-}t_1 = s_2{-}t_2$ ⌊Ax⌋
by subEq
$\Gamma \vdash s_1 = s_2$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2$ ⌊Ax⌋

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** $s{-}t$⌋
by subI
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** s⌋
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** t⌋

$\Gamma \vdash s_1{*}t_1 = s_2{*}t_2$ ⌊Ax⌋
by mulEq
$\Gamma \vdash s_1 = s_2$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2$ ⌊Ax⌋

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** $s{*}t$⌋
by mulI
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** s⌋
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** t⌋

$\Gamma \vdash s_1 \div t_1 = s_2 \div t_2$ ⌊Ax⌋
by divEq
$\Gamma \vdash s_1 = s_2$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2$ ⌊Ax⌋
$\Gamma \vdash t_1 \neq 0$ ⌊Ax⌋

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** $s \div t$⌋
by divI
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** s⌋
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** t⌋

$\Gamma \vdash s_1 \operatorname{rem} t_1 = s_2 \operatorname{rem} t_2$ ⌊Ax⌋
by remEq
$\Gamma \vdash s_1 = s_2$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2$ ⌊Ax⌋
$\Gamma \vdash t_1 \neq 0$ ⌊Ax⌋

$\Gamma \vdash \mathbb{Z}$ ⌊**ext** $s \operatorname{rem} t$⌋
by remI
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** s⌋
$\Gamma \vdash \mathbb{Z}$ ⌊**ext** t⌋

$\Gamma \vdash$ if $u_1{=}v_1$ then s_1 else t_1
= if $u_2{=}v_2$ then s_2 else $t_2 \in T$ ⌊Ax⌋
by int_eqEq
$\Gamma \vdash u_1{=}u_2$ ⌊Ax⌋
$\Gamma \vdash v_1{=}v_2$ ⌊Ax⌋
$\Gamma, v{:}\, u_1{=}v_1 \vdash s_1 = s_2 \in T$ ⌊Ax⌋
$\Gamma, v{:}\, u_1{\neq}v_1 \vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash$ if $u_1{<}v_1$ then s else t
= if $u_2{<}v_2$ then s_2 else $t_2 \in T$ ⌊Ax⌋
by lessEq
$\Gamma \vdash u_1{=}u_2$ ⌊Ax⌋
$\Gamma \vdash v_1{=}v_2$ ⌊Ax⌋
$\Gamma, v{:}\, u_1{<}v_1 \vdash s_1 = s_2 \in T$ ⌊Ax⌋
$\Gamma, v{:}\, u_1{\geq}v_1 \vdash t_1 = t_2 \in T$ ⌊Ax⌋

Abbildung 3.18: Inferenzregeln des Typs $\mathbb{Z}$ (1)

$\Gamma \vdash \mathsf{ind}(u_1; x_1, f_{x1}.s_1; base_1; y_1, f_{y_1}.t_1)$
$= \mathsf{ind}(u_2; x_2, f_{x2}.s_2; base_2; y_2, f_{y_2}.t_2)$
$\in T[u_1/z]$ [Ax]
by indEq z T
$\Gamma \vdash u_1 = u_2$ [Ax]
$\Gamma,\ x{:}\mathbb{Z},\ v{:}\ x<0,\ f_x{:}T[(x+1)/z]$
$\vdash s_1[x, f_x / x_1, f_{x1}] = s_2[x, f_x / x_2, f_{x2}]$
$\in T[x/z]$ [Ax]
$\Gamma \vdash base_1 = base_2 \in T[0/z]$ [Ax]
$\Gamma,\ x{:}\mathbb{Z},\ v{:}\ 0<x,\ f_x{:}T[(x-1)/z]$
$\vdash t_1[x, f_x / y_1, f_{y_1}] = t_2[x, f_x / y_2, f_{y_2}]$
$\in T[x/z]$ [Ax]

$\Gamma,\ z{:}\mathbb{Z},\ \Delta \vdash C$
[**ext** $\mathsf{ind}(z; x, f_x.s[\mathsf{Axiom}/v]; base; x, f_x.t[\mathsf{Axiom}/v])$]
by intE i
$\Gamma,\ z{:}\mathbb{Z},\ \Delta,\ x{:}\mathbb{Z},\ v{:}\ x<0,\ f_x{:}C[(x+1)/z]$
$\vdash C[x/z]$ [**ext** s]
$\Gamma,\ z{:}\mathbb{Z},\ \Delta \vdash C[0/z]$ [**ext** $base$]
$\Gamma,\ z{:}\mathbb{Z},\ \Delta,\ x{:}\mathbb{Z},\ v{:}\ 0<x,\ f_x{:}C[(x-1)/z]$
$\vdash C[x/z]$ [**ext** t]

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; base; y, f_y.t) = t_2 \in T$ [Ax]
by indRedDown
$\Gamma \vdash t[i, \mathsf{ind}(i+1; x, f_x.s; base; y, f_y.t)\ /x, f_x]$
$= t_2 \in T$ [Ax]
$\Gamma \vdash i<0$ [Ax]

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; base; y, f_y.t) = t_2 \in T$ [Ax]
by indRedUp
$\Gamma \vdash t[i, \mathsf{ind}(i-1; x, f_x.s; base; y, f_y.t)\ /y, f_y]$
$= t_2 \in T$ [Ax]
$\Gamma \vdash 0<i$ [Ax]

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; base; y, f_y.t) = t_2 \in T$ [Ax]
by indRedBase
$\Gamma \vdash base = t_2 \in T$ [Ax]
$\Gamma \vdash i=0$ [Ax]

$\Gamma \vdash \mathsf{if}\ u=v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by int_eqRedT
$\Gamma \vdash s = t_2 \in T$ [Ax]
$\Gamma \vdash u=v$ [Ax]

$\Gamma \vdash \mathsf{if}\ u=v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by int_eqRedF
$\Gamma \vdash t = t_2 \in T$ [Ax]
$\Gamma \vdash u \neq v$ [Ax]

$\Gamma \vdash \mathsf{if}\ u<v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by lessRedT
$\Gamma \vdash s = t_2 \in T$ [Ax]
$\Gamma \vdash u<v$ [Ax]

$\Gamma \vdash \mathsf{if}\ u<v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by lessRedF
$\Gamma \vdash t = t_2 \in T$ [Ax]
$\Gamma \vdash u \geq v$ [Ax]

$\Gamma \vdash 0 \leq s \operatorname{rem} t \ \wedge\ s \operatorname{rem} t < t$ [Ax]
by remBounds1
$\Gamma \vdash 0 \leq s$ [Ax]
$\Gamma \vdash 0 < t$ [Ax]

$\Gamma \vdash 0 \leq s \operatorname{rem} t \ \wedge\ s \operatorname{rem} t < -t$ [Ax]
by remBounds2
$\Gamma \vdash 0 \leq s$ [Ax]
$\Gamma \vdash t < 0$ [Ax]

$\Gamma \vdash s \operatorname{rem} t \leq 0 \ \wedge\ s \operatorname{rem} t > -t$ [Ax]
by remBounds3
$\Gamma \vdash s \leq 0$ [Ax]
$\Gamma \vdash 0 < t$ [Ax]

$\Gamma \vdash s \operatorname{rem} t \leq 0 \ \wedge\ s \operatorname{rem} t > t$ [Ax]
by remBounds4
$\Gamma \vdash s \leq 0$ [Ax]
$\Gamma \vdash t < 0$ [Ax]

$\Gamma \vdash s = (s \div t) * t + (s \operatorname{rem} t)$ [Ax]
by divremSum
$\Gamma \vdash s \in \mathbb{Z}$ [Ax]
$\Gamma \vdash t \neq 0$ [Ax]

$\Gamma \vdash s_1<t_1 = s_2<t_2 \in \mathsf{U}_j$ [Ax]
by ltEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]

$\Gamma \vdash \mathsf{Axiom} \in s<t$ [Ax]
by axiomEq_lt
$\Gamma \vdash s<t$ [Ax]

Abbildung 3.19: Inferenzregeln des Typs $\mathbb{Z}$ (2)

Neben den in den Abbildungen 3.18 und 3.19 zusammengestellten Regeln für den Typ der ganzen Zahlen gibt es eine weitere Regel, die in praktischer Hinsicht extrem wichtig ist, aber nur algorithmisch erklärt werden kann. Die Regel `arith` j ist eine arithmetische *Entscheidungsprozedur*, die es erlaubt, in einem einzigen Schritt die Gültigkeit eines arithmetischen Vergleichs zu beweisen, der aus den Hypothesen durch eine eingeschränkte Form arithmetischen Schließens folgt. Die Konklusion muß dabei entweder die Form 'void' haben (falls die Hypothesen arithmetische Widersprüche enthalten) oder disjunktiv aus $=, \neq, <, >, \leq, \geq$ und jeweils zwei Termen des Typs $\mathbb{Z}$ zusammengesetzt sein. Den Algorithmus, der hinter dieser Regel steht, seinen Anwendungsbereich und seine Rechtfertigung als Bestandteil eines formalen Schlußfolgerungssystems werden wir im Kapitel 4.3 besprechen.

3.4.3 Listen

So wie die natürlichen Zahlen gehören Listen zu den grundlegensten Konstrukten der 'realen' Programmierwelt. Sie werden benötigt, wenn endliche, aber beliebig große Strukturen von Datensätzen – wie lineare Listen, Felder, Bäume etc. – innerhalb von Programmen aufgebaut werden müssen. Rein hypothetisch wäre es möglich, den Datentyp der Listen mit Elementen eines Datentyps T in der bisherigen Typentheorie zu simulieren.[54] Eine solche Simulation wäre allerdings verhältnismäßig aufwendig und würde die wesentlichen Eigenschaften von Listen nicht zur Geltung kommen lassen. Aus diesem Grunde wurde der Datentyp T `list` der Listen (Folgen) über einem beliebigen Datentyp T und seine Operatoren explizit zur Typentheorie hinzugenommen und direkt auf einer Formalisierung der gängigen Semantik von Listen abgestützt.

Wie in Abschnitt 2.3.3 auf Seite 56 bereits angedeutet wurde, können Listen als eine Art Erweiterung des Konzepts der natürlichen Zahlen – aufgebaut durch Null und Nachfolgerfunktion – angesehen werden. Die kanonischen Elemente werden gebildet durch die leere Liste `[]` und durch Anwendung der Operation **cons**$\{\}(t;l)$, welche ein Element $t \in T$ vor eine bestehende Liste l anhängt. Der nichtkanonische Term ist ein Gegenstück zum Induktionsterm ind: `list_ind`$(L;\ base;\ x,l,f_l.t)$ beschreibt die Werte einer Funktion f, die ihr Ergebnis durch eine Analyse des induktiven Aufbaus der Liste L bestimmt. Ist L die leere Liste, so lautet das Ergebnis *base*. Hat L dagegen die Gestalt **cons**$\{\}(x;l)$, so ist $t[x,l,f_l]$ das Ergebnis, wobei f_l ein Platzhalter für $f(l)$ ist. Listeninduktion ist also eine Erweiterung der primitiven Rekursion.

Im Gegensatz zu den ganzen Zahlen beschränkt man sich auf diese vier Grundoperationen und formuliert alle anderen Listenoperationen als definitorische Erweiterungen. Die Semantik und die Inferenzregeln von Listen sind leicht zu formalisieren, wenn man die Formalisierung von natürlichen Zahlen aus Abschnitt 3.4.2.1 als Ausgangspunkt nimmt. In Abbildung 3.20 ist die komplette Erweiterung des Typsystems zusammengefaßt.

Wir wollen nun anhand eines größeren Beispiels zeigen, daß das Prinzip, Programme durch Beweisführung zu entwickeln (Entwurfsprinzip 3.4.3), in der Tat ein sinnvolles Konzept ist. Die Extraktion von Programmen aus dem Beweis eines Spezifikationstheorems garantiert nicht nur die Korrektheit des erstellten Programms sondern führt in den meisten Fällen auch zu wesentlich besser durchdachten und effizienteren Algorithmen. Wir hatten bereits im Einführungskapitel im Abschnitt 1.1 mit dem Beispiel der maximalen Segmentsumme illustriert, daß mathematisches Problemlösen und Programmierung sehr ähnliche Aufgaben sind. Wir wollen nun zeigen, wie dieses Beispiel innerhalb eines formalen Schlußfolgerungssystems behandelt werden kann.

Beispiel 3.4.6 (Maximale Segmentsumme)

In Beispiel 1.1.1 auf Seite 2 hatten wir das Problem der Berechnung der maximalen Summe von Teilsegmenten einer gegebenen Folge ganzer Zahlen aufgestellt.

> *Zu einer gegebenen Folge $a_1, a_2, \ldots, a_n$ von n ganzen Zahlen soll die Summe $m = \sum_{i=p}^{q} a_i$ einer zusammenhängenden Teilfolge bestimmt werden, die maximal ist im Bezug auf alle möglichen Summen zusammenhängender Teilfolgen $a_j, a_{j+1} \ldots, a_k$.*

[54] Man kann eine Liste mit Elementen aus T als eine endliche Funktion $l \in \{1..n\} \to T$ interpretieren. Wenn wir die endliche Menge $\{1..n\}$ wiederum durch $x{:}\mathbb{Z} \times (1 \leq x \wedge x \leq n)$ repräsentieren, so können wir den Datentyp T `list` wie folgt simulieren

$$T\ \mathsf{list} \equiv n{:}\mathbb{N} \times (x{:}\mathbb{Z} \times (1 \leq x \wedge x \leq n)) \to T$$

Eine andere Möglichkeit ergibt sich aus der Darstellung von Listen im λ-Kalkül wie in Abschnitt 2.3.3.

kanonisch (Typen)	**kanonisch** (Elemente)	**nichtkanonisch**
$\mathbf{list}\{\}(T)$	$\mathbf{nil}\{\}()$, $\mathbf{cons}\{\}(t;l)$	$\mathbf{list\_ind}\{\}(\boxed{s};\ base;\ x,l,f_{xl}.t)$
$T\ \mathsf{list}$	$[]$, $t.l$	$\mathsf{list\_ind}(\boxed{s};\ base;\ x,l,f_{xl}.t)$

Zusätzliche Einträge in die Operatorentabelle

Redex		**Kontraktum**
$\mathsf{list\_ind}([];\ base;\ x,l,f_{xl}.t)$	$\xrightarrow{\beta}$	$base$
$\mathsf{list\_ind}(s.u;\ base;\ x,l,f_{xl}.t)$	$\xrightarrow{\beta}$	$t[s,u,\mathsf{list\_ind}(u;\ base;\ x,l,f_{xl}.t)\ /x,l,f_{xl}]$

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$T_1\ \mathsf{list} = T_2\ \mathsf{list}$	falls	$T_1 = T_2$
Elementsemantik		
$[] = [] \in T\ \mathsf{list}$	falls	T Typ
$t_1.l_1 = t_2.l_2 \in T\ \mathsf{list}$	falls	T Typ und $t_1 = t_2 \in T$ und $l_1 = l_2 \in T\ \mathsf{list}$
$T_1\ \mathsf{list} = T_2\ \mathsf{list} \in \mathsf{U}_j$	falls	$T_1 = T_2 \in \mathsf{U}_j$

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash T_1\ \mathsf{list} = T_2\ \mathsf{list} \in \mathsf{U}_j$ [Ax]
by listEq
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash [] \in T\ \mathsf{list}$ [Ax]
by nilEq j
$\Gamma \vdash T \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash t_1.l_1 = t_2.l_2 \in T\ \mathsf{list}$ [Ax]
by consEq
$\Gamma \vdash t_1 = t_2 \in T$ [Ax]
$\Gamma \vdash l_1 = l_2 \in T\ \mathsf{list}$ [Ax]

$\Gamma \vdash \mathsf{list\_ind}(s_1;\ base_1;\ x_1,l_1,f_{xl1}.t_1) = \mathsf{list\_ind}(s_2;\ base_2;\ x_2,l_2,f_{xl2}.t_2) \in T[s_1/z]$ [Ax]
by list_indEq z T $S\ \mathsf{list}$
$\Gamma \vdash s_1 = s_2 \in S\ \mathsf{list}$ [Ax]
$\Gamma \vdash base_1 = base_2 \in T[[]/z]$ [Ax]
$\Gamma,\ x{:}S,\ l{:}S\ \mathsf{list},\ f_{xl}{:}T[l/z] \vdash t_1[x,l,f_{xl}\,/\,x_1,l_1,f_{xl1}] = t_1[x,l,f_{xl}\,/\,x_2,l_2,f_{xl2}] \in T[x.l/z]$ [Ax]

$\Gamma \vdash \mathsf{list\_ind}([];\ base;\ x,l,f_{xl}.t) = t_2 \in T$ [Ax]
by list_indRedBase
$\Gamma \vdash base = t_2 \in T$ [Ax]

$\Gamma \vdash \mathsf{list\_ind}(s.u;\ base;\ x,l,f_{xl}.t) = t_2 \in T$ [Ax]
by list_indRedUp
$\Gamma \vdash t[s,u,\mathsf{list\_ind}(u;\ base;\ x,l,f_{xl}.t)\ /x,l,f_{xl}] = t_2 \in T$ [Ax]

$\Gamma \vdash T\ \mathsf{list}$ [**ext** []]
by nilI j
$\Gamma \vdash T \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash T\ \mathsf{list}$ [**ext** $t.l$]
by consI
$\Gamma \vdash T$ [**ext** t]
$\Gamma \vdash T\ \mathsf{list}$ [**ext** l]

$\Gamma,\ z{:}T\ \mathsf{list},\ \Delta \vdash C$ [**ext** $\mathsf{list\_ind}(z;\ base;\ x,l,f_{xl}.t)$]
by listE i
$\Gamma,\ z{:}T\ \mathsf{list},\ \Delta \vdash C[[]/z]$ [**ext** $base$]
$\Gamma,\ z{:}T\ \mathsf{list},\ \Delta,\ x{:}T,\ l{:}T\ \mathsf{list},\ f_{xl}{:}C[l/z] \vdash C[x.l/z]$ [**ext** t]

Inferenzregeln

Abbildung 3.20: Syntax, Semantik und Inferenzregeln des Listentyps

```
⊢ ∀a:ℤ list.∀a1:ℤ.∃L:ℤ.∃M:ℤ. L=maxbeg(a1.a) ∧ M=maxseq(a1.a)
by all_i THEN listE 1 THEN all_i                                   (Induktion auf a)
|\
| a:ℤ list, a1:ℤ ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.[]) ∧ M=maxseq(a1.[])
| by ex_i a1 THEN ex_i a1 THEN and_i
| |\
| | a:ℤ list, a1:ℤ ⊢ a1=maxbeg(a1.[])
| | by ...  Anwendung arithmetischer Lemmata ...
| \
|   a:ℤ list, a1:ℤ ⊢ a1=maxseq(a1.[])
|   by ...  Anwendung arithmetischer Lemmata ...
 \
  a:ℤ list, x:ℤ, l:ℤ list, v: ∀a1:ℤ.∃L:ℤ.∃M:ℤ. L=maxbeg(a1.l) ∧ M=maxseq(a1.l), a1:ℤ
  ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.(x.l)) ∧ M=maxseq(a1.(x.l))
  by all_e 4 x THEN thin 4
   \
    a:ℤ list, x:ℤ, l:ℤ list, a1:ℤ, v1: ∃L:ℤ.∃M:ℤ. L=maxbeg(x.l) ∧ M=maxseq(x.l)
    ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.(x.l)) ∧ M=maxseq(a1.(x.l))
    by ex_e 5 THEN ex_e 6 THEN and_e 7
     \
      a:ℤ list, x:ℤ, l:ℤ list, a1:ℤ, L:ℤ, M:ℤ, v2: L=maxbeg(x.l), v3: M=maxseq(x.l)
      ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.(x.l)) ∧ M=maxseq(a1.(x.l))
      by ex_i max(L+a1,a1) THEN ex_i max(M, max(L+a1,a1)) THEN and_i
      |\
      | a:ℤ list, x:ℤ, l:ℤ list, a1:ℤ, L:ℤ, M:ℤ, v2: L=maxbeg(x.l), v3: M=maxseq(x.l)
      | ⊢ max(L+a1,a1)=maxbeg(a1.(x.l))
      | by ...  Anwendung arithmetischer Lemmata ...
       \
        a:ℤ list, x:ℤ, l:ℤ list, a1:ℤ, L:ℤ, M:ℤ, v2: L=maxbeg(x.l), v3: M=maxseq(x.l)
        ⊢ max(M, max(L+a1,a1))=maxseq(a1.(x.l))
        by ...  Anwendung arithmetischer Lemmata ...
```

Abbildung 3.21: Formaler Induktionsbeweis für die Existenz der maximalen Segmentsumme

Unsere Analyse ergab, daß die naheliegende iterative Lösung (kubische Laufzeit!) durch mathematisch-induktive Betrachtungen zu einem linearen Algorithmus verbessert werden kann. Bestimmt man nämlich parallel zu der maximalen Segmentsumme M_k einer Teilliste von k Elementen auch noch die maximalen Summe L_k von Segmenten, die das letzte Element enthalten, dann ist $M_1{=}L_1{=}a_1$. L_{k+1} ist das Maximum von a_{k+1} und $a_{k+1}{+}L_k$. M_{k+1} ist schließlich das Maximum von L_{k+1} und M_k.

Da ganze Zahlen und Listen vordefiniert sind, ist es nicht schwierig, die obige Problemstellung innerhalb der Typentheorie formal zu beschreiben. Allerdings ist zu berücksichtigen, daß Listen schrittweise nach *vorne* erweitert werden, während in unserer Argumentation die Erweiterung zum rechten Ende hin betrachtet wurde. Es empfiehlt sich daher, zugunsten einer einfacheren Beweisführung das Argument umzudrehen und von der eingebauten Listeninduktion Gebrauch zu machen.

Zudem geht das Argument davon aus, daß die betrachteten Listen immer mindestens ein Element enthalten. Wir müssen unsere Induktion also bei der Länge 1 verankern, denn ansonsten müßten wir $L_0{=}0$ und $M_0{=}\text{-}\infty$ wählen. In Abbildung 3.21 haben wir eine Formalisierung des zentralen Arguments innerhalb eines NuPRL-Beweises skizziert.[55] Wir verwenden dabei die folgenden definitorischen Abkürzungen.

[55]Ohne eine Unterstützung durch Lemmata und Beweistaktiken, welche die Anwendung von Regeln zum Teil automatisieren (siehe Abschnitt 4.2) kann die arithmetische *Rechtfertigung* der in Beispiel 1.1.1 gegebenen Argumentation nicht praktisch formalisiert werden. Der vollständige Beweis macht daher ausgiebig Gebrauch von beidem und soll hier nicht weiter betrachtet werden. Die mathematischen Gleichungen (Lemmata), welche zur Unterstützung verwandt werden, sind die folgenden.

$M_1 = \mathtt{max}(\{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}1\}) = \sum_{i=1}^{1} a_i = a_1$

$M_{n+1} = \mathtt{max}(\{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}n{+}1\}) = \mathtt{max}(\{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}n\} \cup \{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{=}n\})$
$= \mathtt{max}(\mathtt{max}(\{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}n\}), \mathtt{max}(\{\sum_{i=p}^{n} a_i \mid 1{\le}p{\le}n\})) = \mathtt{max}(M_n, L_{n+1})$

$L_1 = \mathtt{max}(\{\sum_{i=p}^{1} a_i \mid 1{\le}p{\le}1\}) = \sum_{i=1}^{1} a_i = a_1$

$L_{n+1} = \mathtt{max}(\{\sum_{i=p}^{n+1} a_i \mid 1{\le}p{\le}n{+}1\}) = \mathtt{max}(\{\sum_{i=p}^{n+1} a_i \mid 1{\le}p{\le}n\} \cup \{\sum_{i=p}^{n+1} a_i \mid 1{\le}p{=}n{+}1\})$
$= \mathtt{max}(\{\sum_{i=p}^{n} a_i{+}a_{n+1} \mid 1{\le}p{\le}n\} \cup \{\sum_{i=n+1}^{n+1} a_i\}) = \mathtt{max}(\mathtt{max}(\{\sum_{i=p}^{n} a_i{+}a_{n+1} \mid 1{\le}p{\le}n\}), \mathtt{max}(\{a_{n+1}\}))$

$$
\begin{array}{lll}
\mathsf{max}(i,j) & \equiv \mathbf{max}\{\}(i;j) & \equiv \mathsf{if}\ i{<}j\ \mathsf{then\ j\ else}\ i \\
|a| & \equiv \mathbf{length}\{\}(a) & \equiv \mathsf{list\_ind}(a;\,0;\,\mathtt{x,a'},\mathsf{lg}_{a'}.\mathsf{lg}_{a'}{+}1) \\
\mathsf{hd}(a) & \equiv \mathbf{head}\{\}(a) & \equiv \mathsf{list\_ind}(a;\,0;\,\mathtt{x,a'},\mathsf{hd}_{a'}.\mathtt{x}) \\
\mathsf{tl}(a) & \equiv \mathbf{tail}\{\}(a) & \equiv \mathsf{list\_ind}(a;\,[];\,\mathtt{x,a'},\mathsf{tl}_{a'}.\mathtt{a'}) \\
a_i & \equiv \mathbf{select}\{\}(a;i) & \equiv \mathsf{hd}(\mathsf{ind}(i;\,\_,\_.a;\,a;\,\mathtt{x,a'}.\mathsf{tl}(\mathtt{a'})\mathsf{tl}(\mathtt{s}))) \\
\sum_{i=p}^{q} a_i & \equiv \mathbf{sum}\{\}(a;p;q) & \equiv \mathsf{ind}(q{-}p;\,\_,\_,0;\ a_p;\ \mathtt{i,sum.\,sum}{+}a_{p+i+1}) \\
M = \mathtt{maxseq}(a) & \equiv \mathbf{maxseq}\{\}(a;M) & \equiv \exists \mathtt{k,j}{:}\mathbb{Z}.\ (1{\le}\mathtt{k}\wedge\mathtt{k}{\le}\mathtt{j}\wedge\mathtt{j}{\le}|a|)\ \wedge\ M{=}\textstyle\sum_{i=k}^{j} a_i \\
 & & \quad \wedge\ \forall \mathtt{p,q}{:}\mathbb{Z}.\ (1{\le}\mathtt{p}\wedge\mathtt{p}{\le}\mathtt{q}\wedge\mathtt{q}{\le}|a|)\ \Rightarrow\ M{\ge}\textstyle\sum_{i=p}^{q} a_i \\
L = \mathtt{maxbeg}(a) & \equiv \mathbf{maxbeg}\{\}(a;L) & \equiv \exists \mathtt{j}{:}\mathbb{Z}.\ (1{\le}\mathtt{j}\wedge\mathtt{j}{\le}|a|)\ \wedge\ L{=}\textstyle\sum_{i=1}^{j} a_i \\
 & & \quad \wedge\ \forall \mathtt{q}{:}\mathbb{Z}.\ (1{\le}\mathtt{q}\wedge\mathtt{q}{\le}|a|)\ \Rightarrow\ L{\ge}\textstyle\sum_{i=1}^{q} a_i
\end{array}
$$

Der Algorithmus, der in diesem Beweis enthalten ist, hat die Form

$$
\begin{array}{l}
\lambda \mathtt{a}.\lambda \mathtt{a}_1.\ \ \mathsf{list\_ind}(\mathtt{a};\ \langle \mathtt{a}_1, \mathtt{a}_1, pf_{base}\rangle; \\
\qquad\qquad \mathtt{x,l,v}.\ \ \lambda \mathtt{a}_1.\mathsf{let}\ \langle \mathtt{L},\langle \mathtt{M},\mathtt{v}_2,\mathtt{v}_3\rangle\rangle = \mathtt{v\ x}\ \mathsf{in}\ \langle \mathsf{max}(\mathtt{L{+}a}_1,\mathtt{a}_1), \mathsf{max}(\mathtt{M},\ \mathsf{max}(\mathtt{L{+}a}_1,\mathtt{a}_1)), pf_{ind}\rangle)
\end{array}
$$

wobei pf_{base} und pf_{ind} Terme sind, welche die Korrektheit der vorgegebenen Lösungen nachweisen. Durch den formalen Beweis ist der Algorithmus, der nur eine einzige Induktionsschleife beinhaltet, als korrekt nachgewiesen und somit erheblich effizienter als eine ad hoc Lösung des Problems.

3.4.4 Teilmengen

Das Beispiel der formalen Herleitung eines Algorithmus zur Bestimmung der maximalen Segmentsumme in einer Liste zeigt nicht nur die Vorteile einer mathematischen Vorgehensweise sondern auch einige Schwächen in der praktischen Ausdruckskraft der bisherigen Theorie.

- Zum einen enthält ein Programm, welches aus dem Beweis eines Spezifikationstheorems extrahiert wird, Teilterme, die nur zum Nachweis der Korrektheit der Resultate, nicht aber zu ihrer Berechnung benötigt werden. Der Grund hierfür liegt in der Tatsache, daß die die Standard-Einbettung der konstruktiven Logik den Existenzquantor $\exists x{:}T.\,A[x]$ mit dem Produktraum $x{:}T{\times}A[x]$ identifiziert und somit ein Beweisterm nicht nur das Element x selbst sondern auch einen Nachweis für $A[x]$ konstruiert. Während aus beweistheoretischer Sicht diese Komponente unbedingt nötig ist, bedeutet sie für den Berechnungsprozeß nur eine unnötige Belastung. Natürlich könnten wir versuchen, sie im nachhinein mithilfe des spread-Konstrukts wieder auszublenden. Bei komplexeren Induktionsbeweisen wie dem in Beispiel 3.4.6 wird dies jedoch kaum (automatisch) durchführbar sein und zudem wäre es ohnehin effizienter, diese Anteile erst gar nicht in den Algorithmus einzufügen. Dies verlangt jedoch eine andersartige Formulierung des Spezifikationstheorems, bei der Existenzquantoren mit Typkonstrukten identifiziert werden, die *nur* das gewünschte Element als Evidenz enthalten und den ungewünschten Beweisterm unterdrücken.

- Ein zweites Problem war die etwas umständliche Formulierung der Tatsache, daß wir nichtleere Listen – also eine Teilmenge des Listentyps – als Eingabebereich der Spezifikation betrachten wollten. Natürlich hätten wir auch formulieren können

  $$\vdash\ \forall \mathtt{a}{:}\mathbb{Z}\,\mathsf{list}.\neg(\mathtt{a{=}[]} \in \mathbb{Z}\,\mathsf{list})\ \Rightarrow\ \exists \mathtt{L}{:}\mathbb{Z}.\exists \mathtt{M}{:}\mathbb{Z}.\ \mathtt{L = maxbeg(a)}\ \wedge\ \mathtt{M = maxseq(a)}.$$

  In diesem Fall wären wir jedoch gezwungen, als Eingabe für den extrahierten Algorithmus nicht nur eine Liste a, sondern auch einen Beweis dafür, daß a nicht leer ist, einzugeben. Was wir in Wirklichkeit formulieren wollen, ist daß der Eingabebereich die Menge $\{\,\mathtt{a}{:}\mathbb{Z}\,\mathsf{list} \mid \neg(\mathtt{a{=}[]} \in \mathbb{Z}\,\mathsf{list})\,\}$ ist. Mit den bisherigen Mitteln aber können wir das nicht ausdrücken, da es uns die Konstrukte fehlen, einen gegebenen Typ auf eine Teilmenge *einzuschränken*.

Wir benötigen also aus mehreren Gründen einen *Teilmengentyp* der Form $\{x{:}S \mid P[x]\,\}$, der aus allen Elementen von S besteht, von denen wir wissen, daß sie die Eigenschaft P besitzen – ohne daß wir hierzu

$= \mathtt{max}(\mathtt{max}(\{\sum_{i=p}^{n} a_i \mid 1{\le}\mathtt{p}{\le}\mathtt{n}\}){+}a_{n+1},\ a_{n+1}) \quad = \quad \mathtt{max}(L_n{+}a_{n+1}, a_{n+1})$

einen separaten Beweis als Evidenz hinzugeben müssen. Ohne ein solches Konstrukt ist eine Formalisierung vieler Standardkonstrukte der Mathematik nicht praktisch durchführbar. Ohne einen Teilmengentyp müßten wir ein so einfaches Problem wie das in Beispiel 3.4.1 vorgestellte Spezifikationstheorem für die Berechnung der Integerquadratwurzel sehr umständlich formulieren, nämlich als

$$\vdash \forall \mathtt{x}{:}\mathbb{Z}.\, 0{\le}\mathtt{x} \;\Rightarrow\; \exists \mathtt{y}{:}\mathbb{Z}.\; 0{\le}\mathtt{y} \;\wedge\; \mathtt{y}^2{\le}\mathtt{x} \;\wedge\; \mathtt{x}{<}(\mathtt{y}{+}1)^2$$

und die eigentlich interessierende Fragestellung würde von den Randbedingungen völlig verschleiert. Miteinander verwandte Typen wie die ganzen und die natürlichen Zahlen können ohne ein Teilmengenkonstrukt nicht wirklich miteinander in Beziehung gesetzt werden und die Tatsache, daß jede natürliche Zahl auch als ganze Zahl zu verstehen ist, würde durch die explizit vorzunehmenden Konversionen völlig verschleiert.

Eine Simulation des Konzepts der Teilmengen durch bisherige Typkonstrukte ist nicht sinnvoll möglich. Als einzig denkbare Abstützung käme der Produkttyp in Frage – also eine Definition der Art $\{x{:}S \mid P\} \equiv x{:}S{\times}P$. Dies würde aber die Intuition, die hinter dem Konzept der Teilmenge steht, nur unvollständig wiederspiegeln. Zwar ist oberflächlich eine Ähnlichkeit vorhanden, weil die Werte, die man für x in $x{:}S{\times}P$ einsetzen darf, tatsächlich genau die Elemente von $\{x{:}S \mid P\}$ sind. Jedoch muß im Produkttyp immer noch eine zweite Komponente hinzukommen, nämlich ein Term, welcher $P[x]$ *beweist*. Diese Komponente aber wollen wir im Teilmengenkonstrukt gerade nicht haben. Es reicht uns, zu *wissen*, daß jedes Element x von $\{x{:}S \mid P\}$ die Eigenschaft $P[x]$ besitzt, aber umgehen wollen wir ausschließlich mit dem Element x selbst. Insbesondere in Algorithmen, die auf Teilmengen operieren, wollen wir nicht gezwungen sein, den Beweisterm als Eingabe mitzuliefern, der vom Algorithmus dann ohnehin weggeworfen wird. Dies wäre eine sehr unnatürliche Form von Algorithmen, die in der Welt der Programmierung niemals Fuß fassen könnte. Anstatt einer Simulation müssen wir daher nach einem eleganteren Weg suchen, Teilmengen zu beschreiben.

Diese Überlegungen führten schließlich dazu, die Typentheorie um einen Teilmengenkonstruktor $\{x{:}S \mid P\}$ zu ergänzen, welcher das Konzept der Teilmengenbildung direkt wiederspiegelt. Dies ist weniger eine Frage nach der syntaktischen Repräsentation als nach der semantischen Elementbeziehung, die präzise fixiert werden muß, und nach einer geeigneten Darstellung von *implizit vorhandenem Wissen* innerhalb der Inferenzregeln. Das Wissen, daß ein Element s von $\{x{:}S \mid P\}$ die Eigenschaft $P[s]$ besitzt, für die aber keine Evidenz – also kein Beweisterm – explizit mitgeliefert wird, muß so verwaltet werden, daß es in Beweisen verwendet werden kann, innerhalb der extrahierten Algorithmen aber keine Rolle spielt.[56]

Die formale Beschreibung des Teilmengentyps – insbesondere der Inferenzregeln – ist angelehnt an den Produkttyp $x{:}S{\times}T$, weicht aber an den Stellen davon ab, wo es um die explizite Benennung der zweiten Komponente t eines Paars $\langle s,t\rangle \in x{:}S{\times}T$ geht. Für den Teilmengentyp $\underline{\{x{:}S \mid T\}}$ reicht es zu wissen, daß es ein solches $t{\in}T[s/x]$ gibt, denn entsprechend dem intuitiven Verständnis sind seine Elemente genau diejenigen Elemente s von S, für die der Typ $T[s/x]$ nicht leer – also beweisbar – ist. Eigene kanonische oder nichtkanonische Terme gibt es für den Teilmengentyp nicht und auch die Gleichheitsrelation auf den Elementen von S wird unverändert übernommen.

Die Verwaltung des Wissens, daß für ein Element s von $\{x{:}S \mid T\}$ der Typ $T[s/x]$ nicht leer ist, verlangt eine leichte Modifikation von Sequenzen und Beweisen. Einerseits müssen die Regeln dafür sorgen, daß zum Nachweis von $s{\in}\{x{:}S \mid T\}$ die Eigenschaft $T[s/x]$ überprüft wird. Andererseits aber ist dieses Wissen nicht im Term s selbst enthalten. Somit darf bei der Elimination von Mengenvariablen eine Evidenz für die Eigenschaft $T[s/x]$ nicht in den erzeugten Algorithmus eingehen.[57] Wir wollen dies an einem einfachen Beispiel erläutern.

[56] Es sei angemerkt, daß dies keine Aufgabe ist, für die es eine eindeutig optimale Antwort gibt, da die Mathematik mit dem Konzept der Teilmenge relativ nachlässig umgeht und es in einem hochgradig unkonstruktiv Sinne verwendet. In einer konstruktiven Denkwelt, die im Zusammenhang mit der Automatisierung des Schließens nun einmal nötig ist, kann diese Vorgehensweise, die bedenkenlos einem Objekt Eigenschaften zuweist, die aus diesem nur unter Hinzunahme zusätzlicher Informationen herleitbar ist, nur unvollständig nachgebildet werden. In verschiedenen Theorien sind hierfür unterschiedliche Lösungen vorgeschlagen worden, die jeweils ihre Stärken und haben. In [Salvesen & Smith, 1988] werden diese Unterschiede ausführlicher beleuchtet.

[57] Eine Ausnahme davon bilden natürlich diejenigen Eigenschaften, die wir *direkt* aus s ableiten können. So kann zum Beispiel für jedes konkrete Element $s \in \{\mathtt{i}{:}\mathbb{Z} \mid \mathtt{i}{\ge}0\}$ die Eigenschaft $\mathtt{i}{\ge}0[s/\mathtt{i}]$ mit der Regel `arith` bewiesen werden.

kanonisch (Typen) (Elemente)	**nichtkanonisch**
$\mathsf{set}\{\}(S\,;\,x\,.\,T)$ $\{x\!:\!S \mid T\}$	

Zusätzliche Einträge in die Operatorentabelle

Redex	**Kontraktum**
— *entfällt* —	

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$\{x_1\!:\!S_1 \mid T_1\} = \{x_2\!:\!S_2 \mid T_2\}$	falls	$S_1{=}S_2$ und es gibt eine Variable x, die weder in T_1 noch in T_2 vorkommt, und zwei Terme p_1 und p_2 mit der Eigenschaft $p_1 \in \forall x\!:\!S_1\,.\,T_1[x/x_1] \Rightarrow T_2[x/x_2]$ und $p_2 \in \forall x\!:\!S_1\,.\,T_2[x/x_2] \Rightarrow T_1[x/x_1]$.
$T = \{S_2 \mid T_2\}$	falls	$T = \{x_2\!:\!S_2 \mid T_2\}$ für ein beliebiges $x_2 \in \mathcal{V}$.
$\{S_1 \mid T_1\} = T$	falls	$\{x_1\!:\!S_1 \mid T_1\} = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
Elementsemantik		
$s = t \in \{x\!:\!S \mid T\}$	falls	$\{x\!:\!S \mid T\}$ Typ und $s = t \in S$ und es gibt einen Term p mit der Eigenschaft $p \in T[s/x]$
$\{x_1\!:\!S_1 \mid T_1\} = \{x_2\!:\!S_2 \mid T_2\} \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s/x_1] \in \mathsf{U}_j$ sowie $T_2[s/x_2] \in \mathsf{U}_j$ gilt für alle Terme s mit $s \in S_1$ und es gibt eine Variable x, die weder in T_1 noch in T_2 vorkommt, und zwei Terme p_1 und p_2 mit der Eigenschaft $p_1 \in \forall x\!:\!S_1\,.\,T_1[x/x_1] \Rightarrow T_2[x/x_2]$ und $p_2 \in \forall x\!:\!S_1\,.\,T_2[x/x_2] \Rightarrow T_1[x/x_1]$
$T = \{S_2 \mid T_2\} \in \mathsf{U}_j$	falls	$T = \{x_2\!:\!S_2 \mid T_2\} \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$\{S_1 \mid T_1\} = T \in \mathsf{U}_j$	falls	$\{x_1\!:\!S_1 \mid T_1\} = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash \{x_1\!:\!S_1 \mid T_1\} = \{x_2\!:\!S_2 \mid T_2\} \in \mathsf{U}_j$ [Ax]
by setEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ [Ax]
$\Gamma,\ x\!:\!S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash s = t \in \{x\!:\!S \mid T\}$ [Ax]
by elementEq j
$\Gamma \vdash s = t \in S$ [Ax]
$\Gamma \vdash T[s/x]$ [Ax]
$\Gamma,\ x'\!:\!S \vdash T[x'/x] \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash \{x\!:\!S \mid T\}$ [**ext** s]
by elementI j s
$\Gamma \vdash s \in S$ [Ax]
$\Gamma \vdash T[s/x]$ [Ax]
$\Gamma,\ x'\!:\!S \vdash T[x'/x] \in \mathsf{U}_j$ [Ax]

$\Gamma,\ z\!:\!\{x\!:\!S \mid T\},\ \Delta \vdash C$ [**ext** $(\lambda y\,.\,t)\ z$]
by setE i
$\Gamma,\ z\!:\!\{x\!:\!S \mid T\},\ y\!:\!S,\ [\![v]\!]\!:\!T[y/x],\ \Delta[y/z] \vdash C[y/z]$ [**ext** t]

$\Gamma \vdash s = t \in \{S \mid T\}$ [Ax]
by elementEq_indep
$\Gamma \vdash s = t \in S$ [Ax]
$\Gamma \vdash T$ [Ax]

$\Gamma \vdash \{S \mid T\}$ [**ext** s]
by elementI_indep
$\Gamma \vdash S$ [**ext** s]
$\Gamma \vdash T$ [Ax]

Inferenzregeln

Abbildung 3.22: Syntax, Semantik und Inferenzregeln des Teilmengentyps

Beispiel 3.4.7

Es gibt zwei naheliegende Möglichkeiten, den Typ aller ganzzahligen Funktionen, welche eine Nullstelle besitzen, zu beschreiben.

$$F_0 \equiv \texttt{f:}\mathbb{Z}\to\mathbb{Z} \times \exists\texttt{y:}\mathbb{Z}\texttt{. f y = 0} \qquad \overline{F_0} \equiv \{\texttt{f:}\mathbb{Z}\to\mathbb{Z} \mid \exists\texttt{y:}\mathbb{Z}\texttt{. f y = 0}\}$$

Es ist relativ einfach, einen Algorithmus zu beschreiben, welcher für jedes Element g von F_0 eine Nullstelle bestimmt. Man muß hierzu nur bedenken, daß g in Wirklichkeit aus einem Paar $\langle f\texttt{,}p\rangle$ besteht, wobei p ein Beweis dafür ist, daß f eine Nullstelle besitzt. p selbst hat wiederum die Form $\langle y\texttt{,}p'\rangle$ und y ist genau die gesuchte Nullstelle. Kurzum, der Algorithmus muß nur auf die erste Komponente y der zweiten Komponente p von g zugreifen um das gewünschte Ergebnis zu liefern.

Eine solche Möglichkeit gibt es für die Elemente von $\overline{F_0}$ nicht. Statt eines direkten Zugriffs muß nun die Nullstelle *gesucht* werden, wobei für die Suche keine obere Grenze angegeben werden kann. In der bisherigen Typentheorie kann der allgemeine Algorithmus zur Bestimmung von Nullstellen daher nicht formuliert werden.[58]

In den Teilen eines Beweises, die zum extrahierten Algorithmus etwas beitragen, darf die Eigenschaft $T[s/x]$ also nicht benutzt werden. Die volle Bedeutung der Menge $\{x\texttt{:}S \mid T\}$ würde aber nicht erfaßt werden, wenn wir die Verwendung von $T[s/x]$ generell verbieten würden. In diesem Fall würde eine Deklaration der Form $s\texttt{:}\{x\texttt{:}S \mid T\}$ nämlich nicht mehr Informationen enthalten als die Deklaration $s\texttt{:}S$. Deshalb ist es nötig, die Information $T[s/x]$ bei der Elimination von Mengenvariablen in die Hypothesen mit aufzunehmen, aber in allen Teilbeweisen zu *verstecken*, die einen algorithmischen Anteil besitzen. Nur in 'nichtkonstruktiven' Teilbeweisen, also solchen, die – wie zum Beispiel alle direkten Beweise für Gleichheiten – Axiom als Extrakt-Term liefern, kann die versteckte Hypothese wieder freigegeben werden.

Um dies zu realisieren, müssen wir das Konzept der Sequenz um die Möglichkeit erweitern, Hypothesen zu verstecken und wieder freizugeben. Als Notation für eine versteckte Hypothese schließen wir die zugehörige Variable in (doppelte) eckige Klammern ein, wie zum Beispiel in $\underline{[\![v]\!]\texttt{:}T[y/x]}$. Dies kennzeichnet, daß das *Wissen* $T[y/x]$ vorhanden ist, aber die *Evidenz* v nicht konstruktiv verwendet werden kann. Versteckte Hypothesen werden von der Eliminationsregel für Mengen `setE` erzeugt (und später auch von der Eliminationsregel `quotient_eq`Efür die Gleichheit im Quotiententyp). Versteckte Hypothesen können nur durch die Anwendung von Regeln, welche Axiom als Extrakt-Term erzeugen, wieder *freigegeben* werden. In den erzeugten Teilzielen trägt die Evidenz v nichts zum gesamten Extrakt-Term bei und darf deshalb wieder benutzt werden.

Zugunsten einer einheitlicheren Darstellung wird neben dem üblichen (abhängigen) Teilmengenkonstruktor $\{x\texttt{:}S \mid T\}$ auch eine unabhängige Version $\{S \mid T\}$ eingeführt. Dieser Typ besitzt entweder dieselben Elemente wie S, wenn T nicht leer ist, und ist leer im anderen Fall. Abbildung 3.22 beschreibt die entsprechenden Erweiterungen von Syntax, Semantik und Inferenzregelsystem der Typentheorie.[59] Der besseren Lesbarkeit wegen haben wir in der Beschreibung der Semantik von den in Definition 3.3.3 auf Seite 134 vorgestellten Logikoperatoren gemacht. Dies trägt dem Gedanken Rechnung, daß T_1 bzw. T_2 Typen sind, die als Propositionen aufgefaßt werden. Die in [Constable *et.al.*, 1986, Kapitel 8.2] gegebene Originaldefinition der Semantik von Mengentypen verwendet stattdessen die entsprechenden Funktionstypen.

Zum Abschluß dieses Abschnittes wollen wir das auf Seite 138 begonnene Beispiel der Integerquadratwurzel zu Ende führen. Wir ergänzen hierzu zunächst einige definitorische Abkürzungen, die für eine natürliche Behandlung von Teilbereichen der ganzen Zahlen von Bedeutung sind.

[58] Der Grund hierfür ist, daß die Typentheorie zugunsten der Beweisbarkeit von Eigenschaften nur terminierende Funktionen enthalten darf. Da die Menge der total-rekursiven Funktionen aber unentscheidbar ist, sind die bisher repräsentierbaren Funktionen nur primitiv rekursiv. Das Teilmengenkonstrukt liefert allerdings die Möglichkeit, *partiell-rekursive* Funktionen zu betrachten, die auf einer bekannten Teilmenge des Eingabetyps terminieren. Hierzu müssen wir das Prinzip der rekursiven Definition, welches wir im Prinzip mit dem **Y**-Kombinator (siehe Definition 2.3.22 auf Seite 57) simulieren könnten, genauer untersuchen und seine Eigenschaften so fixieren, daß wir Suchalgorithmen der gewünschten Art formalisieren und ihre Terminierung mithilfe der Information $g \in \{\texttt{f:}\mathbb{Z}\to\mathbb{Z} \mid \exists\texttt{y:}\mathbb{Z}\texttt{. f y = 0}\}$ nachweisen können. Dieses Thema werden wir im Abschnitt 3.5.2 vertiefen.

[59] Man beachte, daß durch die Hinzunahme des Teilmengenkonstruktors jetzt mehrere verschiedene Möglichkeiten zur Charakterisierung eines Termes bestehen. Während zuvor die Struktur der kanonischen Elemente bereits festlegte, zu welcher Art Typ sie gehören können, bleibt jetzt immer noch die Option offen, daß sie auch zu einem anderen Typ gehören, der über den Teilmengentyp gebildet wurde. So ist zum Beispiel `0` sowohl ein Element von $\mathbb{Z}$ als auch eines von $\mathbb{N} \equiv \{\texttt{n:}\mathbb{Z} \mid \texttt{n}\geq\texttt{0}\}$.

Definition 3.4.8 (Teilbereiche der ganzen Zahlen und zusätzliche Operationen)

$\mathbb{N} \equiv$ **nat**{}() $\equiv \{n{:}\mathbb{Z} \mid n{\geq}0\}$
$\mathbb{N}^+ \equiv$ **nat_plus**{}() $\equiv \{n{:}\mathbb{N} \mid n{>}0\}$
$\{i..j\} \equiv$ **int_iseg**{}(i;j) $\equiv \{n{:}\mathbb{Z} \mid i{\leq}n \wedge n{\leq}j\}$
$n^2 \equiv$ **square**{}(n) $\equiv n{*}n$

Beispiel 3.4.9 (Bestimmung der Integerquadratwurzel)

Die einfachste Form, die Existenz einer Integerquadratwurzel über das Theorem

$\vdash \forall x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ y^2{\leq}x \wedge x{<}(y{+}1)^2$

zu beweisen, besteht in einer simplen Induktion über der Zahl x. Falls x=0 ist, so müssen wir ebenfalls y=0 wählen. Andernfalls können wir davon ausgehen, daß wir die Integerquadratwurzel von x-1, die wir mit z bezeichnen, bereits kennen, d.h. es gilt $z^2{\leq}x{-}1 \wedge x{-}1{<}(z{+}1)^2$. Gilt nun auch $x{<}(z{+}1)^2$, dann können wir z unverändert als Integerquadratwurzel von x übernehmen. Ansonsten müssen wir die nächstgrößere Zahl, also z+1 als Integerquadratwurzel von x angeben. Der folgende formale NuPRL Beweis spiegelt dieses Argument genau wieder.

```
⊢ ∀x:ℕ.∃y:ℕ. y²≤x ∧ x<(y+1)²
by all_i
 \
  x:ℕ ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
  by natE 1
  |\
  | x:ℕ ⊢ ∃y:ℕ. y²≤0 ∧ 0<(y+1)²
  | by ex_i 0
  | |\
  | | x:ℕ ⊢ 0²≤0 ∧ 0(0+1)²
  | | by ......... and_i, arith
  | |  \
  | |   x:ℕ ⊢ 0 ∈ ℕ
  | |   by elementEq 1
  | |   |\
  | |   | x:ℕ ⊢ 0 ∈ ℤ
  | |   | by natnumEq
  | |   |\
  | |   | x:ℕ ⊢ 0≥0
  | |   | by arith 1
  | |   |  \
  | |   |   x:ℕ, i:ℤ ⊢ i≥0 ∈ U1
  | |   |   by ......... funEq, voidEq, ltEq, natnumEq, hypEq
  |  \
  |   x:ℕ, n:ℕ, 0<n, v:∃y:ℕ. y²≤n-1 ∧ n-1<(y+1)² ⊢ ∃y:ℕ. y²≤n ∧ n<(y+1)²
  |   \
  |    by ex_e 4 THEN and_e 5
  |     \
  |      x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)² ⊢ ∃y:ℕ. y²≤n ∧ n<(y+1)²
  |      by cut 6  n<(y+1)² ∨ n=(y+1)²
  |      |\            | x:ℕ, n:ℕ, 0<n, y:ℕ,  y²≤n-1, n-1<(y+1)² ⊢ n<(y+1)² ∨ n=(y+1)²
  |      | by ... arith
  |       \
  |        x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n<(y+1)² ∨ n=(y+1)² ⊢ ∃y:ℕ. y²≤n ∧ n<(y+1)²
  |        by or_e 7
  |        |\
  |        | x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n<(y+1)² ⊢ ∃y:ℕ. y²≤n ∧ n<(y+1)²
  |        | by ex_i y
  |        | |\
  |        | | x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n<(y+1)² ⊢  y²≤n ∧ n<(y+1)²
  |        | | by ......... and_i, arith, hyp
  |        |  \
  |        |   x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n<(y+1)² ⊢ y∈ℕ
  |        |   by hypEq 4
  |         \
  |          x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n=(y+1)² ⊢ ∃y:ℕ. y²≤n ∧ n<(y+1)²
  |          by ex_i y+1
  |          |\
  |          | x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n=(y+1)² ⊢ (y+1)²≤n ∧ n<(y+1+1)²
  |          | by ......... and_i, arith
  |          |  \
  |          |   x:ℕ, n:ℕ, 0<n, y:ℕ, y²≤n-1, n-1<(y+1)², n<(y+1)² ⊢ y+1∈ℕ
  |          |   by ...... addEq, hypEq, natnumEq
```

In dem formalen Beweis müssen wir berücksichtigen, daß der Typ IN als Teilmenge der ganzen Zahlen dargestellt wird und daß die Induktion über ganzen Zahlen auch die negativen Zahlen betrachtet, die in IN gar nicht vorkommen. Die Induktion über `x` – also die Regel `natE` aus Abschnitt 3.4.2.1 – besteht daher aus mehreren Einzelschritten: der Elimination der Menge mit `setE`, der Elimination der ganzen Zahl mit `intE` und der vollständigen Behandlung des negativen Induktionsfalls, der durch arithmetisches Schließen mittels `arith` als widersprüchlich nachgewiesen wird. Hierfür sind weitere Detailsschritte erforderlich, welche die versteckte Hypothese freigeben und zu der Induktionsannahme in Beziehung setzen.

Die Schritte, die hierbei ausgeführt werden, sind in allen Fällen dieselben – hängen also nicht vom konkreten Beweisziel ab. Wir können daher die Regel `natE` als eine definitorische Abkürzung für diese Folge von Einzelschritten betrachten, wobei die Zwischenergebnisse nicht gezeigt werden. Wir hätten, wie man leicht sieht, auch noch weitere Schritte des Beweises zusammenfassen können. So ist zum Beispiel die Typüberprüfung $0 \in \mathbb{N}$, die wir etwas ausführlicher beschrieben haben, eine logische Einheit. Das gleiche gilt für die Fallanalyse, die wir mit der `cut`-Regel eingeleitet haben. Faßt man diese Schritte zu einer jeweils größeren Regel zusammen so ist es möglich, den NuPRL Beweis lesbarer und ähnlicher zum informalen Beweis zu gestalten. Den 'Taktik'-Mechanismus, der es uns erlaubt, elementare Inferenzregeln auf diese Art zu kombinieren, werden wir in Abschnitt 4.2 ausführlich besprechen.

Der obige Beweis liefert uns einen Algorithmus, der im wesentlichen die Gestalt

λ`x. ind(x; _,_,_; 0; n,y.if n<(y+1)^2 then y else y+1)`

hat. Dieser Algorithmus ist linear in `x` und der effizienteste Algorithmus, den man mit Mitteln der Induktion bzw. primitiven Rekursion (also FOR-Schleifen) erreichen kann. Es gibt jedoch auch wesentlich effizientere Algorithmen zur Berechnung der Integerquadratwurzel, nämlich zum Beispiel die lineare Suche nach dem kleinsten Wert `y`, für den `(y+1)^2 > x` gilt. Ein solcher Algorithmus verwendet jedoch eine allgemeinere Form der Rekursion, die mit einfacher Induktion nicht mehr dargestellt werden kann. Die bisherige Ausdruckskraft der Typentheorie reicht daher für eine Betrachtung und Erzeugung realistischer Programme immer noch nicht aus und wir werden sie um eine allgemeinere Form der Rekursion erweitern müssen. Diese Erweiterung werden wir im Abschnitt 3.5 besprechen.

3.4.5 Quotienten

Das Teilmengenkonstrukt des vorhergehenden Abschnitts ermöglicht uns, die Typzugehörigkeitsrelation eines vorgegebenen Datentyps durch Restriktion zu verändern. Auf diese Art können wir verschiedenartige Typen wie die ganzen und die natürlichen Zahlen miteinander in Verbindung bringen, ohne eine Konversion von Elementen vornehmen zu müssen. Es gibt jedoch noch eine andere sinnvolle Möglichkeit, zwei verschieden Typen in Beziehung zu setzen, die im Prinzip doch die gleichen Elemente besitzen. So werden zum Beispiel in der Analysis die rationalen Zahlen mit Paaren ganzer Zahlen und die rellen Zahlen mit konvergierenden unendlichen Folgen rationaler Zahlen identifiziert. Ein Unterschied besteht jedoch darin, wann zwei Elemente als gleich zu gelten haben. Während die Gleichheit von Paaren von Zahlen elementeweise bestimmt wird werden zwei rationale Zahlen als gleich betrachtet, wenn die entsprechenden gekürzten Formen gleich sind. Zwei Folgen rationaler Zahlen sind gleich als relle Zahl, wenn sie gegen den gleichen Grenzwert konvergieren.[60] Somit werden zwar jeweils dieselben Elemente betrachtet, aber die Gleichheitsrelation ist verändert.

Mathematisch betrachtet besteht diese Veränderung in einer Restklassenbildung. Der Typ T der betrachteten Elemente bleibt im Prinzip unverändert, aber er wird ergänzt um eine neue Gleichheitsrelation E (*'equality'*), welche ab nun die semantische Gleichheit des neuen Typs bestimmt. Natürlich muß E hierfür tatsächlich eine *Äquivalenzrelation* sein. Die Schreibweise für diesen Typ ist $\underline{x,y : T//E}$, wobei T ein Typ ist und E ein Äquivalenzprädikat (also auch ein Typ), welches von den Variablen $x, y \in T$ abhängen kann. Die Elemente des so entstandenen Typs sind die Elemente von T. Zwei Elemente s und t gelten als gleich, wenn sie in der Relation E stehen, also wenn es einen Beweis für (ein Element von) $E[s,t\,/\,x,y]$ gibt.

[60] Diese Gleichheiten können natürlich alleine auf der Basis der Eigenschaften von Zahlenpaaren bzw. von Folgen rationaler Zahlen definiert werden: ⟨6,4⟩=⟨9,6⟩ gilt, weil `6*6=4*9` ist.

kanonisch (Typen) (Elemente)	**nichtkanonisch**
quotient{}$(T\,;\,x\,,y\,.\,E)$ $x\,,y : T//E$	

Zusätzliche Einträge in die Operatorentabelle

Redex	**Kontraktum**
— *entfällt* —	

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$x_1\,,y_1 : T_1//E_1 \;= x_2\,,y_2 : T_2//E_2$	falls	$T_1 = T_2$ und es gibt (verschiedene) Variablen x, y, z, die weder in E_1 noch in E_2 vorkommen, und Terme p_1, p_2, r,s und t mit der Eigenschaft $p_1 \in \forall x{:}T_1\,.\forall y{:}T_1\,.\; E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ und $p_2 \in \forall x{:}T_1\,.\forall y{:}T_1\,.\; E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ und $r \in \forall x{:}T_1\,.\, E_1[x,x/x_1,y_1]$ und $s \in \forall x{:}T_1\,.\forall y{:}T_1\,.\; E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$ und $t \in \forall x{:}T_1\,.\forall y{:}T_1\,.\forall z{:}T_1\,.$ $E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$
Elementsemantik		
$s = t \;\in\; x\,,y : T//E$	falls	$x\,,y : T//E$ Typ und $s \in T$ und $t \in T$ und es gibt einen Term p mit der Eigenschaft $p \in E[s,t/x,y]$
$x_1\,,y_1 : T_1//E_1 \;= x_2\,,y_2 : T_2//E_2 \;\in \mathsf{U}_j$	falls	$T_1 = T_2 \in \mathsf{U}_j$ und für alle Terme s,t mit $s \in T_1$ und $t \in T_1$ gilt $E_1[s,t/x_1,y_1] \in \mathsf{U}_j$ sowie $E_2[s,t/x_2,y_2] \in \mathsf{U}_j$ und es gibt (verschiedene) Variablen x, y, z, die weder in E_1 noch in E_2 vorkommen, und Terme p_1, p_2, r,s und t mit der Eigenschaft $p_1 \in \forall x{:}T_1\,.\forall y{:}T_1\,.\; E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ und $p_2 \in \forall x{:}T_1\,.\forall y{:}T_1\,.\; E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ und $r \in \forall x{:}T_1\,.\, E_1[x,x/x_1,y_1]$ und $s \in \forall x{:}T_1\,.\forall y{:}T_1\,.\; E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$ und $t \in \forall x{:}T_1\,.\forall y{:}T_1\,.\forall z{:}T_1\,.$ $E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$

Zusätzliche Einträge in den Semantiktabellen

Abbildung 3.23: Syntax und Semantik des Quotiententyps

Bei der Formalisierung derartiger *Quotiententypen* ist jedoch zu beachten, daß die Evidenz für die veränderte Gleichheit – wie die Evidenz für die Einschränkung bei Teilmengen – nicht als Bestandteil des Elements selbst mitgeführt wird, sondern eine 'abstrakte' Eigenschaft ist. Das Wissen um die veränderte Gleichheit ist in $s = t \;\in\; x\,,y : T//E$ also nur implizit enthalten und darf nicht zu einem Algorithmus beitragen, der aus einem entsprechenden Beweis enthalten ist. Daher muß die Regel, welche Gleichheiten in Kombination mit Quotiententypen analysiert (`quotient_eqE`) ebenfalls eine versteckte Hypothese generieren, welche erst durch Regeln freigegeben wird, deren Extrakt-Term Axiom ist. Alle anderen Bestandteile des Quotiententyps, die in den Abbildungen 3.23 und 3.24 beschrieben sind, ergeben sich direkt aus einer Ausformulierung des Konzeptes der Restklassen unter gegebenen Äquivalenzrelationen.[61]

Ein typisches Beispiel für die Anwendung des Quotiententyps ist die Formalisierung der rationalen Zahlen. Die folgende Definition entstammt [Constable *et.al.*, 1986, Kapitel 11.5] und wird dort ausführlicher diskutiert.

[61] Es sei angemerkt, daß zugunsten des einfacheren Übergangs zwischen T und $x\,,y : T//E$ zwei "schwache" Regeln ergänzt wurden, in denen die Eigenschaften der Restklassenbildung nur zu einem geringen Teil ausgenutzt werden. Diese Regeln "vergeuden" Informationen in dem Sinne, daß die erzeugten Teilziele viel mehr beweisen als im ursprünglichen Ziel gefordert wurde.

$\Gamma \vdash x_1, y_1 : T_1//E_1 \;=\; x_2, y_2 : T_2//E_2 \;\in\; \mathsf{U}_j$ [Ax]
by quotientEq_weak
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]
$\Gamma,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,y/x_1,y_1] = E_2[x,y/x_2,y_2] \in \mathsf{U}_j$ [Ax]
$\Gamma,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,x/x_1,y_1]$ [Ax]
$\Gamma,\ x{:}T_1,\ y{:}T_1,\ v{:}E_1[x,y/x_1,y_1] \vdash E_1[y,x/x_1,y_1]$ [Ax]
$\Gamma,\ x{:}T_1,\ y{:}T_1,\ z{:}T_1,\ v{:}E_1[x,y/x_1,y_1],\ v'{:}E_1[y,z/x_1,y_1] \vdash E_1[x,z/x_1,y_1]$ [Ax]

$\Gamma \vdash x_1, y_1 : T_1//E_1 \;=\; x_2, y_2 : T_2//E_2 \;\in\; \mathsf{U}_j$ [Ax]
by quotientEq
$\Gamma \vdash x_1, y_1 : T_1//E_1 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash x_2, y_2 : T_2//E_2 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]
$\Gamma,\ v{:}\ T_1 = T_2 \in \mathsf{U}_j,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ [Ax]
$\Gamma,\ v{:}\ T_1 = T_2 \in \mathsf{U}_j,\ x{:}T_1,\ y{:}T_1 \vdash E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ [Ax]

$\Gamma \vdash s = t \in x, y : T//E$ [Ax]
by memberEq_weak j
$\Gamma \vdash x, y : T//E \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash s = t \in T$ [Ax]

$\Gamma \vdash x, y : T//E$ [**ext** t]
by memberI j
$\Gamma \vdash x, y : T//E \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T$ [**ext** t]

$\Gamma \vdash s = t \in x, y : T//E$ [Ax]
by memberEq j
$\Gamma \vdash x, y : T//E \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash s \in T$ [Ax]
$\Gamma \vdash t \in T$ [Ax]
$\Gamma \vdash E[s,t/x,y]$ [Ax]

$\Gamma,\ v{:}\ s{=}t \in x, y : T//E,\ \Delta \vdash C$ [**ext** u]
by quotient_eqE i j
$\Gamma,\ v{:}\ s{=}t \in x, y : T//E,\ [\![v']\!]{:}E[s,t/x,y],\ \Delta \vdash C$ [**ext** u]
$\Gamma,\ v{:}\ s{=}t \in x, y : T//E,\ \Delta,\ x'{:}T,\ y'{:}T \vdash E[x',y'/x,y] \in \mathsf{U}_j$ [Ax]

$\Gamma,\ z{:}\,x, y : T//E,\ \Delta \vdash s{=}t \in S$ [Ax]
by quotientE i j
$\Gamma,\ z{:}\,x, y : T//E,\ \Delta,\ x'{:}T,\ y'{:}T \vdash E[x',y'/x,y] \in \mathsf{U}_j$ [Ax]
$\Gamma,\ z{:}\,x, y : T//E,\ \Delta \vdash S \in \mathsf{U}_j$ [Ax]
$\Gamma,\ z{:}\,x, y : T//E,\ \Delta,\ x'{:}T,\ y'{:}T,\ v{:}E[x',y'/x,y] \vdash s[x'/z] = t[y'/z] \in S[x'/z]$ [Ax]

Abbildung 3.24: Inferenzregeln des Quotiententyps

Definition 3.4.10 (Rationale Zahlen)

$x_1 =_q x_2 \equiv \mathbf{rat\_equal}\{\}(x_1;x_2) \equiv \mathsf{let}\ \langle z_1, n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2, n_2\rangle = x_2\ \mathsf{in}\ z_1{*}n_2 = z_2{*}n_1$
$\mathbb{Q} \equiv \mathbf{rat}\{\}() \equiv \mathtt{x,y} : \mathbb{Z}\times\mathbb{N}^+//_{\mathtt{x}=_q\mathtt{y}}$
$x_1{+}x_2 \equiv \mathbf{rat\_add}\{\}(x_1;x_2) \equiv \mathsf{let}\ \langle z_1, n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2, n_2\rangle = x_2\ \mathsf{in}\ \langle z_1{*}n_2{+}z_2{*}n_1, n_1{*}n_2\rangle$
$x_1{-}x_2 \equiv \mathbf{rat\_sub}\{\}(x_1;x_2) \equiv \mathsf{let}\ \langle z_1, n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2, n_2\rangle = x_2\ \mathsf{in}\ \langle z_1{*}n_2{-}z_2{*}n_1, n_1{*}n_2\rangle$
$x_1{*}x_2 \equiv \mathbf{rat\_mul}\{\}(x_1;x_2) \equiv \mathsf{let}\ \langle z_1, n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2, n_2\rangle = x_2\ \mathsf{in}\ \langle z_1{*}z_2, n_1{*}n_2\rangle$

Aus mathematischer Sicht ist der Quotiententyp ein sehr mächtiger Abstraktionsmechanismus, denn er ermöglicht, *benutzerdefinierte Gleichheiten* in die Theorie auf eine Art mit aufzunehmen, daß alle Gleichheitsregeln – insbesondere die Substitutionsregel und eine Entscheidungsprozedur für Gleichheit (die `equality` Regel in Abbildung 3.28) – hierauf anwendbar werden. Dieser Mechanismus muß jedoch mit großer Sorgfalt eingesetzt werden, da die Gleichheit von kanonischen Elementen eines Quotiententyps – im Gegensatz zur Elementgleichheit bei anderen Typkonstrukten – nicht mehr von der Struktur dieser Elemente abhängt sondern von der benutzerdefinierten Gleichheitsrelation.[62] Somit müssen Konstruktionen, an denen Objekte eines Quotiententyps beteiligt sind, unabhängig von der speziellen Darstellung dieser Objekte sein. Genauer gesagt, ein Typ $T[z]$, der von einer Variablen z eines Quotiententyps $Q \equiv x, y : S//E$ abhängt, muß so gestaltet sein, daß eine spezielle Instanz $T[s/z]$ nicht davon abhängt, welchen Term $s \in S$ man gewählt hat: für gleiche Elemente s_1, s_2 von Q – also Elemente, für die $E[s_1, s_2 / x, y]$ gilt – müssen $T[s_1/z]$ und $T[s_2/z]$ gleich sein.

[62] Die uns vertraute Mathematik geht mit dem Konzept der Restklassen – wie im Falle der Teilmengen – leider etwas zu sorglos um. Bei einer vollständigen Formalisierung fallen die kleinen Unstimmigkeiten allerdings auf und müssen entsprechend behandelt werden. Deshalb sind in beiden Fällen die Formalisierungen komplizierter als dies dem intuitiven Verständnis nach sein müsste.

Bei Typen $T[z]$, die tatsächlich einen Datentyp darstellen, ist dies normalerweise kein Problem, da sich hier das intuitive Verständnis mit der formalen Repräsentation deckt. Anders wird dies jedoch, wenn $T[z]$ eine logische Aussage darstellt. Üblicherweise verbindet man hiermit dann nur einen Wahrheitswert und vergißt, daß die formale Repräsentation auch die *Struktur* der Beweise wiedergibt, die durchaus von der speziellen Instanz für die Variable z abhängen kann. Wir wollen dies an einem Beispiel illustrieren.

Beispiel 3.4.11

In Definition 3.4.10 haben wir die rationalen Zahlen über Paare von ganzen und positiven Zahlen definiert, wobei wir die Definition der Gleicheit durch "Ausmultiplizieren" auf die Gleichheit ganzer Zahlen zurückgeführt haben. In gleicher Weise könnte man nun versuchen, andere wichtige Vergleichsoperationen wie die Relation $<$ auf rationale Zahlen fortzusetzen, was zu folgender Definition führen würde:

$$x_1 \mathtt{<} x_2 \;\equiv\; \mathsf{let}\ \langle z_1 \mathtt{,} n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2 \mathtt{,} n_2\rangle = x_2\ \mathsf{in}\ \ z_1 \mathtt{*} n_2 \mathtt{<} z_2 \mathtt{*} n_1$$

Im Gegensatz zu dem intuitiven Verständnis der Relation $<$ haben wir bei dieser Definition jedoch eine Struktur aufgebaut und nicht etwa nur ein Prädikat, das nur wahr oder falsch sein kann. Wie sieht es nun mit der Unabhängigkeit dieser Struktur von der Darstellung rationaler Zahlen aus? Gilt das Urteil $x_1 \mathtt{<} x_2 = x_1' \mathtt{<} x_2'$, wenn $x_1 = x_1' \in \mathbb{Q}$ und $x_2 = x_2' \in \mathbb{Q}$ gilt?

Um dies zu überprüfen, müssen wir kanonische Terme von $\mathbb{Q}$ betrachten, das spread-Konstrukt reduzieren und dann die Typsemantik der Relation < auf ganzen Zahlen berücksichtigen. Gemäß dem Eintrag in Abbildung 3.17 gilt jedoch

$$z_1 \mathtt{*} n_2 \mathtt{<} z_2 \mathtt{*} n_1 \;=\; z_1' \mathtt{*} n_2' \mathtt{<} z_2' \mathtt{*} n_1'$$

nur, wenn die jeweiligen Terme links und rechts vom Symbol < als ganze Zahlen gleich sind, also wenn

$$z_1 \mathtt{*} n_2 \;=\; z_1' \mathtt{*} n_2' \;\in\; \mathbb{Z} \quad \text{und} \quad z_2 \mathtt{*} n_1 \;=\; z_2' \mathtt{*} n_1' \;\in\; \mathbb{Z} \quad \text{gilt.}$$

Daß dies nicht immer der Fall ist, zeigt das Beispiel $x_1 = \langle \mathtt{2,1} \rangle$, $x_1' = \langle \mathtt{4,2} \rangle$, $x_2 = x_2' = \langle \mathtt{1,1} \rangle$, denn es gilt weder $\mathtt{2*1 = 4*1} \in \mathbb{Z}$ noch $\mathtt{1*1 = 1*2} \in \mathbb{Z}$

Diese semantische Analyse macht klar, daß Beweisziele der Art $x_1 \mathtt{:} \mathbb{Q}\mathtt{,}\ x_2 \mathtt{:} \mathbb{Q} \vdash x_1 \mathtt{<} x_2 \in \mathsf{U}_1$ unter Verwendung der obigen Definition nicht bewiesen werden können, weil die Regel `quotientE`, die irgenwann im Verlauf des Beweises angewandt werden *muß*, genau das obige Problem aufdeckt.

Wie kann man dieses Problem nun lösen? Sicherlich wäre es nicht sinnvoll, die Semantik der Relation < auf ganzen Zahlen so abzuschwächen, daß Typgleichheit $i_1 \mathtt{<} j_1 = i_2 \mathtt{<} j_2$ gilt, wenn entweder beide Relationen erfüllt oder beide Relationen falsch sind, denn dies würde nur das spezielle Problem aus Beispiel 3.4.11 lösen und wir hätten dasselbe Problem bei anderen Prädikaten, die über die Relation < auf rationalen Zahlen definiert werden.[63] Wir müssen stattdessen überlegen, wie wir die Überstrukturierung vermeiden können, die wir bei der Definition in Beispiel 3.4.11 erzeugt haben.

Einen sehr einfachen Weg, einen strukturierten Datentyp P, der eigentlich nur eine logische Aussage repräsentieren soll, in einen unstrukturierten Datentyp umzuwandeln, bietet die Verwendung des unabhängigen Teilmengenkonstrukts. Der Typ $\{\mathtt{0} \in \mathbb{Z} \mid P\}$ besteht aus den Elementen von $\mathtt{0} \in \mathbb{Z}$ (d.h. aus Axiom), falls P Elemente hat, und ist andernfalls leer. Die spezifische Information, auf welche Art P die Elemente von P zu konstruieren sind, wird unterdrückt und spielt entsprechend der Semantik des Teilmengenkonstrukts in Abbildung 3.22 auch bei der Typgleichheit keine Rolle mehr. Dies deckt sich genau mit der Sichtweise von P als eine logische Aussage, die entweder wahr (beweisbar) oder falsch (unbeweisbar) ist. Diese Technik, durch Verwendung des unabhängigen Teilmengenkonstrukts die Struktur aus einem Datentyp P zu entfernen und diesen auf 'Wahrheit' zu reduzieren, nennt man *Type-Squashing* (Zerdrücken eines Typs). Sie wird durch die folgende konservative Erweiterung des Typsystems unterstützt.

Definition 3.4.12 (Type-Squashing)

$$\|T\| \;\equiv\; \mathsf{squash}\{\}(T) \;\equiv\; \{\mathtt{0} \in \mathbb{Z} \mid T\}$$

[63] Außerdem würde eines der Grundkonzepte der Typentheorie verletzt, Gleichheit *strukturell* zu definieren, d.h. von der Gleichheit aller beteiligter Teilterme abhängig zu machen.

Type-Squashing bietet sich an, wenn wir Prädikate über einem Quotiententyp $x,y : S//E$ definieren wollen. Durch die Verwendung des Quotiententyps haben wir als Benutzer der Theorie die Gleichheit auf dem zugrundeliegenden Datentyp S verändert. Daher müssen wir als Benutzer bei der Einführung von Prädikaten auch selbst dafür sorgen, daß diese tatsächlich Prädikate über dem selbstdefinierten Quotiententyp bilden.[64]

Wir wollen die Verwendung von Type-Squashing am Beispiel einer Formalisierung der reellen Zahlen in der Typentheorie illustrieren. Diese kann man durch Cauchy-Folgen rationaler Zahlen beschreiben, also durch unendliche Folgen rationaler Zahlen (Funktionen von $\mathbb{N}^+$ nach $\mathbb{Q}$), deren Abstände gegen Null konvergieren. Um dies zu präzisieren, müssen wir zunächst die Relation < auf rationalen Zahlen geeignet definieren.

Definition 3.4.13 (Reelle Zahlen)

$$
\begin{array}{lll}
x_1{<}x_2 & \equiv \textbf{rat\_lt}\{\}(x_1;x_2) & \equiv \|\mathsf{let}\ \langle z_1,n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2,n_2\rangle = x_2\ \mathsf{in}\ z_1{*}n_2{<}z_2{*}n_1\| \\
x_1{\leq}x_2 & \equiv \textbf{rat\_le}\{\}(x_1;x_2) & \equiv x_1{<}x_2\ \vee\ x_1 = x_2 \in \mathbb{Q} \\
z/n & \equiv \textbf{rat\_frac}\{\}(z;n) & \equiv \langle z,n\rangle \\
|x| & \equiv \textbf{rat\_abs}\{\}(x) & \equiv \mathsf{let}\ \langle z,n\rangle = x\ \mathsf{in}\ \mathsf{if}\ z{<}0\ \mathsf{then}\ \langle -z,n\rangle\ \mathsf{else}\ \langle z,n\rangle \\
\mathbb{R}_{pre} & \equiv \textbf{real\_pre}\{\}() & \equiv \{\mathtt{f}:\mathbb{N}^+{\to}\mathbb{Q} \mid \forall \mathtt{m},\mathtt{n}:\mathbb{N}^+.\ |\mathtt{f(n)} - \mathtt{f(m)}| \leq \mathtt{1/m{+}1/n}\} \\
x_1{=_r}x_2 & \equiv \textbf{real\_equal}\{\}(x_1;x_2) & \equiv \forall \mathtt{n}:\mathbb{N}^+.\ |x_1(\mathtt{n}) - x_2(\mathtt{n})| \leq \mathtt{2/n} \\
\mathbb{R} & \equiv \textbf{real}\{\}() & \equiv \mathtt{x,y} : \mathbb{R}_{pre}//_{\mathtt{x}=_r\mathtt{y}} \\
x_1{+}x_2 & \equiv \textbf{real\_add}\{\}(x_1;x_2) & \equiv \lambda\mathtt{n}.\ x_1(\mathtt{n}){+}x_2(\mathtt{n}) \\
x_1{-}x_2 & \equiv \textbf{real\_sub}\{\}(x_1;x_2) & \equiv \lambda\mathtt{n}.\ x_1(\mathtt{n}){-}x_2(\mathtt{n}) \\
|x| & \equiv \textbf{real\_abs}\{\}(x) & \equiv \lambda\mathtt{n}.\ |x(\mathtt{n})|
\end{array}
$$

Auch den Datentyp $\mathbb{B}$ könnte man als Restklasse der ganzen Zahlen betrachten und definieren:

$$\mathbb{B} \equiv \mathtt{i,j} : \mathbb{Z}//(\mathtt{i\ rem\ 2\ =\ j\ rem\ 2})$$

Gerade Zahlen stünden in diesem Fall für **T** und ungerade Zahlen für **F**. Wegen der gedanklichen Verwandschaft zur disjunkten Vereinigung wird $\mathbb{B}$ jedoch üblicherweise als Summe zweier einelementiger Typen definiert.

3.4.6 Strings

Der Vollständigkeit halber enthält die Typentheorie einen Datentyp der Strings, der mit Atom bezeichnet wird. Dadurch wird es möglich, Programme zu betrachten, die feste Textketten als Antworten oder Meldungen ausgeben, ohne diese weiter zu verarbeiten. Die kanonischen Elemente des Typs Atom sind daher einfache Textketten, die in (doppelte) Anführungszeichen gesetzt sind.[65] Die Gleichheit zweier Strings ist leicht zu entscheiden: sie müssen textlich identisch sein. Zur Analyse steht ein Test auf Gleichheit (if $u{=}v$ then s else t) zur Verfügung. Die zugehörigen Ergänzungen des Typsystems sind in Abbildung 3.25 zusammengestellt.

3.5 Rekursion in der Typentheorie

Im vorigen Abschnitt haben wir den Zusammenhang zwischen der Konstruktion formaler Beweise und der Entwicklung von Programmen hervorgehoben und die Vorteile dieser Denkweise illustriert. Dabei stellte sich natürlich heraus, daß die Mächtigkeit und Eleganz der so erzeugten Programme durch die Ausdruckskraft des zugrundeliegenden Inferenzkalküls beschränkt ist. Solange man sich auf reine Prädikatenlogik beschränkt, kann man nur Programme generieren, die aus elementaren Substitutionen und Fallunterscheidungen durch einfache Kompositionen zusammengesetzt sind. Durch die Hinzunahme von Induktionsbeweisen auf Zahlen

[64] Es sei allerdings angemerkt, daß das Mittel des Type-Squashing sehr grob ist und eigentlich zu viele Informationen unterdrückt. Wenn wir Teilinformationen eines Prädikats in Analysen verwenden wollen, die zu einem extrahierten Algorithmus beitragen, dann können wir Type-Squashing in seiner allgemeinen Form nicht verwenden sondern müssen die Struktur zum Teil freigeben und nur die Teile unterdrücken, die wirklich nicht benötigt werden und Störeffekte hervorrufen.

[65] Diese '*String-quotes*' ¨ sind zu unterscheiden von den sogenannten '*Token-quotes*' ‘, welche bei der Angabe von Namen in manchen Regeln benötigt werden.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
Atom{}()	**token**{*string*:t}()	**atom_eq**(⌈*u*⌉; ⌈*v*⌉; *s*; *t*)
Atom	¨*string*¨	if ⌈*u*⌉=⌈*v*⌉ then *s* else *t*

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
if u=v then s else t	$\xrightarrow{\beta}$	*s, falls u = v; ansonsten t*

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik
Atom = Atom

Elementsemantik
¨*string*¨ = ¨*string*¨ ∈ Atom
Atom = Atom ∈ U_j

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash$ Atom = Atom ∈ U_j [Ax]
by atomEq

$\Gamma \vdash$ ¨*string*¨ ∈ Atom [Ax]
by tokEq

$\Gamma \vdash$ Atom ⌊**ext** ¨*string*¨⌋
by tokI ¨*string*¨

$\Gamma \vdash$ if u_1=v_1 then s_1 else t_1
= if u_2=v_2 then s_2 else t_2 ∈ T [Ax]
by atom_eqEq
$\Gamma \vdash u_1$=u_2 ∈ Atom [Ax]
$\Gamma \vdash v_1$=v_2 ∈ Atom [Ax]
Γ, v: u_1=v_1 ∈ Atom $\vdash s_1 = s_2 \in T$ [Ax]
Γ, v: ¬(u_1=v_1 ∈ Atom) $\vdash t_1 = t_2 \in T$ [Ax]

$\Gamma \vdash$ if u=v then s else $t = t_2 \in T$ [Ax]
by atom_eqRedT
$\Gamma \vdash s = t_2 \in T$ [Ax]
$\Gamma \vdash u$=v ∈ Atom [Ax]

$\Gamma \vdash$ if u=v then s else $t = t_2 \in T$ [Ax]
by atom_eqRedF
$\Gamma \vdash t = t_2 \in T$ [Ax]
$\Gamma \vdash$ ¬(u=v ∈ Atom) [Ax]

Inferenzregeln

Abbildung 3.25: Syntax, Semantik und Inferenzregeln des Typs Atom

und Listen kommen elementare arithmetische Operationen und primitive Rekursion hinzu. Über das Niveau der primitiv-rekursiven Funktionen kommt man mit den bisherigen Typkonzepten jedoch nicht hinaus.

Rein theoretisch ist dies zwar keine signifikante Einschränkung, da es so gut wie keine (terminierenden) Programme gibt, deren Effekt sich nicht auch mit den Mitteln der primitiv-rekursiven Funktionen beschreiben läßt. Jedoch haben primitiv-rekursive Funktionen gegenüber den Programmen, die in der Praxis eingesetzt werden, den Nachteil einer erheblich geringeren Eleganz und – wie am Ende von Beispiel 3.4.9 (Seite 154) bereits angedeutet – einer wesentlich schlechteren Komplexität. Dies liegt daran, daß primitiv-rekursive Funktionen – zu der Abarbeitung einer Zählschleife – ausschließlich eine Reduktion des Eingabeargumentes in Einzelschritten zulassen. Komplexere Schritte, eine vorzeitige Beendigung der Rekursion, eine Rekursion auf der Ergebnisvariablen oder gar eine Rekursion ohne eine vorher angegebene obere Schranke für die Anzahl der Rekursionsschritte – also die Charakteristika der üblicherweise eingesetzten Schleifen – lassen sich nicht unmittelbar ausdrücken. Zwar ist es möglich, die ersten drei Aspekte zu simulieren, aber diese Simulation bringt nur scheinbare Vorteile, da die ausdrucksstärkeren Programmierkonstrukte nur sehr ineffizient dargestellt werden können.[66]

Die primitive Rekursion trägt daher ihren Namen "primitiv" zu recht. Die allgemeine Rekursion, die in der Mathematik als auch in der Programmierung verwendet wird, ist an die obengenannten Einschränkungen nicht gebunden. Im Bezug auf Ausdruckskraft, Eleganz und Effizienz wird sie daher von keinem anderen mathematisch-programmiertechnischen Konstrukt übertroffen. Die Möglichkeit, Algorithmen oder mathematische Objekte durch rekursive Gleichungen zu definieren ist essentiell für den Aufbau mathematischer Theorien und die Entwicklung realistischer Programme.

Rekursion birgt jedoch auch Gefahren und Trugschlüsse in sich. Algorithmen können sich endlos rekursiv aufrufen, ohne jemals ein Ergebnis zu liefern. Mathematische Konstruktionen können nicht 'wohlfundiert' sein, weil die Rekursion in sich zwar schlüssig ist, es aber keinen Anfangspunkt gibt, auf dem man sie abstützen kann. Da es bekanntermaßen keine allgemeinen Verfahren gibt, derartige Gefahren auszuschließen,[67] ist ein das logische Schließen über Ausdrücke, die allgemeine Rekursionen enthalten, relativ schwer zu formalisieren, wenn wir zugunsten einer Rechnerunterstützbarkeit des Inferenzsystems garantieren wollen, daß jeder (typisierbare) Ausdruck einem Wert entspricht. Eine (aus theoretischer Sicht) vollständige Einbettung von Rekursion in formale Kalküle ist bisher nicht bekannt und wird vielleicht auch nie zu erreichen sein.

Im folgenden werden wir drei Erweiterungen der Typentheorie um rekursive Definitionen vorstellen, von denen zur Zeit allerdings nur die ersten beiden auch innerhalb des NuPRL Systems implementiert wurden. Die Unterschiede liegen in der Art der Objekte, die durch eine rekursive Gleichung definiert werden.

- Die *induktiven Typkonstruktoren*, die wir in Abschnitt 3.5.1 vorstellen, ermöglichen ein Schließen über rekursiv definierte Datentypen und deren Elemente. Hierbei muß die Rekursionsgleichung allerdings *wohlfundiert* sein, d.h. es muß möglich sein, aus der Rekursionsgleichung 'auszusteigen' und hierbei einen wohldefinierten Typ zu erhalten. Ein typisches Beispiel hierfür ist die Definition 2.2.4 von Formeln der Prädikatenlogik (Seite 24), welche besagt, daß Formeln entweder atomare Formeln sind, oder aus anderen Formeln durch Negation, Disjunktion, Konjunktion, Implikation oder Quantoren zu bilden sind.

- Die Behandlung *partiell rekursiver Funktionen* innerhalb einer Theorie, welche keine nichtterminierenden Reduktionen zuläßt, ist das Thema des Abschnitts 3.5.2. Die Schlüsselidee lautet hierbei, ausschließlich solche Funktionen zu betrachten, bei denen man auf der Grundlage ihrer rekursiven Definition einen Definitionsbereich bestimmen kann, auf dem sie garantiert terminieren. So wird eine Möglichkeit eröffnet, die volle Ausdruckskraft und Eleganz der üblichen Programmiersprachen innerhalb der Typentheorie wiederzuspiegeln, und dennoch die Probleme einzuschränken, die der allgemeine λ-Kalkül mit sich bringt.

[66] Da die primitive Rekursion nach wie vor die Reduktion einer Eingabe in einzelnen Schritten verlangt und keine der fest vordefinierten arithmetischen Operationen mehr als eine konstante Verringerung der Eingabe ermöglicht, können primitiv-rekursive Funktionen bestenfalls in linearer Zeit, gemessen an der Größe der Eingabe, berechnet werden. Logarithmische Zeit ist unerreichbar, obwohl viele praktische Probleme in logarithmischer Zeit lösbar sind.

[67] Die Unentscheidbarkeit des Halteproblems macht es unmöglich, ein allgemeines Verfahren anzugeben, welches nichtterminierende Algorithmen identifiziert bzw. unfundierte Rekursionen als solche entdeckt.

- Die *lässigen Typkonstruktoren*, die wir in Abschnitt 3.5.3 kurz andiskutieren werden, eröffnen eine Möglichkeit zum Schließen über unendliche Objekte wie zum Beispiel Prozesse, die ständig aktiv sind. Der Unterschied zu den induktiven Typkonstruktoren liegt darin, daß auch Gleichungen betrachtet werden können, die keine Ausstiegsmöglichkeit bieten. Ein typisches Beispiel hierfür ist die Definition eines Datenstroms (stream), die besagt, daß ein Datenstrom aus einem Zeichen, gefolgt von einem Datenstrom, besteht. Eine solche Definition kann nur durch unendliche Objekte interpretiert werden.

Induktive und lässige Datentypen sind eine Formalisierung der Grundideen rekursiver Datenstrukturen, die zum Beispiel in [Hoare, 1975] und [Gordon *et.al.*, 1979] ausführlich diskutiert werden. Wir werden im folgenden nur die Grundgedanken dieser drei Ansätze und das zugehörige Inferenzsystem vorstellen. Eine Vertiefung und weitere Details findet man in [Constable & Mendler, 1985, Mendler, 1987a, Constable & Smith, 1987, Constable & Smith, 1988, Mendler *et.al.*, 1986, Mendler, 1987b, Smith, 1988].

3.5.1 Induktive Typen

In den bisherigen Abschnitten haben wir bereits eine Reihe rekursiver Definitionen zur Beschreibung der formalen Sprache der Kalküle benutzt. So war zum Beispiel die Prädikatenlogik (Definition 2.2.4 auf Seite 24) rekursiv über atomare Formeln sowie Negation, Disjunktion, Konjunktion, Implikation und Quantoren definiert. λ-Terme (Definition 2.3.2 auf Seite 48) waren rekursiv über Variablen, Abstraktion und Applikation eingeführt worden. Die Terme der Typentheorie (Definition 3.2.5 auf Seite 104) wurden rekursiv über Operatoren und gebundene Terme erklärt. All diesen Definitionen ist gemeinsam, daß sie die zu erklärende Klasse von Objekten rekursiv durch eine Reihe von Bedingungen definieren. Dabei wird implizit festgelegt, daß die kleinstmögliche Klasse betrachtet werden soll, welche diese Bedingungen erfüllt. Dies bedeutet also, daß das die Klasse *induktiv* definiert wird: es wird eine Art Basisfall definiert und angegeben, wie aus bereits konstruierten Objekten ein neues Objekt aufgebaut werden darf. Für die Klasse der Formeln sind zum Beispiel die atomaren Formel der Basisfall. Für die Klasse der λ-Terme sind es die Variablen.

Die bisherigen rekursiven Definitionen waren informaler Natur. Will man nun typentheoretische Objekte durch rekursive Definitionen erklären, so bietet es sich an, hierzu rekursive Gleichungen zu verwenden und festzulegen, daß hierbei das kleinste Objekt definiert werden soll, welches diese Gleichungen erfüllt. Wir wollen dies an einem einfachen Beispiel illustrieren.

Beispiel 3.5.1 (Binärbäume über ganzen Zahlen)
Eines der Standardbeispiele für rekursiv definierte Datentypen sind Binärbäume über den ganzen Zahlen. Üblicherweise definiert man sie wie folgt.

> *Ein Binärbaum besteht entweder aus einer ganzen Zahl i oder einem Tripel (i, t_l, t_r), wobei t_l und t_r Binärbäume sind und i eine ganze Zahl ist.*

Damit besteht der Datentyp `bintree` aller Binärbäume entweder aus dem Typ $\mathbb{Z}$ der ganzen Zahlen oder dem Produktraum $\mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$. Dies läßt sich formal durch die Gleichung

$$\texttt{bintree} = \mathbb{Z} + \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$$

ausdrücken. Im Gegensatz zu dem oben angegegebenen Text läßt diese Gleichung aber noch mehrere Interpretationen zu, da sie ja nicht festlegt, daß wir die *kleinste* Menge `bintree` betrachten wollen, welche sie erfüllt. Auch unendliche Binärbäume erfüllen diese Gleichung, denn sie bestehen aus einer Integerwurzel und zwei unendlichen Binärbäumen. Deshalb muß bei der Einführung rekursiver Datentypen nicht nur die syntaktische Form sondern gleichzeitig die Semantik fixiert werden. Die naheliegendste Semantik ist die obenangegebene Interpretation als die kleinste Menge, welche eine Gleichung erfüllt. Dies bedeutet in unserem Fall, daß sich jedes Objekt in endlich vielen Schritten durch die in der Gleichung vorkommenden Fälle erzeugen läßt. In NuPRL bezeichnen wir den Term, der mit dieser Semantik identifiziert wird, als *induktiven* Datentyp und schreiben

$$\texttt{rectype bintree} = \mathbb{Z} + \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$$

Terme, die eine andere Semantik rekursiver Gleichungen repräsentieren, werden wir in Abschnitt 3.5.3 kurz ansprechen.

Induktive Datentypen der Form $\underline{\mathsf{rectype}\ X = T_X}$ besitzen also eine Semantik, die – ähnlich wie bei der Definition rekursiver Funktionen im λ-Kalkül auf Seite 57 – durch den *kleinsten* Fixpunkt des Terms T_X auf der rechten Seite der Definitionsgleichung erklärt ist. Fast alle rekursiven Typgleichungen werden in dieser Weise verstanden und deshalb sind induktive Datentypen das wichtigste Konstrukt zur Einführung rekursiv definierter Konzepte.

Anders als bei den bisherigen Typkonstruktoren beschreiben induktive Datentypen zwar eine neue Klasse von Elementen, erzeugen hierfür aber *keine neuen kanonischen* Elemente. Stattdessen wird durch die rekursive Gleichung eine Art Rezept angegeben, wie Elemente des induktiven Typs zu konstruieren sind. So enthält zum Beispiel die oben angegebene Typdefinition

$$\mathsf{rectype}\ \texttt{bintree} = \mathbb{Z}\ +\ \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$$

die Vorschrift, daß die Elemente von `bintree` entweder Elemente von $\mathbb{Z}$ sein müssen, oder aus einem Tripel bestehen, dessen erste Komponente ein Element von $\mathbb{Z}$ ist und dessen andere beiden Komponenten jeweils Elemente von `bintree` sein müssen. Dabei kann im letzteren Fall nur auf bereits erklärte Elemente von `bintree` zurückgegriffen werden.

Um Elemente eines induktiven Datentyps weiterverwenden zu können, müssen wir eine *nichtkanonische Form* bereitstellen, welche besagt, wie der induktiven Aufbau eines solchen Elementes zu analysieren ist. Dies geschieht ähnlich wie im Falle der natürlichen Zahlen, die wir in Abschnitt 3.4.2.1 auf Seite 140 besprochen haben: ein Term $\underline{\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(e)}$[68] beschreibt den Funktionswert einer Funktion f bei Eingabe eines Elementes e von $\mathsf{rectype}\ X = T_X$, wobei für f die rekursive Funktionsgleichung $f(x)$=t gelten soll. Diese Form läßt sich eigentlich auch unabhängig von induktiven Datentypen verwenden (siehe Abschnitt 3.5.2), führt im Zusammenhang mit diesen jedoch zu *immer terminierenden rekursiven Funktionen*, sofern sie mithilfe einer ensprechenden Eliminationsregel generiert wurde. In diesem Fall wird nämlich der induktive Aufbau des Elementes e schrittweise rekursiv abgebaut bis der "Basisfall" erreicht ist und eine feste Antwort berechnet werden kann. Wir wollen dies an einem Beispiel illustrieren

Beispiel 3.5.2

Wenn wir für einen Binärbaum b die Summe der Knoten berechnen wollen, dann reicht es, die in Beispiel 3.5.1 gegebene Struktur zu analysieren. Falls b ein Element von $\mathbb{Z}$ ist, dann ist b selbst die Summe. Andernfalls müssen wir rekursiv die Summe der beiden Teilbäume bestimmen und zum Wert der Wurzel hinzuaddieren. Die Fallanalyse ist durch den Vereinigungstyp bereits implizit vorgegeben. Wir erhalten also folgende formale Beschreibung eines Algorithmus `binsum`(t):

```
let* sum(b-tree)  =
  case b-tree  of inl(leaf) ↦ leaf
                | inr(triple) ↦ let ⟨num,pair⟩ = triple
                                in  let ⟨left,right⟩ = pair
                                    in  num+sum(left)+sum(right)
in sum(t)
```

Durch die nichtkanonische Form $\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(e)$ sind wir nun in der Lage, zur Beschreibung von Funktionen eine unbeschränkte Rekursion verwenden zu dürfen, was die praktische Ausdruckskraft und *Effizienz* von NuPRL Programmen deutlich steigert. Falls keine freien Variablen in dieser Form vorkommen, kann sie als Redex betrachtet werden. Eine Reduktion entspricht in diesem Falle der Auswertung eines einzelnen Rekursionsschrittes: $\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(e)$ reduziert zu $t[\lambda y.\,\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(y),\ e\ /\ f, x]$. Um dies tun zu können, muß allerdings der induktive Aufbau des Elements e vorliegen, wodurch auch die Terminierung der Rekursion gesichert wird. Deshalb ist aus theoretischer Sicht keine Ausdruckskraft gegenüber der primitiven Rekursion gewonnen worden. Eine wirkliche Steigerung ist erst dann möglich, wenn man die Rekursion von der Analyse eines vorgegebenen induktiven Aufbaus entkoppelt und das Risiko nichtterminierender Algorithmen eingeht. Die hierbei entstehenden Probleme und einen Ansatz zu ihrer Lösung werden wir in Abschnitt 3.5.2 besprechen.

[68] Man beachte, daß durch $\mathsf{rectype}\ X = T_X$ und $\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(e)$ nur ein Term erklärt wird, nicht aber der *Name* X oder f vergeben wird. X und f sind – wie die Details in Abbildung 3.26 zeigen – nichts anderes als bindende Variablen

kanonisch (Typen)	(Elemente)	**nichtkanonisch**
$\mathbf{rec}\{\}(X\,.\,T_X)$		$\mathbf{rec\_ind}\{\}(\boxed{e};\, f, x\,.\,t)$
$\mathsf{rectype}\ X = T_X$		$\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(\boxed{e})$

Zusätzliche Einträge in die Operatorentabelle

Redex		**Kontraktum**
$\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(e)$	$\xrightarrow{\beta}$	$t[\lambda y\,.\,\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(y),\ e\ /\ f, x]$

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$\mathsf{rectype}\ X_1 = T_{X1} = \mathsf{rectype}\ X_2 = T_{X2}$	falls	$T_{X1}[X/X_1] = T_{X2}[X/X_2]$ für alle Typen X
Elementsemantik		
$s = t\ \in\ \mathsf{rectype}\ X = T_X$	falls	$\mathsf{rectype}\ X = T_X$ Typ und $s = t\ \in\ T_X[\mathsf{rectype}\ X = T_X\ /\ X]$
$\mathsf{rectype}\ X_1 = T_{X1} = \mathsf{rectype}\ X_2 = T_{X2}\ \in\ \mathsf{U}_j$	falls	$T_{X1}[X/X_1] = T_{X2}[X/X_2]\ \in\ \mathsf{U}_j$ für alle Terme X mit $X\ \in\ \mathsf{U}_j$

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash \mathsf{rectype}\ X_1 = T_{X1}\ \mathsf{=}\ \mathsf{rectype}\ X_2 = T_{X2}\ \in\ \mathsf{U}_j$ [Ax]
by recEq
$\Gamma,\ X\!:\!\mathsf{U}_j \vdash T_{X1}[X/X_1]\ \mathsf{=}\ T_{X2}[X/X_2]\ \in\ \mathsf{U}_j$ [Ax]

$\Gamma \vdash s\ \mathsf{=}\ t\ \in\ \mathsf{rectype}\ X = T_X$ [Ax]
by rec_memEq j
$\Gamma \vdash s\ \mathsf{=}\ t\ \in\ T_X[\mathsf{rectype}\ X = T_X\ /\ X]$ [Ax]
$\Gamma \vdash \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{let}^*\ f_1(x_1) = t_1\ \ \mathsf{in}\ \ f_1(e_1)$
$\quad \mathsf{=}\ \mathsf{let}^*\ f_2(x_2) = t_2\ \ \mathsf{in}\ \ f_2(e_2)\ \in\ T[e_1/z]$ [Ax]
by rec_indEq z T $\mathsf{rectype}\ X = T_X$ j
$\Gamma \vdash e_1\ \mathsf{=}\ e_2\ \in\ \mathsf{rectype}\ X = T_X$ [Ax]
$\Gamma \vdash \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j$ [Ax]
$\Gamma,\ P\!:\!(\mathsf{rectype}\ X = T_X) \to \mathbb{P}_j,$
$\quad f\!:\!(y\!:\!\{x\!:\!\mathsf{rectype}\ X = T_X \mid P(x)\} \to T[y/z]),$
$\quad x\!:\!T_X[\{x\!:\!\mathsf{rectype}\ X = T_X \mid P(x)\}\ /\ X]$
$\quad \vdash t_1[f, x\ /\ f_1, x_1] = t_2[f, x\ /\ f_2, x_2] \in T[x/z]$ [Ax]

$\Gamma \vdash \mathsf{rectype}\ X = T_X$ [**ext** t]
by rec_memI j
$\Gamma \vdash T_X[\mathsf{rectype}\ X = T_X\ /\ X]$ [**ext** t]
$\Gamma \vdash \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j$ [Ax]

$\Gamma,\ z\!:\!\mathsf{rectype}\ X = T_X,\ \Delta \vdash C$
$\quad$ [**ext** $\mathsf{let}^*\ f(x) = t[\lambda y\,.\,\Lambda\ /\ P]\ \ \mathsf{in}\ \ f(z)$]
by recE i j
$\Gamma,\ z\!:\!\mathsf{rectype}\ X = T_X,\ \Delta$
$\quad \vdash \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j$ [Ax]
$\Gamma,\ z\!:\!\mathsf{rectype}\ X = T_X,\ \Delta$
$\quad P\!:\!(\mathsf{rectype}\ X = T_X) \to \mathbb{P}_j,$
$\quad f\!:\!(y\!:\!\{x\!:\!\mathsf{rectype}\ X = T_X \mid P(x)\} \to C[y/z]),$
$\quad x\!:\!T_X[\{x\!:\!\mathsf{rectype}\ X = T_X \mid P(x)\}\ /\ X]$
$\quad\quad \vdash C[x/z]$ [**ext** t]

$\Gamma,\ z\!:\!\mathsf{rectype}\ X = T_X,\ \Delta \vdash C$ [**ext** $t[z/x]$]
by recE_unroll i
$\Gamma,\ z\!:\!\mathsf{rectype}\ X = T_X,\ \Delta,$
$\quad x\!:\!T_X[\mathsf{rectype}\ X = T_X\ /\ X],$
$\quad v\!:\ z\ \mathsf{=}\ x \in T_X[\mathsf{rectype}\ X = T_X\ /\ X]$
$\quad\quad \vdash C[x/z]$ [**ext** t]

Inferenzregeln

Abbildung 3.26: Syntax, Semantik und Inferenzregeln induktiver Typen

Induktive Datentypen können im Prinzip simuliert werden, da wir einen induktiven Datentyp – die natürlichen Zahlen – bereits kennen. So könnte zum Beispiel eine Simulation des Typs der Binärbäume aus Beispiel 3.5.1 zunächst induktiv Binärbäume der maximalen Tiefe i definieren durch

$$\texttt{Bintree}(i) \equiv \mathsf{ind}(i; \_,\_.\_; \mathbb{Z}; \texttt{j},\texttt{bin}_j.\ \texttt{bin}_j + \mathbb{Z} \times \texttt{bin}_j \times \texttt{bin}_j)$$

und dann festlegen

$$\texttt{Bintree} \equiv \texttt{i:}\mathbb{N} \times \texttt{Bintree(i)}.$$

Eine derartige Simulation wäre allerdings verhältnismäßig umständlich und würde die natürliche Form der rekursiven Definition völlig entstellen. Nichtsdestotrotz zeigt die Simulation, daß die Hinzunahme der induktiven Datentypen keine Erweiterung der Ausdruckskraft sondern nur eine naturgetreuere Formalisierung für eine bestimmte Klasse von Datentypen liefert.[69]

Abbildung 3.26 faßt die Syntax, Semantik und Inferenzregeln induktiver Datentypen zusammen. Man beachte, daß die Semantik des nichtkanonischen Terms durch "Aufrollen" (Englisch "*unroll*") der Rekursion erklärt wird, wobei jedoch nur ein einziger Rekursionsschritt durchgeführt wird. Ist der entstehende Term nicht in kanonischer Form, so muß weiter reduziert werden, um die Bedeutung des Ausdrucks zu erhalten. Hierdurch wird sichergestellt, daß im Kalkül selbst keine nichtterminierenden Bestandteile auftauchen obwohl der implizit in $\mathsf{let}^* f(x) = t\ \mathsf{in}\ f(e)$ enthaltene Algorithmus (Auflösen der Rekursion bis zu einem Terminierungspunkt) durchaus nicht zu einem Ende kommen muß.

Die meisten Inferenzregeln stützen sich ebenfalls auf das Aufrollen einer Rekursion ab. Da es keine kanonischen Elemente gibt, muß zum Nachweis der Typzugehörigkeit die rekursive Typdefinition schrittweise aufgefaltet und analysiert werden. Ebenso kann man induktive Typen durch einfaches Aufrollen eliminieren (`recE_unroll`). Im Normalfall (`recE`) ist die Elimination induktiver Typen allerdings komplexer, da hierdurch ein nichtkanonischer Term der Form $\mathsf{let}^* f(x) = t\ \mathsf{in}\ f(e)$ erzeugt werden soll. Um die oben beschriebene Bedeutung (Funktionswert einer rekursiv durch t definierte Funktion f mit Argument x bei Eingabe eines Elementes z) zu erzielen, müssen wir f als eine Funktion auf einer beliebigen Teilmenge von $\mathsf{rectype}\ X = T_X$ und x als ein Element von T_X – wobei für X diese Teilmenge eingesetzt wird – voraussetzen und zeigen, wie wir hieraus den Term t konstruieren. Hierdurch wird sichergestellt, daß der entstehende rekursive Algorithmus wohldefiniert ist und terminiert, wenn man mit der leeren Menge als Ausgangspunkt anfängt.[70]

Beispiel 3.5.3

Als Anwendungsbeispiel wollen wir die Konstruktion eines Algorithmus skizzieren, welcher bei Eingabe eines Binärbaumes und einer ganzen Zahl entscheidet, ob diese Zahl im Baum erscheint oder nicht. Da dieser Algorithmus genauso grundlegend ist, wie die Eigenschaft, die ihn spezifiziert, geben wir als Beweisziel nur eine Typisierung des Algorithmus an und konstruieren diesen dann implizit durch unsere Entscheidungen im Laufe des Beweises. Aus Gründen der Übersichtlichkeit werden wir uns auf eine Skizze der Kernbestandteile des Beweises beschränken. Als definitorische Abkürzung verwenden wir

$$\texttt{Bintree} \equiv \mathsf{rectype}\ \texttt{bintree} = \mathbb{Z} + \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$$

[69] Umgekehrt macht die Existenz der induktiven Datentypen natürlich auch die explizite Definition konkreter induktiver Typen wie natürlicher Zahlen bzw. Listen hinfällig. Man könnte nämlich simulieren:

$$\mathbb{N} \equiv \mathsf{rectype}\ \texttt{N} = \texttt{Unit} + \texttt{N}$$

wobei `Unit` ein vorgegebener einelementiger Datentyp ist und eine Zahl n durch n Rekursionen dargestellt wird. Ebenso lassen sich Listen über dem Typ T beschreiben durch

$$T\ \mathsf{list} \equiv \mathsf{rectype}\ \texttt{T\_list} = T + \texttt{T\_list}.$$

Da jedoch auch diese Simulation unnatürlich wäre, ist eine explizite Formalisierung dieser Datentypen in jedem Fall vorzuziehen.

[70] Diese Vorgehensweise spiegelt die '*Fixpunktinduktion*' wieder. Der induktive Typ $\mathsf{rectype}\ X = T_X$ kann semantisch als Grenzwert der Folge $\emptyset$, $T_X(\emptyset)$, $T_X(T_X(\emptyset))$, ... angesehen werden und jedes Element z von $\mathsf{rectype}\ X = T_X$ gehört zu einer dieser Stufen. Eine rekursive Analyse von z wird also schrittweise diese Stufen abbauen und bei $\emptyset$ terminieren. Da die Stufe von z aber nicht bekannt ist, muß der Analyseterm t auf beliebigen Teilmengen von $\mathsf{rectype}\ X = T_X$ operieren können und sich im Endeffekt auf $\emptyset$ abstützen.

Die genauen Regeln sind das Ergebnis mühevoller Feinarbeit, bei der es darum ging, mögliche Trugschlüsse zu vermeiden. Sie sind daher normalerweise nicht sofort intuitiv klar sondern müssen gründlich durchdacht werden.

```
⊢ Bintree → ℤ → 𝔹
by lambdaI 1                                  ⌊ext λB.λz.let* f(x) = case x of inl(i) ↦ ... | inr(tree) ↦ ... in f(B)⌋
|\
| B:Bintree ⊢ ℤ → 𝔹
| by lambdaI 1                                 ⌊ext λz.let* f(x) = case x of inl(i) ↦ ... | inr(tree) ↦ ... in f(B)⌋
| |\
| | B:Bintree, z:ℤ ⊢ 𝔹
| | by recE 1 1                                ⌊ext let* f(x) = case x of inl(i) ↦ ... | inr(tree) ↦ ... in f(B)⌋
| | |\
| | | B:Bintree, z:ℤ ⊢ Bintree ∈ U1
| | |   siehe unten
| | |\
| | |   B:Bintree, z:ℤ, P:Bintree→ℙ1, f:{x:Bintree | P(x)} → 𝔹,
| | |     x:ℤ + ℤ × {x:Bintree | P(x)} × {x:Bintree | P(x)}
| | |   ⊢ 𝔹
| | |   by unionE 5                           ⌊ext case x of inl(i) ↦ if z=i then T else F | inr(tree) ↦ let .... ⌋
| | |   |\
| | |   | .... i:ℤ ⊢ 𝔹
| | |   | by intro if z=i then T else F                                  ⌊ext if z=i then T else F⌋
| | |   |    ⋮
| | |   |\
| | |     .... tree:ℤ × {x:Bintree | P(x)} × {x:Bintree | P(x)} ⊢ 𝔹
| | |     by productE 6 THEN productE 8          ⌊ext let ⟨i,B1,B2⟩ = tree in if z=i then T else ...⌋
| | |     |\
| | |       .... i:ℤ, pair:{x:Bintree | P(x)} × {x:Bintree | P(x)},
| | |         B1:{x:Bintree | P(x)}, B2:{x:Bintree | P(x)}
| | |       ⊢ 𝔹
| | |       by intro if z=i then T else if f(B1) then T else f(B2)      ⌊ext if z=i then T else ...⌋
| | |          ⋮
| | |\
| | |   ⊢ ℤ ∈ U1
| | |   by intEq
| |\
| | ⊢ Bintree ∈ U1
| | by recEq
| |\
| |   bintree:U1 ⊢ ℤ + ℤ × bintree × bintree ∈ U1
| |   by ..... unionEq, productEq, intEq, hypEq
```

Aus dem Beweis können wir den folgenden rekursiven Algorithmus extrahieren.

```
λB.λz. let* f(x) =
         case x of inl(i) ↦ if z=i then T else F
                 | inr(tree) ↦ let ⟨i,B1,B2⟩ = tree in if z=i then T else if f(B1) then T else f(B2)
       in f(B)
```

Dieser Algorithmus analysiert den rekursiven Aufbau eines Binärbaumes, und vergleicht die gesuchte Zahl mit der Wurzel des Baumes (bzw. dem ganzen Baum im Basisfall). Sind diese Zahlen identisch, so ist die Suche beendet. Ansonsten wird zunächst der linke und danach der rechte Teilbaum durchsucht.

Induktive Datentypen sind ein mächtiges Hilfsmittel zum Schließen über wohlfundierte, rekursive definierte Konstrukte. Sie sind allerdings auch mit Vorsicht einzusetzen, da – im Gegensatz zu den bisher eingeführten Typkonstrukten – nicht jeder formulierbare induktive Datentyp auch sinnvoll ist.

Beispiel 3.5.4

Wir betrachten den Term $T \equiv$ rectype $\mathtt{X} = \mathtt{X} \to \mathbb{Z}$. Die hierin enthaltene Rekursionsgleichung legt fest, daß die Elemente von T Funktionen von T in die Menge der ganzen Zahlen sein müssen. Daß dies zu widersprüchlichen Situationen führt, zeigt das folgende Argument.

> Es sei $t \in T$. Dann ist t auch ein Element von $T \to \mathbb{Z}$ und somit ist $t\,t$ ein wohldefiniertes Element von $\mathbb{Z}$. Abstrahieren wir nun über t, so erhalten wir $\lambda t.\, t\,t \in T \to \mathbb{Z}$ bzw. $\lambda t.\, t\,t \in T$.

Wenn wir also rectype $\mathtt{X} = \mathtt{X} \to \mathbb{Z}$ als korrekten induktiven Datentyp innerhalb der Typentheorie zulassen würden, dann würde ein wohlbekannter nichtterminierender Term typisierbar werden und dies – im

Gegensatz zu unserem Beispiel aus Abbildung 3.15 (Seite 134) – sogar ohne jede Voraussetzung. Wir wären sogar in der Lage, diesen Term aus dem folgenden Beweis zu extrahieren.

```
⊢ T    ⌊ext λt. t t⌋
by rec_memI 1
|\
| ⊢ T→ℤ    ⌊ext λt. t t⌋
|  by lambdaI 1
|  |\
|  | t:T ⊢ ℤ   ⌊ext t t⌋
|  | by recE_unroll 1
|  |  \
|  |   t:T, y:T→ℤ, v: t=y∈T→ℤ ⊢ ℤ   ⌊ext y y⌋
|  |   by functionE 2 y
|  |   |\
|  |   | t:T, y:T→ℤ, v: t=y∈T→ℤ ⊢ y∈T
|  |   |  Substitution etc.
|  |    \
|  |     t:T, y:T→ℤ, v: t=y∈T→ℤ, z:ℤ, v': z = y y ∈ ℤ ⊢ ℤ   ⌊ext z⌋
|  |     by hyp 4
|   \
|    ⊢ T ∈ U1
|      siehe unten
 \
  ⊢ T ∈ U1
  by recEq
   \
    X:U1 ⊢ X→ℤ ∈ U1
    by functionEq
     |\
     | X:U1 ⊢ X ∈ U1
     | by hypEq 1
      \
       X:U1 ⊢ ℤ ∈ U1
       by intEq
```

Dieses Beispiel zeigt, daß wir Terme der Form rectype X = X→T nicht zulassen dürfen, da wir auf die schwache Normalisierbarkeit typisierbarer Terme nicht verzichten wollen. Diese Einschränkung ist durchaus sehr natürlich, da auch intuitiv ein Typ nicht sein eigener Funktionenraum sein darf. Was aber genau ist die Ursache des Problems?

Im Beispiel haben wir gesehen, daß die Kombination der Einführungsregel für λ-Terme und der `unroll`-Regel für induktive Datentypen die Einführung des Terms $\lambda t.\, t\, t$ ermöglicht. Dies ist aber nur dann der Fall, wenn der induktive Datentyp T ein Funktionenraum ist, der sich selbst im Definitionsbereich enthält. Ein Datentyp der Form rectype X = T→X ist dagegen durchaus unproblematisch.

Wie können wir nun *syntaktische* Einschränkungen an induktive Datentypen geben, welche die problematischen Fälle schon im Vorfeld ausschließen, ohne dabei übermäßig restriktiv zu sein? Eine genaue syntaktische Charakterisierung der zulässigen Möglichkeiten konnte bisher noch nicht gefunden werden. Die bisher beste Restriktion ist die Forderung, daß in einem induktiven Datentyp rectype $X = T_X$ die Typvariable X in T_X nur *positiv* vorkommen darf, was in etwa[71] dasselbe ist wie die Bedingung, daß X nicht auf der linken Seite eines Funktionenraumkonstruktors in T_X erscheinen darf. Diese Bedingung, die sich syntaktisch relativ leicht überprüfen läßt, stellt sicher, daß nur induktive Datentypen mit einer wohldefinierten Semantik in formalen Definitionen und Beweisen verwendet werden können.

Der bisher vorgestellte induktive Datentyp reicht aus, um die meisten in der Praxis vorkommenden rekursiven Konzepte zu beschreiben. In manchen Fällen ist es jedoch sinnvoll, die rekursive Definition zu *parametrisieren*.

[71] Eine ausführliche Definition dieses Begriffs findet man in [Constable & Mendler, 1985, Mendler, 1987a].

Beispiel 3.5.5

Um den Datentyp der logischen Propositionen erster Stufe innerhalb der Typentheorie präzise zu beschreiben, müßte man eine entsprechende Einschränkung des Universums U_1 charakterisieren, die nur solche Terme zuläßt, deren äußere Gestalt einer logischen Formel im Sinne von Definition 2.2.4 entspricht:

$$\mathbb{P} \equiv \{\mathtt{F}{:}\mathsf{U}_1 \mid \mathtt{is\_formula(F)}\}$$

Dabei soll `is_formula(F)` das Prädikat (den Datentyp) repräsentieren, welches beschreibt, daß `F` eine Formel ist. Dieses Prädikat hat – entsprechend der Definition 2.2.4 – eine rekursive Charakterisierung:

$$\begin{array}{l}
\mathtt{is_formula(F)} \\
\quad \Leftrightarrow \quad \mathtt{atomic(F)} \\
\qquad \vee \ \mathtt{F} = \Lambda \in \mathsf{U}_1 \\
\qquad \vee \ \exists \mathtt{A}{:}\mathsf{U}_1 . \exists \mathtt{B}{:}\mathsf{U}_1 . \ \mathtt{is_formula(A)} \ \wedge \ \mathtt{is_formula(A)} \\
\qquad\quad \wedge \ \mathtt{F} = \neg\mathtt{A} \in \mathsf{U}_1 \\
\qquad\quad \vee \ \mathtt{F} = \mathtt{A} \wedge \mathtt{B} \in \mathsf{U}_1 \\
\qquad\quad \vee \ \mathtt{F} = \mathtt{A} \vee \mathtt{B} \in \mathsf{U}_1 \\
\qquad\quad \vee \ \mathtt{F} = \mathtt{A} \Rightarrow \mathtt{B} \in \mathsf{U}_1 \\
\qquad \vee \ \exists \mathtt{T}{:}\mathsf{U}_1 . \exists \mathtt{A}{:}\mathtt{T} \to \mathsf{U}_1 . \ \forall \mathtt{x}{:}\mathtt{T}.\mathtt{is_formula(\ A(x)\ )} \\
\qquad\quad \wedge \ \mathtt{F} = \forall \mathtt{x}{:}\mathtt{T}.\mathtt{A(x)} \in \mathsf{U}_1 \\
\qquad\quad \vee \ \mathtt{F} = \exists \mathtt{x}{:}\mathtt{T}.\mathtt{A(x)} \in \mathsf{U}_1
\end{array}$$

Will man dies nun in einen induktiven Datentyp umsetzen, so stellt man fest, daß die Rekursion nicht nur von `is_formula` abhängt, sondern auch das Argument des Prädikats sich ständig ändert. Mit dem bisherigen – einfachen – Konzept der induktiven Datentypen läßt sich dies nicht mehr auf natürliche Art ausdrücken.

Eine Simulation parametrisierter induktiver Datentypen mithilfe der einfachen Version und des Produkttyps ist prinzipiell möglich, führt aber zu größeren Komplikationen. Es gibt daher Überlegungen, parametrisierte induktive Datentypen als Grundform in die Typentheorie mit aufzunehmen.[72]

Neben der Parametrisierung gibt es noch eine weitere Erweiterungsmöglichkeit für induktive Datentypen. So hatten wir in der Definition der Terme der Typentheorie (siehe Seite 104) das Konzept der Terme auf gebundene Terme und dieses wieder auf das der Terme abgestützt. Eine solche simultane (Englisch *mutual*=gegenseitig) rekursive Definition zweier Konzepte taucht auch in modernen Programmiersprachen relativ häufig auf: zwei Prozeduren können sich wechselseitig immer wieder aufrufen, bis ein Ergebnis geliefert wird. Zwar gibt es auch hierfür eine Simulation (siehe z.B. [Mendler, 1987a, Seite 17]), aber die Natürlichkeit der simultanen Rekursion macht es sinnvoll, diese ebenfalls explizit in die Typentheorie zu integrieren.

Die allgemeinste Form des induktiven Datentyps würde also n durch simultane Induktion definierte Datentypen X_i enthalten, die durch Paramter x_i parametrisiert sind. Zusätzlich müßte angegeben werden, welcher dieser Datentypen X_i nun gefragt ist und mit welchem Wert a_i der Parameter x_i initialisiert werden soll. Als Term wird hierfür vorgeschlagen:

$$\mathbf{mrec}\{\}(X_1, x_1 . T_{X_1}; \ldots; X_n, x_n . T_{X_n};\ X_i; a_i)^{73}$$

Eine vollständige Formalisierung derartiger induktiver Datentypen sowie ihre Semantik und die zugehörigen Regeln wird in [Constable & Mendler, 1985, Mendler, 1987a] ausführlich behandelt.

[72] In früheren Versionen des NuPRL Systems waren diese parametrisierten Versionen auch enthalten. Da sie aber erheblich aufwendiger sind und so gut wie keine praktische Anwendung fanden, wurden sie zugunsten der einfacheren Handhabung durch die hier vorgestellten "einfachen rekursiven Datentypen" ersetzt.

[73] Da dieser Typ bisher kaum Anwendung fand, gibt es eine bisher keine sinnvolle Displayform. In Anlehnung an die Programmiersprache ML könnte man vielleicht schreiben: $\mathsf{rectype}\ X_1(x_1) = T_{X_1}\ \mathsf{and}\ \ldots\ \mathsf{and}\ X_n(x_n) = T_{X_n}\ \mathsf{select}\ X_i(a_i)$

3.5.2 (Partiell) Rekursive Funktionen

Im vorhergehenden Abschnitt haben wir gesehen, daß die nichtkanonischen Terme, die wir im Zusammenhang mit induktiven Datentypen eingeführt haben, eine Möglichkeit bieten, allgemeine Rekursion in die Typentheorie mit aufzunehmen. Der Term $\mathsf{let}^* f(x) = t \ \mathsf{in} \ f(e)$ beschreibt eine rekursive Funktion in Abhängigkeit von einem Element eines induktiven Datentyps. Hierdurch gewinnen wir die Eleganz und Effizienz der allgemeinen Rekursion und behalten dennoch die Eigenschaft, daß alle typisierbaren Algorithmen terminieren. Dennoch bleibt etwas Unnatürliches in diesem Ansatz, da zugunsten der Wohlfundiertheit die in $\mathsf{let}^* f(x) = t \ \mathsf{in} \ f(e)$ definierte Funktion f an einen vorgegebenen rekursiven Datentyp gebunden ist, welcher ihren Definitionsbereich beschreibt. In vielen Fällen wird jedoch eine rekursive Funktion *nicht* innerhalb eines Beweises als Extraktterm eines induktiven Datentyps konstruiert. Stattdessen ist oft nur der Algorithmus gegeben, ohne daß der konkrete Definitionsbereich bekannt ist. Wir wollen hierfür einige Beispiele geben.

Beispiel 3.5.6

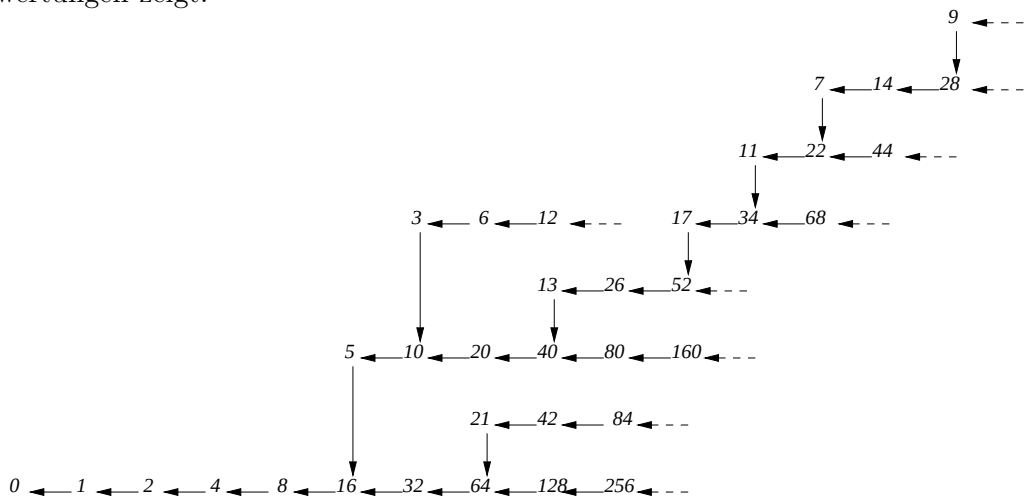
1. In der Mathematik ist die sogenannte 3x+1-Funktion ein beliebtes Beispiel für eine leicht zu definierende rekursive Funktion, deren Definitionsbereich man nicht auf natürliche Art beschreiben kann. Sie ist definiert durch

$$f(x) = \begin{cases} 0 & \text{falls } x = 1, \\ f(x/2) & \text{falls } x \text{ gerade ist,} \\ f(3x+1) & \text{sonst} \end{cases}$$

   Löst man die Koppelung der rekursiven Funktionsdefinition von der Vorgabe eines induktiven Definitionsbereichs, so läßt sich diese Funktion formalisieren durch

$$\lambda \mathtt{x}.\ \mathsf{let}^*\ \mathtt{f(y)} = \mathsf{if}\ \mathtt{y{=}1}\ \mathsf{then}\ \mathtt{0}\ \mathsf{else}\ \mathsf{if}\ \mathtt{y}\ \mathsf{rem}\ \mathtt{2 = 0}\ \mathsf{then}\ \mathtt{f(x \div 2)}\ \mathsf{else}\ \mathtt{f(3{*}x{+}1)}\ \ \mathsf{in}\ \mathtt{f(x)}$$

   Ihr Definitionsbereich ist allerdings relativ kompliziert strukturiert, wie die folgende Skizze des Verlaufes einiger Auswertungen zeigt.



   Zwar ist es prinzipiell möglich, diesen Definitionsbereich mithilfe eines parametrisierten induktiven Datentyps zu beschreiben, welcher genau der Funktionsdefinition entspricht, aber dies ist nicht nur relativ kompliziert, sondern auch unnatürlich. Hierdurch wir nämlich nicht ausgedrückt, daß es sich um eine – möglicherweise partielle – Funktion auf natürlichen Zahlen handelt, sondern der Bereich der natürlichen Zahlen muß entsprechend obiger Abbildung in seltsamer Weise neu angeordnet werden.

2. Unbeschränkte Suche nach der Nullstelle einer beliebigen Funktion auf den natürlichen Zahlen (vergleiche Beispiel 3.4.7 auf Seite 153) läßt sich ebenfalls leicht als rekursive Funktion beschreiben.

$$\lambda f.\ \mathsf{let}^*\ \mathtt{min}_f\mathtt{(y)} = \mathsf{if}\ f\mathtt{(y){=}0}\ \mathsf{then}\ \mathtt{y}\ \mathsf{else}\ \mathtt{min}_f\ \mathtt{(y{+}1)}\ \mathsf{in}\ \mathtt{min}_f\mathtt{(0)}$$

   Diese Funktion ist wohldefiniert und terminiert offensichtlich auch auf allen Elementen der Funktionenmenge $\{\mathtt{f}:\mathbb{N}\to\mathbb{Z} \mid \exists \mathtt{y}:\mathbb{N}.\ \mathtt{f\ (y) = 0}\}$. Nichtsdestotrotz haben wir keinerlei Informationen über die rekursive Struktur dieses Definitionsbereiches.

3. In Beispiel 3.4.9 auf Seite 154 haben wir einen linearen Algorithmus zur Berechnung der Integerquadratwurzel einer natürlichen Zahl `x` hergeleitet und darauf hingewiesen, daß dies der effizienteste Algorithmus ist, den man mit dem Mittel der Induktion bzw. der primitiven Rekursion erzeugen kann. Natürlich aber gibt es erheblich effizientere Algorithmen, wenn man anstelle einer Rekursion auf der Eingabe `x` eine Rekursion auf dem Ergebnis verwenden kann. In diesem Falle ist es nämlich möglich, nach dem kleinsten Wert `y` zu suchen, für den $\texttt{(y+1)}^2\texttt{>x}$ gilt. Diese Vorgehensweise führt zum Beispiel zu dem folgenden rekursiven Algorithmus

   $$\lambda\texttt{x. let}^*\ \texttt{sq-search(y)} = \texttt{if x<(y+1)}^2\ \texttt{then y else sq-search (y+1) in sq-search(0)}$$

   Auch dieser Algorithmus ist sehr leicht zu verstehen und terminiert auf allen natürlichen Zahlen. Eine Koppelung an eine rekursive Datenstruktur müßte die natürlichen Zahlen jedoch in einer ganz anderen, dem Problem angepaßten Weise strukturieren.

Diese Beispiele zeigen, daß eine Koppelung der allgemeinen Rekursion an einen induktiven Datentyp eine massive Einschränkung der praktischen Anwendbarkeit rekursiver Funktionsdefinitionen mit sich bringt. Man muß nämlich immer erst den Definitionsbereich so umstrukturieren, daß der vorgesehene Algorithmus genau auf dieser Struktur arbeiten kann. Erst danach kann man dann den Algorithmus entwickeln.

In der Praxis geht man jedoch andersherum vor. Im Vordergrund steht der Algorithmus, der auf einem fest vorgegeben Definitionsbereich oder einer Teilmenge davon operieren soll. Die induktive Struktur dieses Definitionsbereiches ergibt sich dann implizit aus der Abarbeitungsstruktur des Algorithmus, wird aber nicht als eigentlicher Definitionsbereich angesehen. So ist zum Beispiel der Definitionsbereich der Integerquadratwurzelfunktion die Menge der natürlichen Zahlen und nicht etwa ein komplizierter induktiver Typ, in dem natürliche Zahlen blockweise strukturiert sind. Ähnliches gilt für das Nullstellensuchprogramm, dessen Definitionsbereich eine Teilmenge der ganzzahligen Funktionen ist.

Es ist also wünschenswert, die Typentheorie um ein Konzept zu ergänzen, welches erlaubt, *alle* rekursiven Algorithmen zu betrachten, ohne daß dafür im Voraus die genaue Struktur des Definitionsbereiches bekannt sein muß. Dies würde erlauben, innerhalb der Typentheorie sehr effiziente Algorithmen zu programmieren und zu analysieren und weit über die Ausdruckskraft der primitiv-rekursiven Funktionen hinauszugehen. Erst die Entkoppelung der rekursiven Funktionen von den induktiven Datentypen macht es möglich, formal über "reale" Programme zu argumentieren. Damit wäre die Typentheorie nicht nur theoretisch dazu geeignet, das Schließen über Programmierung zu formalisieren, sondern auch ein praktisch adäquater Formalismus in dem Sinne, daß sie tatsächlich auch als reale Programmiersprache eingesetzt werden kann. Entsprechend den Erkenntnisssen der Theorie der Berechenbarkeit müßten wir hierfür allerdings auch Algorithmen akzeptieren, die von ihrer Natur her auch partiell sein könnten

Wie können wir nun allgemeine Rekursion in die Typentheorie mit aufnehmen, wenn doch gerade die unbeschränkte Rekursion die Ursache aller Probleme ist, die wir in Kapitel 2.3.7 feststellen mußten? Wie können wir also die Ausdruckskraft der Typentheorie auf elegante Art um ein Konzept partiell rekursiver Funktionen erweitern, ohne dabei das Inferenzsystem mit nichtterminierenden Bestandteilen zu belasten?

Die naheliegendste Idee ist, partiell rekursive Funktionen von einem Typ S in einen Typ T durch totale Funktionen zu repräsentieren, die auf einer Teilmenge von S operieren. Man könnte also den Datentyp $\underline{S \not\to T}$ der partiellen Funktionen simulieren durch

$$S \not\to T \equiv \texttt{DOM:}S \to \mathbb{P}_1 \times \{\texttt{x:}S \mid \texttt{DOM(x)}\} \to T.$$

Diese Darstellung ist sehr einfach und leicht zu handhaben, hat aber den Nachteil, daß man zu jedem Programm eine Beschreibung des Definitionsbereiches gleich mitliefern muß. $S \not\to T$ wäre also nicht ein Datentyp von partiellen Funktionen im eigentlichen Sinne.

Ein wesentlich natürlicherer und ebenso einfacher Ansatz, partielle Funktionen in die Typentheorie zu integrieren ist es, Rekursion zunächst einmal als ein *unabhängiges* Berechnungskonzept zu betrachten und die Definitionsbereiche rekursiver Funktionen aus ihrem Algorithmus *herzuleiten*. Dabei ist natürlich klar, daß es nicht immer möglich ist, den genauen Bereich zu bestimmen, auf dem eine Funktion terminiert. Der

hergeleitete Definitionsbereich muß sich daher daran orientieren, was über den gegebenen Algorithmus mit Sicherheit bewiesen werden kann.[74] Dieser Weg macht das Schließen über partielle Funktionen zwar etwas aufwendiger, als wenn der Definitionsbereich vorgegeben ist, entspricht aber dem intuitiven Verständnis partieller Funktionen erheblich besser.

Formal bedeutet dies, daß jeder Funktion $f \in S \not\to T$ ein *Domain-Prädikat* $\underline{\mathsf{dom}(f)} \in S \to \mathbb{P}_1$ zugeordnet wird, welches für Elemente von S angibt, ob sie zum Definitionsbereich von S gehören oder nicht. Die Konsequenz dieser Vorgehensweise ist, daß eine partielle Funktion $f \in S \not\to T$ in der Typentheorie tatsächlich wie eine *totale* Funktion behandelt werden kann, nämlich als ein Element von $\{\mathtt{x}:S \mid \mathsf{dom}(f)(\mathtt{x})\} \to T$.

In Anlehnung an die Notation der Programmiersprache ML bezeichnen wir die kanonischen Elemente des Typs $S \not\to T$ mit $\underline{\mathsf{letrec}\ f(x) = t}$.[75] Dabei ist f allerdings kein Name, der hierdurch neu eingeführt wird, sondern wie x nur eine Variable, die in t gebunden wird. Die nichtkanonische Funktionsapplikation bezeichnen wir wie bei den 'normalen' Funktionen mit $\underline{f\,(t)}$, wobei jedoch zu bedenken ist, daß hinter dieser Darstellungsform ein völlig anderer interner Term steht. Die Reduktion einer Applikation rekursiv definierter Funktionen $(\mathsf{letrec}\ f(x) = t)\ (u)$ wird definiert durch Ausführung eines Rekursionsschrittes und ergibt $t[\mathsf{letrec}\ f(x) = t,\ u\ /\ f, x]$. Der Term $\mathsf{letrec}\ f(x) = t$ zeigt somit bei der Reduktion dasselbe Verhalten wie die Funktion $\lambda \mathtt{y}.\,\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(\mathtt{y})$. Da aber die Koppelung an einen fest vorstrukturierten Definitionsbereich entfällt, können wir $\mathsf{letrec}\ f(x) = t$ als eine Erweiterung der rekursiven Induktion betrachten.

Prinzipiell könnten wir bereits alleine auf der Basis der Reduktionsregel definieren, wann eine rekursive Funktion auf einer Eingabe $s \in S$ terminiert und einen Wert $t \in T$ liefert. In Einzelfällen ist es durchaus möglich, durch eine Reihe von Reduktionsschritten zu beweisen, daß $(\mathsf{letrec}\ f(x) = t)\ (u)\ =\ t' \in T$ gilt. So können wir zum Beispiel problemlos beweisen, daß

$$[\mathsf{letrec}\ \mathtt{f(y)} = \mathsf{if}\ \mathtt{y{=}1}\ \mathsf{then}\ \mathtt{0}\ \mathsf{else\ if}\ \mathtt{y}\ \mathsf{rem}\ \mathtt{2}\ \mathtt{=0}\ \mathsf{then}\ \mathtt{f(x\div 2)}\ \mathsf{else}\ \mathtt{f(3{*}x{+}1)}]\ (1)\ =\ 0\ \in\ \mathbb{Z}$$

gilt. Ähnlich kann man dies für viele andere Eingaben rekursiv definierter Funktionen tun. Das Problem ist jedoch, daß diese Beweise für jede Eingabe einzeln geführt werden müssen. Ein induktives Schließen über den *gesamten* Definitionsbereich $\{\mathtt{y}:S \mid (\mathsf{letrec}\ f(x) = t)\ (u)\ \in\ T\}$, also über die exakte Menge aller Eingaben, auf denen die Funktion terminiert, ist dagegen im Allgemeinfall nicht möglich, obwohl diese eine induktive Struktur besitzt.

Genau aus diesem Grunde wurde das oben erwähnte Domain-Prädikat als weiterer nichtkanonischer Term partiell-rekursiver Funktionen eingeführt. $\mathsf{dom}(f)$ beschreibt einerseits eine Menge von Eingaben, auf denen f garantiert terminiert und bietet andererseits eine Möglichkeit, auf die induktive Struktur des Definitionsbereiches von f zuzugreifen. Dies geschieht dadurch, daß für kanonische Elemente von $S \not\to T$ das Redex $\mathsf{dom}(\mathsf{letrec}\ f(x) = t)$ zu einem induktiven Datentyp-Prädikat reduziert. Wegen der komplexen Struktur der Definitionsbereiche partiell-rekursiver Funktionen kann diese Reduktion jedoch nicht durch ein einfaches Termschema beschrieben werden, sondern muß durch einen Algorithmus berechnet werden. Das Kontraktum hat daher die Gestalt $\lambda x.\,\mathsf{rectype}\ F = \mathcal{E}[[t]]$, wobei $\mathcal{E}$ kein Ausdruck der Typentheorie ist sondern einen Meta-Algorithmus zur Transformation von NuPRL-Termen beschreibt, der bei der Ausführung der Reduktion aufgerufen wird.[76]

---

[74] Dieser Ansatz entstammt einer Idee, die Herbrand [van Heijenoort, 1967] bei einer Untersuchung von Algorithmen im Zusammenhang mit konstruktiver Mathematik und Logik gewonnen hat. Man verzichtet auf den genauen Definitionsbereich, den man wegen des Halteproblems nicht automatisch bestimmen kann, und schränkt sich ein auf diejenigen Elemente ein, bei denen eine Terminierung aus der syntaktischen Struktur des Algorithmus gefolgert werden kann. Zugunsten der Beweisbarkeit werden also unter Umständen einige Eingaben, auf denen der Algorithmus terminiert, als unzulässig erklärt. Dies öffnet einen Weg, partielle Funktionen in ein immer terminierendes Beweiskonzept zu integrieren.

[75] Die interne Bezeichnung $\mathsf{fix}\{\}(f, x.\,t)$ ist an die LCF Tradition [Gordon *et.al.*, 1979] angelehnt und soll daran erinnern, daß die Semantik rekursiver Funktionen durch den kleinsten Fixpunkt der Rekursionsgleichung erklärt ist.

[76] Dieser Algorithmus muß natürlich immer terminieren, da wir verlangen, daß einzelne Reduktionsschritte in der Typentheorie immer zu einem Ergebnis führen. Zudem muß der Definitionsbereich für kanonische Elemente immer wohldefiniert sein, um ein formales Schließen zu ermöglichen. Aufgrund dieser Forderung fällt die Beschreibung des Definitionsbereichs etwas grober aus, als eine optimale Charakterisierung. Eine ausführlichere Beschreibung von $\mathcal{E}$ findet man in [Constable & Mendler, 1985] und [Constable *et.al.*, 1986, Seite 249].

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
$\mathsf{pfun}\{\}(S;T)$	$\mathsf{fix}\{\}(f,x.t)$	$\mathsf{dom}\{\}(\boxed{f})$, $\mathsf{apply\_p}\{\}(\boxed{f};t)$
$S \not\to T$	$\mathsf{letrec}\ f(x)=t$	$\mathsf{dom}(\boxed{f})$, $\boxed{f}(t)$

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
$(\mathsf{letrec}\ f(x)=t)\ (u)$	$\xrightarrow{\beta}$	$t[\mathsf{letrec}\ f(x)=t,\ u\ /\ f,x]$
$\mathsf{dom}(\mathsf{letrec}\ f(x)=t)$	$\xrightarrow{\beta}$	$\lambda x.\,\mathsf{rectype}\ F=\mathcal{E}[[t]]$

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$S_1 \not\to T_1 = S_2 \not\to T_2$	falls	$S_1 = S_2$ und $T_1 = T_2$
Elementsemantik		
$\mathsf{letrec}\ f_1(x_1)=t_1 = \mathsf{letrec}\ f_2(x_2)=t_2 \in S \not\to T$	falls	$S \not\to T$ Typ und $\{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1)=t_1)(x)\}$ $= \{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_2(x_2)=t_2)(x)\}$ und $t_1[\mathsf{letrec}\ f_1(x_1)=t_1,\ s_1\,/\,f_1,x_1]$ $= t_2[\mathsf{letrec}\ f_2(x_2)=t_2,\ s_2\,/\,f_2,x_2] \in T$ für alle Terme s_1 und s_2 mit $s_1{=}s_2 \in S$.
$S_1 \not\to T_1 = S_2 \not\to T_2 \in \mathsf{U}_j$	falls	$S_1 = S_2 \in \mathsf{U}_j$ und $T_1 = T_2 \in \mathsf{U}_j$

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash S_1 \not\to T_1 = S_2 \not\to T_2 \in \mathsf{U}_j$ [Ax]
by pfunEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash (\mathsf{letrec}\ f_1(x_1)=t_1)$
$= (\mathsf{letrec}\ f_2(x_2)=t_2) \in S \not\to T$ [Ax]
by fixEq j
$\Gamma \vdash \mathsf{letrec}\ f_1(x_1)=t_1 \in S \not\to T$ [Ax]
$\Gamma \vdash \mathsf{letrec}\ f_2(x_2)=t_2 \in S \not\to T$ [Ax]
$\Gamma \vdash \{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1)=t_1)(x)\}$
$= \{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_2(x_2)=t_2)(x)\} \in \mathsf{U}_j$ [Ax]
$\Gamma,\ y{:}\{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1)=t_1)(x)\}$
$\vdash (\mathsf{letrec}\ f_1(x_1)=t_1)\ (y)$
$= (\mathsf{letrec}\ f_2(x_2)=t_2)\ (y) \in T$ [Ax]

$\Gamma \vdash f_1(t_1) = f_2(t_2) \in T$ [Ax]
by apply_pEq $S \not\to T$
$\Gamma \vdash f_1 = f_2 \in S \not\to T$ [Ax]
$\Gamma \vdash t_1 = t_2 \in \{x:S \mid \mathsf{dom}(f_1)(x)\}$ [Ax]

$\Gamma \vdash (\mathsf{letrec}\ f(x)=t)\ (u) = t_2 \in T$ [Ax]
by apply_pRed
$\Gamma \vdash t[\mathsf{letrec}\ f(x)=t,\ u\ /\ f,x] = t_2 \in T$ [Ax]

$\Gamma \vdash \mathsf{letrec}\ f(x)=t \in S \not\to T$ [Ax]
by fixMem j
$\Gamma \vdash S \not\to T \in \mathsf{U}_j$ [Ax]
$\Gamma,\ f'{:}S \not\to T,\ x'{:}S \vdash \mathcal{E}[[\,t[f',x'\,/\,f,x]\,]] \in \mathsf{U}_j$ [Ax]
$\Gamma,\ f'{:}S \not\to T,\ x'{:}S,\ \mathcal{E}[[\,t[f',x'\,/\,f,x]\,]]$
$\vdash t[f',x'\,/\,f,x] \in T$ [Ax]

$\Gamma,\ f{:}S \not\to T,\ \Delta \vdash C$ [**ext** $t[f(s),\mathsf{Axiom}\,/\,y,v]$]
by pfunE i s
$\Gamma,\ f{:}S \not\to T,\ \Delta$
$\vdash s \in \{x:S \mid \mathsf{dom}(f_1)(x)\}$ [Ax]
$\Gamma,\ f{:}S \not\to T,\ \Delta,\ y{:}T,\ v{:}\ y{=}f(s) \in T$
$\vdash C$ [**ext** t]

$\Gamma \vdash \mathsf{dom}(f_1) = \mathsf{dom}(f_2) \in S \to \mathbb{P}_j$ [Ax]
by domEq $S \not\to T$
$\Gamma \vdash f_1 = f_2 \in S \not\to T$ [Ax]
$\Gamma \vdash S \not\to T \in \mathsf{U}_j$ [Ax]

Inferenzregeln

Abbildung 3.27: Syntax, Semantik und Inferenzregeln partiell rekursiver Funktionen

Im Falle der oben erwähnten 3x+1-Funktion liefert der Algorithmus $\mathcal{E}$ zum Beispiel folgendes Domain-Prädikat:

$\lambda x.$ rectype D(y) = y = 1 $\in \mathbb{Z}$
$\vee$ 1<y $\wedge$ y rem 2 = 0 $\in \mathbb{Z}$ $\wedge$ D(y÷2)
$\vee$ 1<y $\wedge$ y rem 2 = 1 $\in \mathbb{Z}$ $\wedge$ D(3*y+1)
select D(x)

Erwartungsgemäß besitzt dieses Domain-Prädikat, das wir zugunsten der natürlicheren Darstellung durch einen parametrisierten induktive Datentyp beschreiben (, in dem wir Datentypkonstrukte durch ihr logisches Gegenstück ersetzt haben), eine sehr große Ähnlichkeit zu der ursprünglichen Funktionsdefinition, da es die rekursive Abarbeitung des Algorithmus in einer induktiven Typstruktur wiederspiegeln muß.

Syntax, Semantik und Regeln der partiellen Funktionen sind in Abbildung 3.27 zusammengefaßt.[77] Man beachte hierbei, daß das Domain-Prädikat beim Schließen über rekursive Funktionen eine zentrale Rolle spielt, sowohl was die Einführung als auch was die Applikation betrifft. Um also $f\,(t)$ untersuchen zu können, muß insbesondere überprüft werden, daß t tatsächlich auch zum Definitionsbereich von f gehört. Dies garantiert, daß f tatsächlich auf t terminiert. Es sei allerdings nochmals darauf hingewiesen, daß f auch auf Eingaben t' terminieren kann, für die dom(f)(t') *nicht* nachgewiesen werden kann. In diesen Fällen kann man allerdings keinerlei Eigenschaften von $f\,(t')$ formal beweisen.

Partiell rekursive Funktionen können nicht als Extrakt-Term einer Einführungsregel fixI erzeugt werden, da hierfür die rekursive Struktur des Definitionsbereiches bereits bekannt sein muß.[78] In diesem Fall kann man auf die Regel recE der induktiven Datentypen zurückgreifen, die das wohlfundierte Gegenstück zu einer rekursiven Funktion erzeugt.

Im Gegensatz zu totalen Funktionenräumen dürfen partiell-rekursive Funktionenräume ohne Einschränkung innerhalb von induktiven Datentypen vorkommen. So ist zum Beispiel $T \equiv$ rectype X = X$\rightharpoonup\mathbb{Z}$ ein erlaubter Datentyp, da der Term letrec f(x) = x (x), dessen Gegenstück λx. x x in Beispiel 3.5.4 so viele Probleme erzeugte, ein legitimes Element von T ist. Grund hierfür ist, daß letrec f(x) = x (x) eine *partielle* Funktion beschreibt, die nur auf Elementen definiert ist, welche selbst wiederum partielle Funktionen sind und auf sich selbst angewandt werden dürfen – für die also dom(x)(x) gilt). Ein solches Element des Definitionsbereiches von letrec f(x) = x (x) ist zum Beispiel die Identitätsfunktion letrec f(x) = x, die ohne Bedenken auf sich selbst angewandt werden kann.

Zum Abschluß sei bemerkt, daß wie bei den induktiven Datentypen auch bei partiellen Funktionen ist eine simultane Rekursion sinnvoll ist. Die allgemeinste Form einer rekursiven Funktionsdefinition lautet daher

letrec f_1(x_1) = t_1 and ... and f_n(x_n) = t_n select f_i

3.5.3 Unendliche Objekte

Bei der Untersuchung rekursiv definierter Datentypen in Abschnitt 3.5.1 haben wir die kleinste Lösung einer rekursiven Typgleichung betrachtet. Die Gleichung

bintree = $\mathbb{Z}$ + $\mathbb{Z}$ $\times$ bintree $\times$ bintree

wurde als Beschreibung aller Terme interpretiert, die sich in endlich vielen Schritten durch die beiden Alternativen der rechten Seite ausdrücken lassen. Dies führte zu einer Interpretation von bintree als dem Datentyp

[77] Es sei an dieser Stelle erwähnt, daß das gegenwärtige NuPRL System die partiellen Funktionen nicht explizit unterstützt sondern durch den Fixpunktkombinator **Y** (siehe Definition 2.3.22 auf Seite 57) simuliert. Der Algorithmus $\mathcal{E}$ zur bestimmung des Definitionsbereiches ist allerdings auf der Meta-Ebene eingebaut. Man kann daher im System durch ein Kommando add_recdef auf der Meta-Ebene eine rekursive Funktionsgleichung eingeben und das System generiert sowohl die rekursive Funktion f als auch eine Beschreibung des Definitionsbereiches, indem es ein Theorem der Form

$\vdash f \in \{x{:}S \mid \text{rectype } F = \mathcal{E}[[t]]\} \to T$

generiert. Diese Simulation hat allerdings den Nachteil, daß der Typ $S\rightharpoonup T$ nicht explizit in Beweisen benutzt werden kann.

[78] In fixMem wird der Algorithmus $\mathcal{E}$ eingesetzt, der eine vorgegebene Funktionsdefinition benötigt. Diese Regel ist also nicht so leicht umkehrbar in eine implizite Erzeugungsregel, wie dies bei anderen Typkonstrukten der Fall ist.

aller endlichen Binärbäume über ganzen Zahlen. Genauso aber hätten wir auch nach der *größten* Lösung dieser Gleichung suchen können. Ein unendlicher Binärbaum erfüllt nämlich ebenfalls die Rekursionsgleichung, da er aus einer Wurzel und zwei unendlichen Binärbäumen zusammengesetzt ist. Somit könnten wir `bintree` genausogut auch als Datentyp aller *endlichen und unendlichen* Binärbäume auffassen. Um diese Semantik innerhalb der Typentheorie zu fixieren, müssen wir einen neuen Term einführen, dessen Syntax ähnlich zu derjenigen der induktiven Typen ist, aber doch auf den Unterschied hinweist. Um auszudrücken, daß wir `bintree` als den *größten* Fixpunkt interpretieren wollen, der die obige Gleichung erfüllt schreiben wir daher

$$\mathsf{inftype}\ \texttt{bintree} = \mathbb{Z} \ + \ \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}.$$

Eine derartige maximale Fixpunktsemantik ist durchaus von praktischer Bedeutung für die Programmierung. Will man zum Beispiel dauernd aktive Prozesse wie Betriebssysteme oder Editoren programmieren, so muß man prinzipiell davon ausgehen, daß ein unendlicher Datenstrom von Eingaben verarbeitet werden muß. Um derartige Datenströme präzise zu beschreiben, braucht man rekursive Gleichungen, die unendliche Lösungen zulassen. Eine NuPRL Formalisierung wäre zum Beispiel:

$$\mathsf{inftype}\ \texttt{stream} = \mathsf{Atom} \ \times \ \texttt{stream}$$

Unendliche Datentypen können maschinenintern zum Beispiel relativ einfach durch zyklische Pointerstrukturen (rückwärts verkettete Listen) realisiert werden, die nicht vollständig sondern nur bedarfsweise verarbeitet werden. Aus diesem Grunde werden sie auch als *lässige Datentypen* angesehen.

Derzeit sind unendliche Datentypen noch nicht als fester Bestandteil in die Typentheorie aufgenommen worden und sollen deshalb hier auch nicht weiter vertieft werden. Eine ausführlichere Abhandlung über unendliche Datentypen kann man in [Mendler, 1987a] finden.

3.6 Ergänzungen zugunsten der praktischen Anwendbarkeit

In den bisherigen Abschnitten haben wir die Grundkonzepte der intuitionistischen Typentheorie besprochen und Inferenzregeln vorgestellt, die eine formales Schließen über diese Konzepte ermöglichen. Wir haben uns dabei bewußt auf einen relativ kleinen Satz von Regeln für jedes einzelne Typkonstrukt eingeschränkt, um die ohnehin schon sehr große Anzahl formaler Regeln nicht völlig unüberschaubar werden. Daher ist in vielen Einzelfällen das formale Schließen über relativ einfache Zusammenhänge bereits sehr aufwendig. Deshalb wurde das Inferenzsystem von NuPRL um eine Reihe von Regeln ergänzt, die aus theoretischer Sicht nicht nötig (weil redundant) sind, das praktische Arbeiten mit dem NuPRL System aber erheblich erleichtern. Die meisten dieser Regeln, die wir im folgenden kurz beschreiben wollen, beinhalten etwas komplexere Algorithmen oder beziehen sich auf *Objekte* wie Theoreme und Definitionen, die innerhalb der Bibliothek des NuPRL Systems unter einem bestimmten Namen abgelegt sind.

- Bei der Fixierung der Bedeutung typentheoretischer Ausdrücke haben wir in Definition 3.2.15 Urteile über nichtkanonische Terme mithilfe von Urteilen über die Werte dieser Terme definiert. Es ist also legitim, einen nichtkanonischen Term t in einer Konklusion oder einer Annahme durch einen anderen Term zu ersetzen, welcher zu dem gleichen Wert reduziert werden kann, insbesondere also einen Teilterm von t zu reduzieren oder eine Reduktion, die zu einem Teilterm von t geführt hat, wieder rückgängig zu machen. Zu diesem Zweck wurden insgesamt vier *Berechnungsregeln* eingeführt, deren Anwendung durch eine *Markierung* (tagging) der entsprechenden Teilterme kontrolliert wird.

  Eine solche Markierung eines Teilterms s von t geschieht dadurch, daß s durch `[[*:`s`]]` bzw. `[[`n`:`s`]]` ersetzt wird, wobei n die Anzahl der durchzuführenden Reduktionsschritte angibt. Eine Markierung mit $*$ bedeutet, daß so viele (lässige) Reduktionsschritte wie möglich ausgeführt werden sollen. Im Falle einer Rückwärtsberechnung muß anstelle von s der markierte Term angegeben werden, der zu s reduziert. Für die Anwendung der Regel ist der gesamte Term mit seinen markierten Teiltermen anzugeben. Die Berechnungsregel führt die entsprechenden Reduktionen durch und ersetzt t dann durch den reduzierten bzw. rückreduzierten Term (bezeichnet durch $t{\downarrow_{tagt}}$ bzw. $t{\uparrow_{tagt}}$).

$\Gamma \vdash C$ ⌊**ext** t⌋ **by** `compute` *tagC* $\Gamma \vdash C\downarrow_{tagC}$ ⌊**ext** t⌋	$\Gamma \vdash C$ ⌊**ext** t⌋ **by** `rev_compute` *tagC* $\Gamma, \vdash C\uparrow_{tagC}$ ⌊**ext** t⌋
$\Gamma, z{:}T, \Delta \vdash C$ ⌊**ext** t⌋ **by** `computeHyp` i *tagT* $\Gamma, z{:}T\downarrow_{tagT}, \Delta \vdash C$ ⌊**ext** t⌋	$\Gamma, z{:}T, \Delta \vdash C$ ⌊**ext** t⌋ **by** `rev_computeHyp` i *tagT* $\Gamma, z{:}T\uparrow_{tagT}, \Delta \vdash C$ ⌊**ext** t⌋
$\Gamma \vdash C$ ⌊**ext** t⌋ **by** `unfold` *def-name* $\Gamma \vdash C_{\downarrow}$ ⌊**ext** t⌋	$\Gamma \vdash C_{\downarrow}$ ⌊**ext** t⌋ **by** `fold` *def-name* $\Gamma \vdash C$ ⌊**ext** t⌋
$\Gamma, z{:}T, \Delta \vdash C$ ⌊**ext** t⌋ **by** `unfoldHyp` i *def-name* $\Gamma, z{:}T_{\downarrow}, \Delta \vdash C$ ⌊**ext** t⌋	$\Gamma, z{:}T_{\downarrow}, \Delta \vdash C$ ⌊**ext** t⌋ **by** `foldHyp` i *def-name* $\Gamma, z{:}T, \Delta \vdash C$ ⌊**ext** t⌋
$\Gamma, z{:}T, \Delta \vdash C$ ⌊**ext** t⌋ **by** `replaceHyp` i S j $\Gamma, z{:}S, \Delta \vdash C$ ⌊**ext** t⌋ $\Gamma, z{:}T, \Delta \vdash T = S \in \mathsf{U}_j$ ⌊Ax⌋	
$\Gamma \vdash C$ ⌊**ext** t⌋ **by** `lemma` *theorem-name*	$\Gamma \vdash t \in T$ ⌊Ax⌋ **by** `extract` *theorem-name*
$\Gamma \vdash C$ ⌊**ext** $t[\sigma]$⌋ **by** `instantiate` Γ' C' σ $\Gamma' \vdash C'$ ⌊**ext** t⌋	$\Gamma \vdash C$ ⌊**ext** $t[y/x]$⌋ **by** `rename` y x $\Gamma[x/y] \vdash C[x/y]$ ⌊**ext** t⌋
$\Gamma \vdash s = t \in T$ ⌊Ax⌋ **by** `equality`	$\Gamma \vdash C$ ⌊**ext** t⌋ **by** `arith` j $\Gamma \vdash s_1 \in \mathbb{Z}$ ⌊Ax⌋ ⋮

Abbildung 3.28: Zusätzliche Inferenzregeln von NuPRL

Die Berechnungsregeln machen die einzelnen Reduktionsregeln eigentlich hinfällig, da sie diese subsumieren. Aus theoretischen Gründen müssen diese jedoch in der Theorie enthalten bleiben. Für partiell rekursive Funktionen gibt es *keine direkten Berechnungsregeln*, da für diese eine Terminierung nicht garantiert wäre.

- Die Existenz von benutzerdefinierten Erweiterungen wie zum Beispiel Logik-Operatoren gemäß Definition 3.3.3 auf Seite 134 macht es zuweilen nötig, zwischen dem neu eingeführten Term und seiner ausführlichen Definition hin- und herzuwechseln. Dies ist insbesondere dann wichtig, wenn die Grundeigenschaften des neuen Konzeptes bewiesen werden sollen. Aus diesem Grunde wurden Regeln hinzugenommen, die in einer Konklusion bzw. einer Hypothese derartige Definitionen auflösen (`unfold`) oder zurückfalten (`fold`). Anzugeben ist hierbei jeweils der Name, unter dem die Definition in der Bibliothek abgelegt ist.[79]

- Die `replaceHyp`-Regel ist eine Variante der Substitutionsregel. Sie erlaubt es, eine Hypothese T durch eine gleichwertige Hypothese S zu ersetzen

- Die `lemma`- und `extract`-Regeln erlauben einen Zugriff auf bereits bewiesene Theoreme der NuPRL-Bibliothek. Die `lemma`-Regel entspricht der `hypothesis`-Regel in dem Sinne, daß sie eine Konklusion

[79] Intern wird die Auflösung von Definitionen genauso gehandhabt wie die Durchführung eines Reduktionsschrittes.

dadurch beweist, daß auf eine bereits bekannte Tatsache – nämlich das bewiesene Theorem – verwiesen wird. Die `extract`-Regel entspricht in ähnlicher Weise der `hypEq`-Regel. hier wird eine Konklusion $t \in T$ dadurch bewiesen, daß ein Theorem genannt wird, dessen Beweisziel T und dessen Extraktterm t ist. In beiden Fällen ist der Name des entsprechenden Theorems anzugeben.

- Die `instantiate`-Regel ermöglicht eine Verallgemeinerung des Beweiszieles. $\Gamma \vdash C$ wird als Spezialfall (Instanz) eines allgemeineren Zieles $\Gamma' \vdash C'$ angesehen und folgt deshalb hieraus. Als Parameter sind Γ', C' und die Substitution σ anzugeben, welche Γ' zu Γ und C' zu C instantiiert.

- Mit der `rename`-Regel können Variablen innerhalb eines Beweiszieles umbenannt werden. Dies ist nützlich, wenn die Variablennamen mit einer Bedeutung assoziiert werden sollen oder wenn unsichtbare Variablen (wie bei unabhängigen Funktionenräumen) sichtbar gemacht werden sollen.

- Die `equality`- und `arith`-Regeln schließlich sind dafür geschaffen worden, in zwei speziellen Anwendungsbereichen dem Benutzer die Anwendung vieler einzelner Regeln zum Beweis relativ trivialer Aussagen zu ersparen. Hinter diesen Regeln steckt nicht mehr ein einzelner Beweisschritt sondern ein aufwendiger, theoretisch abgesicherter Algorithmus, welcher Entscheidungen über die Gültigkeit von Gleichheitsaussagen bzw. von einfachen arithmetische Aussagen treffen kann.

  So subsumiert die `equality`-Regel die Anwendung vieler Einzelschritte, die nur auf Kommutativität, (typisierter) Reflexivität oder Transitivität beruhen, in einem einzigen Schritt. Die `arith`-Regel beinhaltet alle Regeln über Addition, Subtraktion, Multiplikation, Division, Divisionsrest, arithmetische Tests sowie diverse Monotoniegesetze und kann praktisch alle arithmetischen Aussagen in einem Schritt beweisen, die keine Induktion benötigen.

  Die Algorithmen, die hinter diesen *Entscheidungsprozeduren* stehen und ihre Rechtfertigung werden wir im Kapitel 4.3 ausführlich besprechen.

3.7 Diskussion

Wir haben in diesem Kapitel die intuitionistische Typentheorie als einen Formalismus vorgestellt, der für sich in Anspruch nimmt, alle Aspekte von Mathematik und Programmierung zuverlässig repräsentieren zu können. Dabei sind wir davon ausgegangen, daß es für gewisse mathematische Grundkonstrukte ein intuitives Verständnis gibt, was sich nicht weiter in sinnvolle elementarer Bestandteile zerlegen läßt. Beim Aufbau der formalen Theorie ging es deshalb darum, die "natürlichen" Gesetze der zentralen in Mathematik und Programmierung vorkommenden Grundkonzepte in Form einer präzisen Semantikdefinition auszuformulieren und darüber hinaus auch für einen praktisch einsetzbaren Inferenzkalkül verwendbar zu machen. Auf eine minimale Theorie, in der kein Konzept durch ein anderes simuliert werden kann, wurde zugunsten einer natürlicheren Formalisierung der Grundkonstrukte verzichtet. Aus diesem Grunde ist die Theorie relativ umfangreich und enthält mehr als 100 Inferenzregeln.[80] Um die Theorie dennoch überschaubar zu halten und eine spätere Automatisierung der Beweisführung zu unterstützen, haben wir zu Beginn dieses Kapitels eine Systematik für einen einheitlichen Aufbau von Syntax, Semantik und Inferenzregeln entwickelt und anschließend die eigentliche Theorie in 4 Phasen – mathematische Grundkonstrukte, Logik, Grundkonstrukte der Programmierung und Rekursion – entwickelt.

Im Laufe dieser Entwicklung sind einige Prinzipien wichtig geworden, welche die konkrete Ausgestaltung der Theorie entscheidend beeinflußt hatten.

[80] Dies ist aber nur ein scheinbarer Nachteil gegenüber minimalen Theorien. Bei letzteren müßten Konzepte wie Zahlen, Listen auf die bestehende Theorie aufgesetzt und ihre Grundgesetze als Theoreme bewiesen werden, bevor man sie benutzen kann. Andernfalls würde alles formale Schließen über Zahlen auf einer Ebene unterhalb der Grundgesetze von Zahlen stattfinden, was eine (interaktive) Beweisführung durch einen Menschen praktisch ausschließt.

1. Die Typentheorie ist als eine konstruktive mathematische *Grundlagentheorie* ausgelegt, deren Konzepte sich direkt aus einem intuitiven mathematischen Verständnis ableiten lassen und nicht formal auf einer anderen Theorie abgestützt werden (siehe Abschnitt 3.1.1, Seite 94).

2. Die wichtigsten Grundkonstrukte der Mathematik und Programmierung sowie ihre Gesetze und Regeln sind als feste Bestandteile der Theorie formalisiert (Abschnitt 3.1.2, Seite 95).

3. Die Abstraktionsform eines Terms und seine Darstellungsform werden getrennt voneinander behandelt aber simultan betrachtet (Entwurfsprinzip 3.2.4, Seite 103).

4. Semantik und Inferenzregeln basieren auf einem immer terminierenden Reduktionskonzept. Als Berechnungsverfahren wurde *Lazy Evaluation* fixiert (Entwurfsprinzip 3.2.9, Seite 107).

5. *Urteile* sind die semantischen Grundbausteine, mit denen Eigenschaften von Termen und die Beziehungen zwischen Termen formuliert werden (Abschnitt 3.2.2.2, Seite 109).

6. Die semantische Typeigenschaft wird innerhalb des Inferenzsystems durch eine kumulative Hierarchie von Universen syntaktisch repräsentiert (Entwurfsprinzip 3.2.21, Seite 115).

7. Die Typentheorie unterstützt explizites und implizites Schließen, also die Überprüfung eines Urteils und die Konstruktion von Termen, die ein Urteil erfüllen. Zu diesem Zweck werden Urteile syntaktisch immer implizit dargestellt (Entwurfsprinzip 3.2.24, Seite 117).

8. Logische Aussagen werden dargestellt durch Typen, deren Elemente den Beweisen der Aussagen entsprechen ("*Propositionen als Typen*"-Prinzip 3.2.22, Seite 116).

   Konsequenterweise ist die Prädikatenlogik kein Grundbestandteil der Typentheorie sondern ein simulierbares Konzept, welches als *konservative Erweiterung* hinzugenommen werden kann. Hierbei spielt die Curry-Howard Isomorphie zwischen Logik und Typentheorie, die im Prinzip auch eine Logik höherer Stufe ermöglicht, eine fundamentale Rolle (Abschnitt 3.3.1, Seite 127).

9. Aufgrund des Prinzips "*Propositionen als Typen*" ist Programmentwicklung dasselbe wie Beweisführung ("*Beweise als Programme*"-Prinzip 3.4.3, Seite 139). Insbesondere ist induktive Beweisführung dasselbe wie die Konstruktion rekursiver Programme (Entwurfsprinzip 3.4.4, Seite 142).

Hinter vielen dieser Prinzipien steckt eine bewußte Entwurfsentscheidung, bei denen die absehbaren Nachteile zugunsten der erwarteten Vorteile in Kauf genommen wurden. Die Entscheidungen hätten durchaus auch anders gefällt werden können und hätten dann zu anderen Ausformulierungen der Typentheorie geführt.[81] Aus praktischer Hinsicht sind besonders die letzten beiden Prinzipien bedeutend, denn sie ermöglichen es, innerhalb eines einzigen in sich geschlossenen Kalküls über Programme und Beweise zu schließen und diese aus einer vorgegebenen Problemstellung auch zu konstruieren. Die Typentheorie eignet sich damit für die mathematische Beweisführung, Programmverifikation und die Synthese von Programmen aus ihren Spezifikationen.

Aus theoretischer Hinsicht ist die Typentheorie ein extrem mächtiger Formalismus, da es nun prinzipiell möglich ist, formale Schlüsse über alle Aspekte der Mathematik und Programmierung zu führen. Dennoch zeigen die Beispiele relativ schnell Grenzen des praktischen Einsatzes der bisherigen Theorie auf. Bereits einfache Aufgaben – wie zum Beispiel der Nachweis der Existenz einer Integerquadratwurzel (siehe Beispiel 3.4.9 auf Seite 154) – können nur mit einem relativ großen Aufwand gelöst werden und lassen sich kaum vollständig präsentieren. Zwei wesentliche Probleme machen sich hierbei bemerkbar.

[81] Diese Wege wurden durchaus beschritten und zu anderen Typtheorien mit entsprechend anderen Schwerpunkten geführt haben. Die wichtigsten Vertreter sind Girard's *System* $\mathcal{F}$ und $\mathcal{F}_\omega$ [Girard, 1971, Girard, 1972, Girard, 1986, Girard *et.al.*, 1989], der *Kalkül der Konstruktionen* von Coquand und Huet [Coquand & Huet, 1985, Coquand & Huet, 1988, Harper & Pollack, 1989, Coquand, 1990] sowie seine Erweiterung *ECC* [Luo, 1989, Luo, 1990, Pollack, 1994] die *lineare Logik* [Girard, 1987] sowie diverse leicht modifizierte Formulierungen von Martin-Löf's Typentheorie.

- Die Entwicklung eines formalen Beweises ist für einen Menschen mit extrem viel Schreibarbeit verbunden, bei der immer eine gewisse Gefahr für Schreibfehler besteht, welche den geführten Beweis verfälschen könnten. Dies ist nicht verwunderlich, denn der Zweck formaler Kalküle ist ja eigentlich, eine schematische Beweisführung zu unterstützen, die nicht unbedingt von einem Menschen ausgeführt werden muß. Die Ausführung formaler Beweise "von Hand" dient eigentlich nur Kontroll- und Übungszwecken, um ein gewisses Gefühl für den Kalkül zu entwickeln. Wirkliche Beweise sollten allerdings interaktiv mit Hilfe von Rechnern geführt werden, welche die Ausführung von Regeln übernehmen und vom Menschen nur die Angabe der auszuführenden Regel erwarten. Ein praktischer Umgang mit der Typentheorie macht also den Bau *interaktiver Beweissysteme* unbedingt erforderlich.

- Dies aber ist noch nicht genug, denn auch interaktive geführte Beweise enthalten eine große Menge von Details, welche sie sehr unübersichtlich werden lassen. Dies liegt zum einen an der großen Menge von Regeln, die nötig ist, um einen Beweis vollständig zu führen und zum anderen an den vielen Parametern, die für die Ausführung einer Regel erforderlich sind. Aus diesem Grunde ist es notwendig, eine Rechnerunterstützung anzubieten, die über die Möglichkeiten interaktiver Beweissysteme hinausgeht. Es müssen Techniken bereitgestellt werden, mit denen die Beweisführung zumindest zum Teil automatisiert werden kann – sowohl was die Auswahl der Regeln als auch die Bestimmung ihrer Parameter angeht.

Mit der Implementierung zuverlässiger interaktiver Beweissysteme (für die intuitionistische Typentheorie) und den prinzipiellen Möglichkeiten der Automatisierung der Beweisführung werden wir uns im Kapitel 4 befassen. Konkrete Formalisierungen von mathematischen Theorien und Programmierproblemen sowie Strategien für die Suche nach Beweisen und die Entwicklung von Programmen aus formalen Spezifikationen werden uns dann in den darauffolgenden Kapiteln beschäftigen.

3.8 Ergänzende Literatur

Zur intuitionistischen Typentheorie gibt es noch kein zusammenfassendes Lehrbuch sondern nur eine Reihe von Büchern und Fachartikeln, in denen komplette Formalisierungen vorgestellt oder spezielle Ergänzungen im Detail untersucht werden. Die Bandbreite der Notationen ist noch sehr breit und konnte bisher nicht standardisiert werden. Die hier betrachtete Formalisierung der Martin-Löf'schen Typentheorie lehnt sich an die im NuPRL-System verwandte Syntax an und ist weitestgehend in [Constable *et.al.*, 1986] beschrieben.

Lesenswert sind auch die Einführungskapitel einiger Bücher und Tutorials, die allgemeine Einführungen in die Martin-Löf'sche Typentheorie geben wie [Martin-Löf, 1973, Martin-Löf, 1984, Nordström *et.al.*, 1990, Backhouse *et.al.*, 1988a, Backhouse *et.al.*, 1988b, Backhouse, 1989]. Von Bedeutung für das Verständnis sind auch Martin-Löf's frühere Originalarbeiten [Martin-Löf, 1970, Martin-Löf, 1982]. Eine alternative Typentheorie beschreibt das Buch von Girard [Girard *et.al.*, 1989]. Es enthält aber eine Reihe detaillierter Beweise über Grundeigenschaften der Typentheorie, von denen sich fast alle direkt auf die hier vorgestellte Theorie übertragen lassen. Eine andere gute Quelle ist das Buch von Andrews [Andrews, 1986].

Der Aspekt der Programmentwicklung in der Typentheorie ind die Extraktion von Programmen aus Beweisen ist das Thema vieler Fachartikel wie [Bates & Constable, 1985, Backhouse, 1985, Constable, 1983, Constable, 1984, Constable, 1985, Constable, 1988, Constable & Howe, 1990b, Hayashi, 1986, Chisholm, 1987, Smith, 1983b, Nordström & Smith, 1984, Nordström *et.al.*, 1990, Paulin-Mohring, 1989, Leivant, 1990]. Jede Methode bringt vor und Nachteile mit sich. Eine optimale Extraktionsform konnte bisher allerdings noch nicht gefunden werden.

Ähnlich aufwendig sind rekursive Datentypen und Funktionen, für die ebenfalls noch keine optimale Darstellung gefunden werden konnte. Der gegenwärtige Stand der Forschung ist in [Constable *et.al.*, 1986, Kapitel 12] und Arbeiten wie [Constable & Mendler, 1985, Constable & Smith, 1987, Constable & Smith, 1988, Constable & Smith, 1993, Mendler, 1987a, Mendler, 1987b, Smith, 1988, Coquand & Paulin, 1988, Freyd, 1990,

Martin-Löf, 1988] dokumentiert. Weitere Betrachtungen zu einzelnen Typkonstrukten und ihren Anwendungen sind in [Amadio & Cardelli, 1990, Backhouse, 1984, Chisholm, 1985, Cleaveland, 1987, Felty, 1991, Paulson, 1986, Pitts, 1989, Salvesen & Smith, 1988] zu finden.

Semantische Aspekte der Typentheorie werden in [Aczel, 1978, Allen, 1987a, Allen, 1987b, Allen *et.al.*, 1990, Constable, 1989, Constable & Howe, 1990b, Mendler, 1987a, Smith, 1984, Schwartzbach, 1986] betrachtet. Diese nicht immer ganz leicht zu lesenden Arbeiten geben wertvolle Einsichten in das Selbstverständnis der Typentheorie als formaler Grundlagentheorie und ihrer konkreten Ausformulierungen. Das Buch von Bishop [Bishop, 1967] liefert weitere Hintergründe zur Denkweise der konstruktiven Mathematik. Für ihr klassisches Gegenstück – die klassische Mengentheorie – lohnt es sich [Suppes, 1972, Quine, 1963] zu konsultieren.

Kapitel 4

Automatisierung des formalen Schließens

Im vorhergehenden Kapitel haben wir die intuitionistische Typentheorie schrittweise aus dem einfach getypten λ-Kalkül heraus entwickelt und damit einen mächtigen Formalismus aufgestellt, mit dem wir formale Schlüsse über alle Aspekte der Mathematik und Programmierung ziehen können. Diese Theorie liefert uns alles, was wir brauchen, um formale Beweise "von Hand" zu entwerfen. Ihr *praktischer* Nutzen liegt vor allem darin, daß durch die Formulierung von Inferenzregeln gezeigt wird, wie semantisches Schließen und die Verwendung mathematischer Erkenntnisse auf eine rein syntaktische Manipulation von Texten (Sequenzen) reduziert werden kann – also auf eine Aufgabe, für die der Einsatz von Computern geradezu prädestiniert ist.

Wir wollen uns in diesem Kapitel damit auseinandersetzen, auf welche Art eine sinnvolle maschinelle Unterstützung für die Entwicklung von Programmen und dem Beweis von Theoremen in der intuitionistischen Typentheorie geschaffen werden kann, also wie wir unserem ursprünglichen Ziel – der Automatisierung von Logik und Programmierung – durch den Entwurf von Beweissystemen für die Typentheorie näher kommen können. Welche grundsätzlichen Möglichkeiten bieten sich hierfür nun an?

Die elementarste Art der maschinellen Unterstützung für das formale Beweisen ist *Proof Checking* – die Überprüfung formal geführter Beweise durch einen Computer. Diese Vorgehensweise wurde erstmalig in voller Konsequenz innerhalb des AUTOMATH Projektes [Bruijn, 1980, van Benthem Jutting, 1977] verfolgt und bietet sich vor allem dann an, wenn eine sehr ausdrucksstarke formale Sprache zur Verfügung steht, die in der Lage ist, die für eine rigorose Formalisierung mathematischer Konzepte notwendigen Abstraktionen zu erfassen. Da Proof Checking sich nur auf ein Minimum algorithmischer Beweisführung stützt – nämlich nur auf die Kontrolle einer korrekten Regelanwendung, die in einem Beweisterm codiert ist – sind die entsprechenden Systeme sehr sicher und leicht zu programmieren. Für einen Benutzer sind sie allerdings nur sehr mühsam zu handhaben, da er praktisch alle formalen Informationen von Hand im Voraus bestimmen muß. Aufgrund der geringen maschinellen Unterstützung ist der *Verlustfaktor*[1] zwischen formalen und gewöhnlichen Beweisen extrem hoch, was formale Beweise sehr schwer zu lesen macht und den Umgang mit Proof Checkern wenig begeisternd erscheinen läßt.

Deutlich eleganter und genauso sicher und einfach zu programmieren sind Beweis*entwicklungs*systeme (*Proof editors*), bei denen die formalen Regeln des Kalküls nicht zur Überprüfung sondern zum Entwurf von Beweisen eingesetzt werden können. Der Benutzer wird hierbei davon entlastet, den Beweisterm im Voraus anzugeben. Stattdessen entwickelt er ihn in Kooperation mit dem System, indem er schrittweise die jeweils anzuwendende Regel angibt. Die Umwandlung der Regeln der einfachen Typentheorie in Regeln mit einer impliziten Darstellung der Beweisterme in Abschnitt 3.2.3.3 (Seite 117ff.) zielte genau auf diesen Vorteil ab. Auch hier ist das Maß an automatischer Unterstützung jedoch noch sehr gering, da ein Benutzer nur von dem Aufschreiben formaler Beweisterme entlastet wird, nicht aber davon, den gesamten Beweis selbst zu führen.

Das Gegenextrem zur Beweisüberprüfung bildet das *automatische Theorembeweisen*, das von verschiedenen Beweissystemen [Bledsoe, 1977, Bibel, 1987, Bibel *et.al.*, 1994, Bläsius *et.al.*, 1981, Boyer & Moore, 1979,

[1] Dieser Verlustfaktor drückt aus, um wieviel länger formale Beweise werden, wenn man sorgfältige 'normale' Beweise in die formale Sprache überträgt. In den meisten Proof Checkern liegt er zwischen 20 und 50. Bei der *Entwicklung* von Beweisen mit den 'reinen' Regeln der Typentheorie ist der Faktor geringer, aber immer noch zu hoch für ein praktisches Arbeiten.

Letz *et.al.*, 1992, Wos *et.al.*, 1984, Wos *et.al.*, 1990] angestrebt wird. Dieser Ansatz stützt sich auf die Tatsache, daß alle mit einem Kalkül beweisbaren Theoreme im Endeffekt durch eine vollständige Suche gefunden werden können.[2] Tatsächlich konnten durch automatische Theorembeweiser bereits eine Reihe bisher unbekannter Resultate nachgewiesen werden. Im allgemeinen ist Theorembeweisen jedoch sehr aufwendig, da die Gültigkeit mathematischer Sätze schon in der Prädikatenlogik erster Stufe nicht mehr entscheidbar ist. Die meisten Verfahren stützen sich daher auf maschinennahe Charakterisierungen der Gültigkeit von Sätzen der betrachteten Theorie und vor allem auf heuristische Suchstrategien, um auf diese Art die Effizienz der Suche zu steigern und eine größere Menge von Problemen in akzeptabler Zeit lösen zu können. Dies ist bisher jedoch nur für relativ "kleine" Theorien wie die Prädikatenlogik erster Stufe mit geringfügigen Erweiterungen möglich und macht eine Übertragung der Ergebnisse auf reichhaltigere Theorien nahezu unmöglich. Für die Typentheorie und andere formale Theorien mit einer ähnlich hohen Ausdruckskraft ist dieser Ansatz daher unbrauchbar.

Ein praktikabler Mittelweg zwischen reiner Beweisüberprüfung und heuristisch gesteuerten automatischen Beweisern ist, Beweisentwicklungssysteme um *Entscheidungsprozeduren* für gewisse einfache Teiltheorien zu erweitern. In derartigen Systemen beschränkt sich die automatische Unterstützung auf solche Probleme, die mit Hilfe von Algorithmen schnell erkannt und *entschieden* werden können. Dabei basieren die eingesetzten Entscheidungsprozeduren auf einer grundlegenden Analyse der Teiltheorie, für die sie eingesetzt werden, und auf einem komplexen, aber immer terminierenden Algorithmus, dessen Korrektheit nachgewiesen werden kann. Auf diese Art werden die Benutzer des Systems von vielen lästigen Teilaufgaben entlastet und die Zuverlässigkeit des Gesamtsystems bleibt gesichert.

Eine Möglichkeit die Stärken dieses Ansatzes mit denen der automatischen Beweiser zu verbindet, bildet das Konzept der Beweis*taktiken*, welches erstmals im LCF Project [Gordon *et.al.*, 1979] der University of Edinburgh untersucht wurde und sich im Laufe der Jahre als sehr leistungsfähig herausgestellt hat.[3] Die Schlüsselidee war dabei, ein flexibles System zum Experimentieren mit einer Vielfalt von Strategien zu entwickeln, indem einem Benutzer ermöglicht wird, den Inferenzmechanismus um eigene Methoden zu erweitern, ohne daß hierdurch die Sicherheit des Systems gefährdet werden kann. Dies kann dadurch geschehen, daß die Regeln des zugrundeliegenden Kalküls nach wie vor die einzige Möglichkeit zur Manipulation von Beweisen sind, aber ein Mechanismus geschaffen wird, ihre Anwendung durch (Meta-)Programme zu steuern. Dies verlangt allerdings eine Formalisierung der bis dahin nur informal vorliegenden Metasprache des Kalküls als eine interaktive Programmiersprache, in der alle objektsprachlichen Konzepte wie Terme, Sequenzen, Regeln und Beweise zu programmieren sind.

In diesem Kapitel wollen wir nun die grundsätzlichen Techniken vorstellen, die bei der Realisierung eines derartigen taktischen Theorembeweisers eine Rolle spielen können, und diese am Beispiel der konkreten Implementierung des NuPRL-Systems illustrieren.[4] Konkrete Inferenzmethoden zur Automatisierung der Beweisführung stehen in diesem Kapitel eher im Hintergrund, da es uns mehr um die Möglichkeiten dieser Techniken als solche geht. In Abschnitt 4.1 werden wir zunächst diskutieren, welche Grundbausteine notwendig oder sinnvoll sind um Systeme zu bauen, mit denen absolut korrekte Beweise interaktiv entwickelt werden können. In Abschnitt 4.2 werden wir zeigen, wie die Rechnerunterstützung bei der Beweisführung durch das Konzept der Taktiken gesteigert werden kann und in Abschnitt 4.3 werden wir am Beispiel zweier erfolgreicher Entscheidungsprozeduren des NuPRL Systems beschreiben, wie man Teiltheorien der Typentheorie vollautomatisch entscheiden kann.

[2] Das ist der aus der theoretischen Informatik bekannte Zusammenhang zwischen *Beweisbarkeit* und *Aufzählbarkeit*.

[3] Das Taktik-Konzept wurde vor allem von Beweissystemen aufgegriffen, mit denen sehr komplexe Aufgaben gelöst werden sollten. Neben dem NuPRL-System, das wir hier detaillierter vorstellen werden, sind dies vor allem Cambridge LCF (Schließen über funktionale Programme) [Paulson, 1987], HOL (Logik höherer Ordnung) [Gordon., 1985, Gordon., 1987], λ-Prolog (Prolog mit einer Erweiterung um λ-Terme) [Felty & Miller, 1988, Felty & Miller, 1990], OYSTER (ein Planungssystem für Programmsynthese) [Bundy, 1989, Bundy *et.al.*, 1990], ISABELLE (ein universeller (generischer) Beweiser) [Paulson, 1989, Paulson, 1990] und KIV (ein Verifikationssystem für imperative Programme) [Heisel *et.al.*, 1988, Heisel *et.al.*, 1990], LEGO (ein Beweissystem für ECC) [Luo & Pollack, 1992, Pollack, 1994], ALF (ein Beweissystem für Martin-Löf's Typentheorie) [Altenkirch *et.al.*, 1994].

[4] Es sei angemerkt, daß die in diesem Kapitel besprochenen Methoden im Prinzip nicht von der Typentheorie abhängen sondern sich genauso mit anderen Objekttheorien realisieren lassen.

4.1 Grundbausteine interaktiver Beweisentwicklungssysteme

Im Gegensatz zu vollautomatischen Beweissystemen, bei denen die Effizienz der Beweisführung im Vordergrund steht, kommt es bei interaktiven Beweissystemen vor allem darauf an, Beweise und Programme in einer für Menschen verständlichen Form entwickeln zu können und in einer Art aufzuschreiben, wie dies auch in einem mathematischen Lehrbuch üblich ist. Das bedeutet, daß ein Benutzer in der Lage sein muß, Definitionen einzuführen, Theoreme aufzustellen und mit Hilfe des Systems zu beweisen (wobei dieses die Korrektheit des Beweises garantiert), Programme aus Beweisen zu extrahieren und auszuführen und seine Ergebnisse in einer "Bibliothek" zu sammeln, die praktisch einem Buch entspricht.

Um dies zu unterstützen, braucht ein praktisch nutzbares Beweisentwicklungssystem neben einer Implementierung der objektsprachlichen Konzepte eine Reihe von Mechanismen, welche eine Interaktion mit einem Benutzer unterstützen.[5]

- Eine *Bibliothek* (*library*), in der verschiedene vom Benutzer eingeführte Objekte wie Definitionen, Sätze, Kommentare etc. enthalten sind.
- Eine *Kommandoebene*, mit der Objekte der Bibliothek erzeugt, gelöscht oder anderweitig manipuliert werden können (dies schließt den Aufruf geeigneter Editoren ein) und andere Interaktionen mit dem System – wie das Laden, Sichern oder Aufbereiten von Bibliotheken – gesteuert werden können.
- Ein *Beweiseditor*, mit dem die Behauptungen eines Theorems aufgestellt und bewiesen werden können. Dieser hat vor allem die Aufgabe, die Korrektheit von Beweisen sicherzustellen und dem Anwender unnötige Schreibarbeit zu ersparen. Zu dem Beweiseditor gehört auch ein *Extraktionsmechanismus*, mit dem die implizit in einem Beweis enthaltenen Programme extrahiert werden können.
- Einen *Text- und Termeditor*, der die Erstellung syntaktisch korrekter Terme unterstützt.
- Einen *Definitionsmechanismus*, welcher konservative Erweiterungen der zugrundeliegenden Theorie mit einer flexiblen Darstellungsform unterstützt.
- Ein *Programmevaluator*, mit dem die generierten Programme auch innerhalb des Systems überprüft und ausgetestet werden können.

Im folgenden werden wir die wichtigsten Aspekte dieser Komponenten und einer Implementierung der objektsprachlichen Konzepte diskutieren. Zuvor werden wir kurz auf die Formalisierung der Metasprache eingehen, die wir für die Implementierungsarbeiten und eine Integration des Taktik-Konzeptes benötigen.

4.1.1 ML als formale Beschreibungssprache

Zur Beschreibung der Konzepte, die beim Aufbau eines formalen Kalküls eine Rolle spielen, hatten wir uns bisher einer halbformalen Metasprache bedient, die gemäß unserer Vereinbarung in Abschnitt 2.1.4 aus der natürlichen Sprache, Bestandteilen der Objektsprache und sogenannten syntaktischen Metavariablen bestand. Wenn wir nun beschreiben wollen, wie diese Konzepte innerhalb von Beweisunterstützungssystemen durch Algorithmen und Datenstrukturen zu realisieren sind, dann liegt es nahe, die Metasprache stärker zu formalisieren und als Ausgangspunkt einer Implementierung zu verwenden. Diese formale Metasprache hat den

[5] Prinzipiell könnte man auch ohne die hier genannten Konzepte auskommen und sich allein auf die Implementierung der Objektsprache, die Kommandoebene und ein Taktik-Konzept konzentrieren. Ein Verzicht auf den Beweiseditor hätte jedoch zur Folge, daß ein Benutzer seinen Beweis komplett von außen programmieren müßte. Der Verzicht auf einen flexiblen Definitionsmechanismus macht formale Theoreme nahezu unlesbar. Ohne den Termeditor müßte die Flexibilität der Darstellung von Termen auf dem Bildschirm begrenzt werden und ein Benutzer müßte sich die korrekte Syntax aller Terme merken. Aus diesem Grunde sollte man in einem praktisch verwendbaren System nicht darauf verzichten.

Vorteil, daß sie aus intuitiv verständlichen mathematischen Konzepten aufgebaut ist, die keiner ausführlichen Erklärung bedürfen, und dennoch formal genug ist, um als Programmiersprache verwendbar zu sein.

Da man im allgemeinen davon ausgehen kann, daß die Idee einer Funktion intuitiv klar ist, bietet es sich an, diese Metasprache aus bekannten einfachen Funktionen aufzubauen und diese um einfache Strukturierungskonzepte zu ergänzen. Dieser Gedanke wurde erstmals im Rahmen des LCF-Projects [Gordon *et.al.*, 1979] verfolgt und führte zur Entwicklung der formalen Metasprache ML (MetaLanguage), die ebenfalls bei der Implementierung von NuPRL eingesetzt wird.[6] Drei wichtige Charakteristika machen ML für diese Zwecke besonders geeignet.

- ML ist – wie der λ-Kalkül – eine funktionale Programmiersprache *höherer Stufe*: es gibt keine prinzipiellen Restriktionen an die Argumente einer Funktion.
- ML besitzt eine *erweiterbare und polymorphe Typdisziplin* mit sicheren (abstrakten) Datentypen. Als Kontrollinstrument dient eine erweiterte Form des Typechecking Algorithmus von Hindley und Milner.
- ML besitzt einen Mechanismus um *Ausnahmen* (*exceptions*) zu erzeugen und zu verarbeiten.

Da ML im wesentlichen die übliche mathematische Notation verwendet, wollen wir uns in diesem Abschnitt auf die zentralenen Grundkonstrukte und Besonderheiten von ML beschränken. Eine ausführlichere Beschreibung findet man in [Gordon *et.al.*, 1979], [Constable *et.al.*, 1986, Kapitel 6&9] und [Jackson, 1993a]. Um ML-Konstrukte von eventuell gleichlautenden Konstrukten der Objektsprache zu unterscheiden, werden wir sie unterstrichen darstellen.

4.1.1.1 Funktionen

Grundlage aller ML-Programme ist die Definition und Applikation von Funktionen. In ML werden diese entweder als *Abstraktion*

```
let divides = \x.\y.((x/y)*y = x);;
```

oder als definitorische Gleichung

```
let divides x y = ((x/y)*y = x);;
```

eingeführt. Beide Formen definieren die gleiche Funktion `divides`, welche eine ganze Zahl in eine Funktion von den ganzen Zahlen in Boole'sche Werte abbildet.[7] \ ist ein Abstraktionsoperator, der eine ASCII-Repräsentation des vertrauteren λ ist. Die definitorische Gleichung ist jedoch etwas allgemeiner als die Abstraktionsform, da sie auch benutzt werden kann, um rekursive Funktionen auf elegante Art einzuführen, wie in

```
letrec MIN f start = if f(start)=0 then start else MIN f (start+1).
```

Die Funktion `MIN` ist hierbei eine Funktion *höherer Ordnung*. Sie nimmt eine Funktion `f` $\in$ `int->int` als Argument und bildet sie in eine Funktion von den ganzen Zahlen in ganze Zahlen ab. In ML dürfen beliebige Funktionen als Argumente von anderen Funktionen vorkommen, solange sie typisierbar sind.

[6] Diese Sprache wurde später standardisiert und zu einer echten funktionalen Programmiersprache ausgebaut, die mittlerweile bei der Implementierung von Systemen, in denen Symbolverarbeitung eine wichtige Rolle spielt, weltweit Verbreitung gefunden hat und die zuvor dominierende Sprache LISP zu verdrängen beginnt. Die wichtigsten ML-Dialekte, die in der Praxis eingesetzt werden, sind CAML (Categorial Abstract Machine Language) [Cousineau & Huet, 1990, Mauny, 1991, Weis *et.al.*, 1990] und SML (Standard ML).

Funktionale Programmiersprachen haben gegenüber den imperativen Programmiersprachen den generellen Vorteil, daß der Programmieraufwand relativ gering ist, wenn man bereits eine präzise Beschreibung des Problems kennt. Was die Geschwindigkeit angeht, sind sie mittlerweile genauso effizient wie imperative Sprachen, solange nicht nur ständig einzelne Werte in komplexen Datenstrukturen verändert werden. In Kauf nehmen muß man allerdings einen relativ großen Speicherverbrauch, was in Anbetracht der heutigen Hardware allerdings kein Problem mehr ist.

[7] Die obige Gleichung wird vom ML-Interpreter zunächst auf ihre Typisierbarkeit überprüft. Ist eine Typisierung möglich, so wird die Funktion samt ihres Typs in die "Welt" von ML aufgenommen und es erscheint die Kontrollmeldung

```
divides = - :(int -> int -> bool)
```

Andernfalls erscheint eine Fehlermeldung und der Name `divides` gilt weiterhin als unbekannt, falls er zuvor unbekannt war.

Konstanten werden wie Funktionen durch eine definitorische Gleichung deklariert. Zwischen nullstelligen Funktionen (`let f () =` *ausdruck*) und Konstanten (`let f =` *ausdruck*) besteht jedoch ein Unterschied, da der Wert einer Konstanten zur Zeit der Deklaration berechnet wird, während ein Funktionskörper erst bei einer Applikation ausgewertet wird.[8]

4.1.1.2 Typen

In ML wird jedem Objekt, auch den Funktionen, ein *Typ* zugeordnet, zu dem es gehören soll. Dies ermöglicht es, Typeinschränkungen der Argumente und Ergebnisse von Funktionen auszudrücken und zu erzwingen. Die Basistypen von ML sind ganze Zahlen, Boole'sche Werte, Token und Strings (`int`, `bool`, `tok`, `string`) und ein einelementiger Datentyp `unit`. Token und Strings unterscheiden sich durch ihren Verwendungszweck und werden dadurch unterschieden, daß ein Token durch `‘`*token-quotes*`‘` umgeben wird. Komplexere Datentypen können hieraus durch die Typkonstruktoren `->`, `#`, `+` und `list` (Funktionenraum, Produkt, disjunkte Vereinigung und Listen) gebildet werden. Zugunsten einer automatischen Typisierbarkeit von ML-Ausdrücken sind abhängige Typkonstruktoren kein Bestandteil von ML.

Um eine höhere Flexibilität und Klarheit bei der Programmierung komplexerer Algorithmen zu errreichen, darf das Typsystem durch *anwenderdefinierte Datentypen* konservativ erweitert werden. Dies kann auf zwei Arten geschehen. Durch eine Deklaration

```
lettype intervals = int#int
```

wird einfach ein neuer Name für den Typ `int#int` eingeführt, der ab sofort als Abkürzung verwendet wird. Einen besonderen Unterschied zwischen `intervals` und `int#int` gibt es ansonsten nicht.

Darüber hinaus erlaubt ML aber auch die Deklaration *abstrakter Datentypen*, in denen die interne Darstellung der Elemente nach außen hin unsichtbar bleibt und Zugriffe nur über Funktionen möglich sind, die innerhalb der Deklaration des abstrakten Datentyps definiert wurden. Durch diese *Datenkapselung* kann man verhindern, daß Anwenderprogramme in unerwünschter Weise – zum Beispiel durch direkte Manipulation einer Komponente – auf die Daten zugreifen. Diese Eigenschaft ist besonders wichtig, wenn Systeme mit sensiblen Datenstrukturen wie Terme der Typentheorie oder Beweise programmiert werden sollen, die zugunsten einer Korrektheitsgarantie nur kontrollierte Veränderungen der Daten zulassen. Ein einfaches Beispiel für einen solchen abstrakten Datentyp ist der Datentyp `time`:

```
abstype time = int # int
 with maketime(hrs,mins) = if  hrs<0 or 23<hrs or mins<0 or 59<mins
                           then fail
                           else abs_time(hrs,mins)
 and hours t   = fst(rep_time t)
 and minutes t = snd(rep_time t);;
```

Diese Deklaration erklärt den Datentyp `time` zusammen mit drei Funktionen `maketime`, `hours` und `minutes`. Die Funktionen `abs_time` und `rep_time` sind nur innerhalb dieser Deklaration bekannt und stellen Konversionen von der expliziten zur abstrakten Repräsentation bzw. umgekehrt dar, die bei der Programmierung der mit `time` assoziierten Funktionen benötigt werden. Durch die abstrakte Deklaration wird sichergestellt, daß `time`-Objekte nur durch `maketime` verändert und nur durch `hours` und `minutes` analysiert werden können.

Typen dürfen auch *rekursiv* definiert werden, was unbedingt erforderlich ist, um Konstrukte wie Bäume, Graphen, Terme oder Beweise beschreiben zu können. Ebenso ist es möglich *generische* Datentypen zu erzeugen, also Datentypen, die einen *Typparameter* enthalten. So könnte man zum Beispiel Binärbäume über einem beliebigen Datentyp wie folgt deklarieren.

[8] Als ein Zugeständnis an die Effizienz bietet ML auch imperative Konzepte wie globale Variablen an. Diese sind explizit mit `letref` ... als solche zu deklarieren und dürfen dann Werte *zugewiesen* bekommen.

```
absrectype * bintree = * + (* bintree) # (* bintree)
  with mk_tree(s1,s2) = abs_bintree (inr(s1,s2) )
  and left s          = fst ( outr(rep_bintree s) )
  and right s         = snd ( outr(rep_bintree s) )
  and atomic s        = isl(rep_bintree s)
  and mk_atom a       = abs_bintree(inl a)
;;
```

Bei der Deklaration einer Funktion ist es normalerweise nicht erforderlich, den Datentyp der Argumente bzw. des Ergebnisses mit anzugeben, da ML jedem Term automatisch einen *Typ* zuordnet, sofern dies möglich ist. Hierzu wird eine erweiterte Form des *Typechecking Algorithmus* von Hindley und Milner (siehe Abschnitt 2.4.4 auf Seite 75 und [Hindley, 1969, Milner, 1978, Damas & Milner, 1982]) eingesetzt. Dabei kann der Typ einer Funktion auch *polymorph* sein, was bedeutet, daß in der Typisierung *Typvariablen* (üblicherweise `*, **, ***` etc.) auftreten können, für bei einer Anwendung der Funktion beliebige konrekte Datentypen eingesetzt werden dürfen. So erhält zum Beispiel die Funktion

```
\x.x
```

den Datentyp `(* -> *)`, was besagt, daß der Ergebnistyp von `\x.x` identisch mit dem Typ des Argumentes sein muß, aber sonst keinerlei Beschränkungen existieren. `\x.x` kann somit als Identitätsfunktion auf beliebigen Datentypen eingesetzt werden. Ein weiteres Beispiel ist die oben deklarierte Funktion `mk_atom`, deren Datentyp `(* -> * bintree)` polymorph[9] ist, weil sie als Bestandteil eines generischen Datentyps deklariert wurde.

4.1.1.3 Vordefinierte Operationen

Die meisten der vordefinierten ML-Funktionen verwenden Bezeichnungen, die in der Mathematik geläufig sind. Auf `int` gibt es die üblichen Operationen `+,-,*,/,<,>,=`. Boole'sche Operationen sind `not, &, or`. Paare werden durch Kommata wie in `1,2` gebildet und durch `fst` und `snd` analysiert. Für die disjunkte Vereinigung verwendet man `inl, inr, outl, outr` und `isl` und für Listen `[], null, hd, tl` sowie die Punktnotation `a.liste`, um ein Element vor eine Liste zu hängen. Eine explizite Auflistung schreibt man in eckige Klammern durch Semikolon getrennt wie in `[1;2;3;4;5]`. Klammern sind einzusetzen, wenn die Eindeutigkeit es erfordert. Über diese Grundoperationen hinaus gibt es eine große Menge weiterer vordefinierter Funktionen. Für Details verweisen wir auf [Jackson, 1993a, Kapitel 6 & 7].

4.1.1.4 Abstraktionen

Um zu vermeiden, daß komplexe Teilausdrücke mehrmals explizit in einem Term genannt und ausgewertet werden müssen, kann man abkürzende Bezeichnungen einführen, die nur lokale Gültigkeit haben.

```
let x = 2*y*y+3*y+4 in x*x
```

bedeutet zum Beispiel, daß der Wert des Teilausdrucks `2*y*y+3*y+4` nur einmal bestimmt wird und dann alle Vorkommen von `x` im Ausdruck `x*x` durch diesen Wert ersetzt werden. Dieses Konstrukt kommt häufig innerhalb von Funktionsdeklarationen vor. Dabei dürfen durchaus auch (rekursive) Funktionen als Abkürzungen eingeführt werden wie zum Beispiel in

```
let upto from to   = letrec aux from to partial_list =
                        if to < from then partial_list
                                     else aux from (to-1) (to.partial_list)
                     in
                        aux from to []
                        ;;
```

[9] Man beachte, daß der Begriff der *Polymorphie* ("vielgestaltig") in der Informatik z.T. auch eine weitergehende Bedeutung bekommen hat, der im Zusammenhang mit den Konzepten Vererbung und dynamischem Binden der objektorientierten Programmierung steht.

Diese Funktion berechnet die Liste `[from;...;to]` indem sie diese schrittweise aus einer partiellen Liste aufbaut, die mit `[]` initialisiert wird.

Eine Besonderheit von ML ist, daß auf der linken Seite von Deklarationen und Abstraktionen auch zusammengesetzte Ausdrücke stehen dürfen. ML versucht dann, die Komponenten der linken Seite gegen den Wert der rechten Seite zu *matchen*[10] und die auf der linken Seite vorkommenden Variablen entsprechend zu belegen. Der Ausdruck auf der linken Seite darf aus Variablen, einer Dummy-Variablen `()`, und den *Konstruktoren* für Tupel (`,`) und Listen (`.` bzw. `[ ; ; ; ]`) aufgebaut sein. Dies erspart die Verwendung von Destruktoren wie `fst, snd, hd, tl` und ermöglicht sehr elegante Deklarationen wie zum Beispiel

```
let x.y.rest = upto 1 5 in x,y.
```

Hier wird `x` mit `1` und `y` mit `2` belegt und das Paar `1,2` zurückgegeben.

4.1.1.5 Ausnahmen

ML besitzt einen wohldurchdachten Mechanismus zur Behandlung von *Ausnahmesituationen* (*exceptions*). Einige Funktionen wie zum Beispiel die Division `/` oder die Funktion `hd` liefern bei der Eingabe bestimmter Argumente einen Laufzeitfehler (*failure*), da sie hierfür nicht sinnvoll definiert werden können. Ebenso können beim Matching in Deklarationen Fehler entstehen – zum Beispiel, wenn `let [x;y] = L in ...` ausgewertet werden soll, aber `L` die leere Liste ist. ML bietet nun die Möglichkeit an, derartige Ausnahmesituationen abzufangen (*failure catching*) und zu einem wohldefinierten Ende zu bringen. Auf diese Art kann ein unkontrollierter Abbruch des Programms innerhalb dessen der Fehler auftrat, vermieden werden.

Hierzu steht es ein spezieller Operator `?` zur Verfügung, der folgenden Effekt hat: ein Ausdruck e_1 `?` e_2 liefert als Ergebnis das Resultat der Auswertung von e_1, sofern diese keine Ausnahme erzeugt, und ansonsten das Ergebnis der Auswertung von e_2. So liefert zum Beispiel

```
2/2 ? 1000
```

den Wert `1`, während

```
2/0 ? 1000
```

den Wert `1000` liefert. Diesen Mechanismus kann man sich immer dann zunutze machen, wenn in einer Funktion eine andere Funktion benutzt wird, die einen Fehler erzeugen könnte. Durch die Deklaration

```
let divides x y = ((x/y)*y = x) ? false;;
```

wird zum Beispiel vermieden, daß `divides x 0` zu einem Fehler führt. Stattdessen wird das gewünschte Ergebnis `false` zurückgegeben.

Es ist auch möglich, Ausnahmen mit Hilfe des Ausdrucks `fail` gezielt zu erzeugen, um ein unerwünschtes Verhalten – besonders in rekursiven Funktionen – gezielt beenden zu können. Dadurch erspart man es sich, die Auswertung der Funktion zuende laufen lassen zu müssen und dabei ständig eine Fehlermeldung mitzuführen, die dann am Ende ausgegeben werden kann.

Der Ausnahmebehandlungsmechanismus hat gegenüber anderen Fehlerbehandlungsmöglichkeiten den Vorteil, daß man nicht von Anfang an alle Eingaben abfangen muß, die *möglicherweise* einen Fehler erzeugen. Er ist – sorgfältig eingesetzt – die effizienteste und eleganteste Art der Fehlerbehandlung, kann allerdings auch zu einem undurchsichtigen Programmierstil mißbraucht werden.

4.1.2 Implementierung der Objektsprache

Die Entwicklung der Programmiersprache ML als Formalisierung einer Metasprache, die bei der Beschreibung formaler Kalküle benutzt wurde, ermöglicht es, die Implementierung der objektsprachlichen Konzepte

[10] Im Deutschen gibt es hierfür kein einheitlich anerkanntes Wort. Manchmal wird der Begriff *mustern* verwendet.

unmittelbar an die in Abschnitt 3.2 gegebenen Definitionen von Termen, Sequenzen, Regeln und Beweisen anzulehnen. Wir müssen hierzu nur die informalen Definitionen in abstrakte Datentypen übertragen und dabei Funktionen für einen Zugriff auf Elemente dieser Datentypen einführen. Da wir in diesen Definitionen bereits eine strikte Trennung zwischen der allgemeinen Struktur von Termen und Regeln und den konkreten Bestandteilen der Typentheorie vorgenommen haben, erhalten wir eine sehr flexible und leicht zu wartende Implementierung, die auch für eine Realisierung anderer formaler Theorien verwendbar ist. Die Implementierung einer konkreten Theorie kann dann durch Einträge in separaten Tabellen (bzw. durch Objekte der Bibliothek) durchgeführt werden.

4.1.2.1 Terme

Der Datentyp `term` ist ein rekursiver abstrakter Datentyp, der die Definition 3.2.5 auf Seite 104 wiederspiegeln soll. Hierzu müssen wir Terme und gebundene Terme simultan definieren. Man beachte, daß sich die Darstellungsform von Termen und gebundenen Termen von ihrer internen Repräsentation unterscheidet.

```
abstype  var  = tok
  with mkvar t = abs_var t
  and  dvar  v = rep_var v
;;
abstype  level_exp  = tok + int
  with mk_var_level_exp t     = abs_level_exp (inl t)
  and  mk_const_level_exp i   = abs_level_exp (inr i)
  and  dest_var_level_exp l   = outl (rep_level_exp l)
  and  dest_const_level_exp l = outr (rep_level_exp l)
  and    ⋮
;;
abstype  parm = int + tok + string + var + level_exp + bool
  with mk_int_parm i      = abs_parm (inl i)
  and  mk_tok_parm t      = abs_parm (inl (inr t))
  and  mk_string_parm s   = abs_parm (inl (inr (inr s)))
  and  mk_var_parm v      = abs_parm (inl (inr (inr (inr v))))
  and  mk_level_parm l    = abs_parm (inl (inr (inr (inr (inr l)))))
  and  mk_bool_parm b     = abs_parm (inr (inr (inr (inr (inr b)))))
  and  dest_int_parm p    = outl (rep_parm p)
  and  dest_tok_parm p    = outl (outr (rep_parm p))
  and  dest_string_parm p = outl (outr (outr (rep_parm p)))
  and  dest_var_parm p    = outl (outr (outr (outr (rep_parm p))))
  and  dest_level_parm p  = outl (outr (outr (outr (outr (rep_parm p)))))
  and  dest_bool_parm p   = outr (outr (outr (outr (outr (rep_parm p)))))
;;
absrectype  term = (tok # parm list) # bterm list
and         bterm = var list # term
  with mk_term (opid,parms) bterms = abs_term((opid,parms),bterms)
  and  dest_term t                 = rep_term t
  and  mk_bterm vars t             = abs_bterm(vars,t)
  and  dest_bterm bt               = rep_bterm bt
;;
```

Im abstrakten Datentyp `parm` sind die verschiedene Parametertypen aus Abbildung 3.1 (Seite 104) direkt repräsentiert. Zur Bildung von Level Expressions gibt es noch weitere Möglichkeiten als die hier direkt angegebenen. Terme werden gebildet, indem man ihren Operatornamen, ihre Parameterliste und ihre Teilterme angibt. So wird zum Beispiel der Term **U**{1:l}() erzeugt durch:

```
mk_term (‘universe‘,[mk_level_parm (mk_const_level_exp 1)]) []
```
[11]

[11] Es sei angemerkt, daß in NuPRL 4.0 für alle Terme der Typentheorie bereits spezialisierte Funktionen wie `mk_universe_term`, `mk_function_term` etc. vordefiniert sind, welche Ausdrücke der obigen Art abkürzen.

Die Mechanismen zur Darstellung von NuPRL-Termen, die wir in Abschnitt 4.1.7.2 kurz ansprechen werden, sorgen dafür, daß dieser Term normalerweise das Erscheinungsbild U_1 hat, sofern nicht explizit etwas anderes gefordert wird.

4.1.2.2 Regeln und Beweise

Beweise werden gemäß Definition 3.2.27 auf Seite 120 als Bäume dargestellt, deren Knoten aus Sequenzen und Beweisregeln bestehen. Unvollständige Beweise enthalten Blätter, die nur aus einer Sequenz bestehen. Eine Sequenz wiederum besteht aus einer Liste von Deklarationen und einer Konklusion (ein Term), wobei Deklarationen aus Variablen und Termen aufgebaut sind. Die Formulierung der entsprechenden Datentypen und Zugriffsfunktionen ist verhältnismäßig naheliegend, zumal die Definition der Beweise bereits in rekursiver Form vorliegt.

Durch eine abstrakte Definition des Datentyps proof kann jede unbefugte Manipulation von Beweisen unterbunden und somit die gewünschte Sicherheit des gesamten Beweisentwicklungssystems garantiert werden. Auf die Komponenten eines Beweises kann nur durch Selektorfunktionen `hypotheses`, `conclusion`, `refinement` und `children` zugegriffen werden. Veränderungen eines Beweises sind nur durch Erzeugung eines unbewiesenen Beweisziels mittels `mk_proof_goal` und durch Anwendung einer Regel auf einen Beweisknoten mit Hilfe der Funktion `refine` möglich.[12]

Die Regeln, mit denen Beweise manipuliert werden dürfen, repräsentieren die konkrete formale Theorie, welche durch das Beweissystem verarbeitet werden kann. In Definition 3.2.26 auf Seite 119 hatten wir definiert, daß eine Regel eine Sequenz – also einen unvollständigen Beweis – in eine Liste von Teilbeweisen abbildet und als Validierung angibt, wie die Extraktterme der Teilziele zu einem Extraktterm der Originalsequenz zusammenzusetzen sind. Im Kontext formaler Beweise muß die Rolle der Validierung jedoch etwas abstrakter gesehen werden sein, als nur einen Extraktterm zu generieren. Sie soll, wie der Name bereits andeutet, eine Evidenz konstruieren, *warum* die ursprüngliche Sequenz ein gültiges Urteil repräsentiert, wenn dies für die Teilziele gilt. Mit anderen Worten, sie soll beliebige Beweise der Teilziele – seien sie nun vollständig oder unvollständig – in einen Beweis des ursprünglichen Ziels zusammensetzen können. Das bedeutet, daß der tatsächliche Beweis*baum* durch die Validierung aufgebaut wird und nicht etwa durch die Regel selbst. Die Konstruktion des Extraktterms ist implizit in der Validierung enthalten, da dieser im wesentlichen als eine Term-Darstellung des konstruierten Beweises betrachtet werden kann.

Wozu ist dieser zusätzliche Aufwand nun nötig? Man könnte sicherlich darauf verzichten, wenn man Beweise ausschließlich mit Hilfe der elementaren Regeln des Kalküls konstruieren will, da diese alle Informationen enthalten, welche für die Dekomposition eines Beweiszieles und die Konstruktion eines Extrakttermes als Evidenz nötig sind. Dies reicht jedoch nicht mehr aus, wenn man mehrere Regeln zu einer Beweisregel zusammensetzen oder Beweise durch Meta-Programme von außen steuern will. In diesem Falle muß man nämlich berechnen können, welche unbewiesenen Teilziele übrigbleiben (das ist einfach) und welche Evidenz aus den Evidenzen der übriggebliebenen Teilziele entstehen soll. Letzteres aber würde bedeuten, objektsprachliche Terme zusammenzusetzen und hierzu auch auf Informationen aus Zwischenzielen zuzugreifen, die nicht mehr übrigbleiben. Im Endeffekt ist dies dasselbe wie einen Beweis aus einer Liste von Teilbeweisen zusammenzusetzen. Es ist somit einfacher und natürlicher, Validierungen als Funktionen von `proof list` nach `proof` zu beschreiben und ihnen auch die tatsächliche Erzeugung der Beweisknoten zu überlassen.

Diese Sichtweise auf Validierungen, die erstmalig im Rahmen des LCF-Konzepts der *Beweistaktiken* entstanden ist [Gordon *et.al.*, 1979] und in Abschnitt 4.2 vertieft wird, führt dazu, daß Regeln als spezielle Instanz von Beweistaktiken betrachtet werden, in die sie mit Hilfe der Funktion `refine` umgewandelt werden. Taktiken wiederum sind Funktionen, welche einen Beweis in eine Liste von Teilbeweisen und eine Validierung abbilden. Die Anwendung der Validierung auf die Liste der erzeugten Teilbeweise generiert schließlich den neuen

[12] Normalerweise müßte hierzu neben dem Namen der Regel auch die Position des zu modifizierenden Knotens im Beweis angegeben werden. In der NuPRL-Implementierung kann hierauf verzichtet werden, da der Beweiseditor (siehe Abschnitt 4.1.6) interaktive Bewegungen im Beweis und somit auch lokale Veränderungen von Beweisknoten ermöglicht.

Beweisknoten.[13] Diese Betrachtungen führen zu der folgenden Repräsentation von Beweisen durch abstrakte ML-Datentypen.

```
abstype  declaration = var # term
  with mk_assumption v t = abs_declaration(v,t)
  and  dest_assumption d = rep_declaration d
;;
lettype  sequent = declaration list # term;;
abstype  rule    = .....

absrectype  proof = (declaration list # term) # rule # proof list
  with mk_proof_goal decs t = abs_proof((decs,t), ◇,[])
  and  refine r p   = let children  = deduce_children r p
                      and validation= deduce_validation r p
                      in
                         children, validation
  and  hypotheses p = fst (fst (rep_proof p))
  and  conclusion p = snd (fst (rep_proof p))
  and  refinement p = fst (snd (rep_proof p))
  and  children   p = snd (snd (rep_proof p))
;;
lettype  validation = proof list -> proof;;
lettype  tactic     = proof -> (proof list # validation);;
```

In dieser Darstellung ist ◇ eine Abkürzung für eine interne "Nullregel", die keinerlei Aktionen auslöst und nur als Platzhalter dient. Die genaue Struktur der Regeln, auf die wir hier nicht im Detail eingehen wollen, enthält eine schematische (bei komplexeren Regeln wie `arith` auch eine algorithmische) Beschreibung, wie Teilziele und Validierungen erzeugt werden sollen. Die Umwandlung dieser Beschreibungen in eine Liste von Beweisen und eine Validierung geschieht innerhalb der Funktion `refine` mit Hilfe der internen Funktionen `deduce_children` und `deduce_validation`. Dabei sind diese Funktionen so ausgelegt, daß Anwendung dieser Validierung auf die Liste der Teilbeweise einen Beweis der Gestalt

```
abs_proof( (hypotheses p, conclusion p), r, deduce_children r p)
```

generiert. Teilbeweise, die vor Anwendung der Regel in `p` enthalten waren, werden somit überschrieben. Die Funktion `refine` erzeugt eine Ausnahmesituation, wenn die Regel nicht anwendbar ist. Diese Ausnahme kann vom Beweiseditor aufgefangen werden und in eine Fehlermeldung umgewandelt werden. Für weitere Details verweisen wir auf [Constable *et.al.*, 1986, Kapitel 9.2].

Insgesamt hängt die Korrektheit eines maschinell geführten Beweises also nur von einer korrekten Repräsentation der theoretisch vorgegebenen Regeln und einer fehlerfreien Implementierung der Funktion `refine` ab. Alle anderen Komponenten des Systems beeinflussen die Eleganz des Umgangs mit dem System, haben aber keinen Einfluß auf seine Sicherheit.

4.1.3 Bibliothekskonzepte

Die Bibliothek eines Beweisentwicklungssystems ist das formale Gegenstück zu einem mathematischen Lehrbuch, in dem alle Definitionen, Sätze, Beweise, Methoden und Anmerkungen zu einem bestimmten Gebiet in linearer Reihenfolge gesammelt werden. Sie besteht aus *Objekten*, welche Terme, Beweise oder Definitionen

[13] In NuPRL wird diese Anwendung der Validierung auf die Teilziele automatisch vom Beweiseditor ausgelöst. In Systemen ohne derartige Interaktionsmöglichkeiten würde die Beweiskonstruktion erheblich komplizierter und undurchsichtiger. So wurden zum Beispiel in LCF zuerst alle Regeln zu *einer* Taktik zusammengesetzt und dann geschlossen auf den Beweis angewandt.

Es sei angemerkt, daß alle in diesem Skript angegebenen Regeln im NuPRL-System bereits als Taktiken repräsentiert sind, welche aus den tatsächlichen, meist gleichnamigen Regeln durch die Funktion `refine` (und einen Mechanismus zur Generierung der notwendigen Variablennamen) entstanden sind. Dies erspart weitere Konversionen, wenn mehrere Regeln zu einer aufwendigeren Taktik zusammengesetzt (Siehe Abschnitt 4.2.4) werden sollen.

der konkreten formalen Theorie oder auch allgemeine mathematische Methoden (also Taktiken) oder Texte (Kommentare) enthalten. Zu jedem dieser Objekte gehört ein Name, eine Bezeichnung der Art des Objektes, sein Status (*vollständig, unvollständig, fehlerhaft, leer* – gekennzeichnet durch `*,#,-` und `?`) und sein Position in der Bibliothek.

Die einfachste Art, derartige Bibliotheken zu repräsentieren ist eine lineare Liste, wobei man aus Effizienzgründen einen Mechanismus für einen schnellen Zugriff über den Namen eines Objektes hinzufügen sollte. Die wichtigsten Operationen auf einer Bibliothek sind

- Erzeugen (`create`) eines neuen (leeren) Objektes durch Angabe eines Namens, der Art (`thm, abs, disp, ml, ....`) und einer Position in der Bibliothek (dem Namen des Objektes, *vor* dem es erscheinen soll) – jeweils als String.
- Löschen (`delete`) eines Objektes durch Angabe seines Namens.
- Editieren (`view`) eines Objektes durch Angabe seines Namens. Hierdurch wird der zum Objekt passende Editor aufgerufen, also ein Beweiseditor für Theoreme und ein Text-/Termeditor in allen anderen Fällen.

Darüber hinaus gibt es eine Reihe anderer nützlicher Operationen wie das Verschieben oder Umbenennen eines Objektes, eine Überprüfung des Inhaltes, das Laden von Teilbibliotheken von einer Datei, das Ablegen einer Reihe von Objekten in einer Datei, die Aufbereitung einer Reihe von Objekten für eine textliche Darstellung (z.B. in LaTeX), der Zugriff auf einzelne Komponenten eines Objektes (wie den in einem Theorem-Objekt enthaltenen Beweis oder Extrakt-Term) usw. In NuPRL ist die Bibliothek, auf die sich alle aktuellen Kommandos beziehen, als globale Variable vom Typ `library` deklariert. Dies Bibliothek wird ständig in einem speziellen Library-Fenster angezeigt und kann mit speziellen Befehlen (und Mausoperationen) durchgeblättert werden. Andere Bibliotheken können im Hintergrund gesichert und bei Bedarf zur aktuellen Bibliothek erklärt werden.

Die Existenz einer Bibliothek macht auch die in Abbildung 3.28 auf Seite 175 angegebenen Inferenzregeln `lemma` und `extract` zu einem sinnvollen Bestandteil des Inferenzsystems. Aus theoretischer Sicht kann man die Menge aller bewiesenen Theoreme einer Bibliothek als zusätzliche Hypothesen einer Beweissequenz betrachten, auf die man über den Namen des Theorems zugreifen kann. Damit sind diese beiden Regeln nichts anderes als eine besondere Form der Regeln `hypothesis` und `hypEq`, die unmittelbar auf den Hypothesen arbeiten.

Da die Darstellung einer Bibliothek als lineare Liste von Objekten eigentlich zu wenig Struktur enthält, um ein echtes Gegenstück zu einem Buch zu sein, welches aus Kapiteln, Unterkapiteln und Abschnitten besteht, gibt es in NuPRL einen simplen Mechanismus, eine Bibliothek in *Theorien* zu unterteilen. Dies geschieht durch Einfügen spezieller Kommentarobjekte, welche den Anfang und das Ende einer Theorie kennzeichnen sowie durch die Verwendung von Tabellen (globale ML-Variablen), in denen die Abhängigkeiten der Theorien untereinander und die zu einer Theorie assoziierten Filenamen enthalten sind. Die speziellen Details dieses Mechanismus sowie die wichtigsten Operationen zur Manipulation einer Bibliothek sind in [Jackson, 1993b, Kapitel 3] zu finden.

4.1.4 Die Kommandoebene

Die Kommandoebene stellt das zentrale Interface zwischen einem Beweisentwicklungssystem und seinem Benutzer dar. Sie dient dazu, das Bibliotheks-Fenster zu kontrollieren, Theorien und Teilbibliotheken zu laden und abzulegen, Editoren für Objekte der Bibliothek zu starten, mit Funktionen der Metasprache und Termen der Objektsprache zu experimentieren und externe ML-Dateien in das System hineinzuladen.

In den meisten Beweisentwicklungssystemen ist die Kommandoebene identisch mit einem Interpreter der Metasprache, mit dem die anstehenden Aufgaben in eleganter und unkomplizierter Weise gesteuert werden können, und läuft in einer Shell des Betriebssystems ab. In NuPRL hat man sich zugunsten einer Möglichkeit,

sichtbar mit Termen der Objektsprache zu experimentieren, dazu entschieden, die Kommandoebene in ein spezielles *Term-Editor* Fenster zu integrieren, innerhalb dessen sowohl Texte der Metasprache (die durch den ML-Parser kontrolliert werden) als auch Terme der Objektsprache in ihrer Display-Form editiert werden können.[14]

4.1.5 Der Text- und Termeditor

Die Verwendung verständlicher Notation innerhalb eines formalen Systems zur "Implementierung" mathematischer Theorien ist ein aktives Forschungsgebiet seit der Entwicklung der ersten Programmiersprachen. Das wesentliche Problem ist hierbei, daß einerseits zugunsten einer eleganten Handhabung durch menschliche Benutzer eine möglichst freie Syntax zur Verfügung stehen muß, andererseits die formale Sprache aber auch durch einen Computer decodierbar bleiben muß, wobei die Zeit zur Wiedererkennung objektsprachlicher Ausdrücke im Verhältnis zu den eigentlich durchzuführenden Berechnungen sehr gering sein muß.

Aus der Theorie der formalen Sprachen ist bekannt, daß eine Sprache im wesentlichen *kontextfrei* sein muß, um effizient mit Hilfe von Parsern decodiert werden zu können. Während man sich bei Programmiersprachen mittlerweile an derartige Einschränkungen in der Freiheit der Ausdrucksweise gewöhnt hat, ist dies zur Darstellung mathematischer Konzepte völlig unakzeptabel. Das Verständnis mathematischer Texte hängt zu einem Großteil von einer klaren und leicht zu merkenden Notation ab, die normalerweise viel zu komplex ist, um durch ASCII Text oder gar eine kontextfreie Syntax beschrieben werden zu können.

Beweisentwicklungssysteme, bei denen eine Interaktion mit einem Benutzer vorgesehen ist – und sei es nur für die Eingabe des Problems selbst – sind darauf angewiesen, einen Großteil der mathematischen Notation auch auf dem Bildschirm wiedergeben zu können. Aus diesem Grunde gibt es Bemühungen, durch *ausgefeiltere Parser* die Syntax formaler Sprachen flexibler zu gestalten. Dieser Ansatz ist jedoch problematisch, da die mathematische Notation voller Zweideutigkeiten steckt, die nur aus dem allgemeinen Kontext heraus richtig interpretiert werden können. So kann zum Beispiel die Juxtaposition `x y` einmal als Multiplikation zweier Zahlen oder als Applikation der Funktion `x` auf das Argument `y` interpretiert werden. Um derartige Zweideutigkeiten (also "*overloading*" in der Denkweise der Programmiersprachen) aufzulösen, muß man den Parser mit einem Type-Checker koppeln, was bei einer reichhaltigen Typstruktur sehr ineffizient werden kann.

Wesentlich effizienterer und langfristig auch vielseitiger als eine ständige Erweiterung eines Parsers ist es, die Eingabe von Termen der Objektsprache durch einen *Strukturediteor* zu kontrollieren, der in der Lage ist, die Baumstruktur eines Termes direkt zu generieren, auf dem Bildschirm aber die Darstellungsform der Terme zu präsentieren. Der Vorteil dieser Vorgehensweise ist, daß

- überhaupt kein Parser mehr erforderlich ist, weil die Baumstruktur nach dem Editieren vorliegt,
- Mehrdeutigkeiten nicht mehr beachtet werden müssen, weil diese nur in der textlichen Präsentation, nicht aber intern vorkommen und auf Wunsch leicht aufgelöst werden können (man lasse sich die interne Form zeigen),
- keinerlei Beschränkung für die textliche Darstellungsform besteht – man kann alle Möglichkeiten von Textverarbeitungssystemen wie LaTeX ausschöpfen,
- ein einheitlicher Wechsel der Notation extrem einfach wird,
- Formatierung sich direkt an dem zur Verfügung stehenden Platz orientiert, und
- der Benutzer die genaue Syntax nicht kennen muß sondern nur den Namen des darzustellenden Terms.

Der größte Nachteil solcher Strukturediteoren ist, daß sie ein Umdenken erfordern, wenn man gewohnt ist, die Syntax (wie in Emacs) direkt einzugeben, und daß sie bisher noch nicht so ausgereift sind wie gewöhnliche

[14] Beispiele für die Verarbeitung von Kommandos in diesem "ML Top Loop" findet man in [Jackson, 1993b, Kapitel 2&3]

Texteditoren. Dieser Nachteil ist jedoch akzeptabel, wenn man bedenkt, daß Strukturediteren bisher der einzige Weg sind, die flexible Notation mathematischer Textbücher auf formale Beweissysteme zu übertragen.

In den Texteditor des NuPRL Systems ist deshalb ein strukturierter Termeditor integriert, welcher durch ein spezielles Kommando (CONTROL-0) in einem bestimmten Bereich der Eingabe aktiviert werden kann. Dieser integrierte Text- und Termeditor steht im ML-Top Loop und beim Editieren aller Objekte mit Ausnahme der Beweisbäume zur Verfügung. Wir wollen die typische Arbeitsweise an einem Beispiel illustrieren.

Beispiel 4.1.1

Bei der Erzeugung eines Beweisziels mit dem Beweiseditor (siehe Abschnitt 4.1.6) ist der Editor automatisch im Term-Modus. Wenn wir nun einen existentiell quantifizierten Term eingeben wollen, so müssen wir nur den Namen dieses Terms, also `exists` eintippen und die RETURN taste betätigen. Danach erscheint im Display

∃[var]:[type]. [prop]

und der Cursor steht im var-Feld. Die Bezeichner `[var], [type]` und `[prop]` dienen als Platzhalter für ein Eingabefeld und deuten an, welche Art von Information eingegeben werden soll. Sie verschwinden, sobald ein Stück Text und RETURN eingegeben wurde. Dabei wird wieder überprüft, ob der Text Name eines Terms ist, sofern wir in einem *Term-Slot* sind. Nach Eingabe von `x` RETURN haben wir

∃x:[type]. [prop]

und der Cursor steht im type-Feld. Wir geben nat RETURN ein, was dazu führt, daß der Term IN eingetragen wird, und anschließend eqi und erhalten

∃x:IN. [int]=[int]

wobei der Cursor im ersten int-Feld steht. Nach Eingabe von `x` RETURN und `4` RETURN haben wir als Endergebnis

∃x:IN. x=4.

Will man diesen Term verändern, so braucht man nur mit dem Cursor über den gewünschten Teilterm zu fahren und diesen zu ändern. Man beachte jedoch, daß nur die Inhalte der Slots oder der gesamte (Teil-)Term verändert werden können.

Erfahrungsgemäß dauert es nicht lange, bis man sich an die Vorteile eines Strukturediteors gewöhnt hat und Bewegungen in Texten und Termbäumen nicht mehr miteinander verwechselt. Details über diesen Editor – vor allem die Befehle und Tastenbelegungen – findet man in [Jackson, 1993b, Kapitel 4].

4.1.6 Der Beweiseditor

Im Prinzip reichen Texteditor und Kommandoebene für das Arbeiten mit einem Beweisentwicklungssystem völlig aus. Es ist durchaus möglich, Beweise durch ML Kommandos zu manipulieren und sich das Ergebnis wieder anzeigen zu lassen. Dennoch ist diese Vorgehensweise äußerst unpraktisch, wenn man Beweise interaktiv entwickeln möchte. Man müßte ständig die unbearbeiteten Beweisknoten eruieren und ihren Inhalt ansehen, bevor man weiterarbeiten kann. Wesentlich sinnvoller ist daher, die Manipulation von Beweisen durch einen speziellen Beweiseditor zu unterstützen, mit dem man sich quasi graphisch durch den Beweisbaum bewegen und Knoten ansehen und modifizieren kann. Dabei beschränken sich die Manipulationsmöglichkeiten natürlich auf die Eingabe des initialen Beweisziels und der Regeln, mit denen man die Sequenz eines Knotens verfeinern möchte, sowie auf Bewegungen innerhalb des Beweisbauems. Abgesehen davon, daß man Beweisknoten verständlich darstellen muß, ist ein solcher Beweis- oder Verfeinerungseditor eine einfache Programmieraufgabe und daher der Steuerung von Beweisen durch Kommandos vorzuziehen.

Der Beweiseditor des NuPRL Systems, der aufgerufen wird, wann immer ein Theorem-Objekt mit `view` betrachtet wird, arbeitet knotenorientiert. Man sieht also nie den gesamten Beweisbaum, sondern nur einen

Name des Theorems	EDIT THM intsqrt
Status, Position relativ zur Wurzel	# top 1
Erste Hypothese des Beweisziels	1. x:ℕ
Konklusion	⊢ ∃y:ℕ. $y^2 \leq x \wedge x < (y+1)^2$
Regel	BY natE 1
Erstes Teilziel – Status, Konklusion	1# ⊢ ∃y:ℕ. $y^2 \leq 0 \wedge 0 < (y+1)^2$
Zweites Teilziel – Status, neue Hypothesen	2# 2. n:ℕ 3. 0<n 4. v: ∃y:ℕ. $y^2 \leq n-1 \wedge n-1 < (y+1)^2$
Konklusion	⊢ ∃y:ℕ. $y^2 \leq n \wedge n < (y+1)^2$

Abbildung 4.1: Darstellung eines Beweisknotens im Editorfenster

speziellen Knoten, dessen Position relativ zur Wurzel des Beweises im Editorfenster angezeigt wird. Angezeigt werden außerdem das aktuelle Beweisziel und – sofern vorhanden – die angewandte Regel sowie die erzeugten Unterziele. Abbildung 4.1 zeigt ein typisches Beispiel für die Darstellung eines Beweisknotens.

Beim Aufruf befindet man sich im Wurzelknoten und kann sich mit Hilfe von Maus und speziellen Tastenkombinationen durch den Beweis bewegen.[15] Die lokale Sicht auf den Beweis unterstützt das Arbeiten mit dem Sequenzenkalkül, dessen großer Vorteil ja gerade die lokale Behandlung von Beweiszielen ist. Durch den Editor wird sichergestellt, daß nur das Beweisziel des Wurzelknotens und die jeweiligen Regeln des Beweises verändert werden können. Alle anderen Veränderungen werden (durch Aufruf der Funktion `refine`) automatisch bestimmt. Wir wollen die typische Arbeitsweise des Beweiseditors an einem Beispiel erläutern.

Beispiel 4.1.2

Um ein Programm zur Berechnung von Integerquadratwurzeln (vergleiche Beispiel 3.4.9 auf Seite 154) mit Hilfe eines formalen NuPRL-Beweises zu generieren, erzeugen wir zunächst mit `create` ein geeignetes Theorem-Objekt und rufen dann mit `view` den Editor auf. Es erscheint das folgende Fenster

```
EDIT THM intsqrt
 ? top
 <main proof goal>
```

Nun wird (z.B. mit der Maus) das Feld des Beweiszieles selektiert und somit der Termeditor aufgerufen. Mit diesem Editor erzeugt man das Beweisziel und schließt es dann wieder. Dabei wird das Beweisziel auf syntaktische Korrektheit überprüft und in das entsprechende Feld übernommen.[16] Der Status, der vorher *leer* (“?”) war, wird in *unvollständig* (“#”) verändert. Das Editorfenster zeigt nun folgende Inhalte an.

```
EDIT THM intsqrt
 # top
 ⊢ ∀x:ℕ.∃y:ℕ. y^2≤x ∧ x<(y+1)^2

 BY <refinement rule>
```

Um dieses Theorem nun zu beweisen, müssen wir für das Regel-Feld des Beweisknotens mit Hilfe des Termeditors eine Regel (oder eine Taktik) angeben. In diesem speziellen Fall wird die Regel `all_i` im Termeditor angegeben und dieser wieder geschlossen. Dies löst die folgenden internen Schritte aus.

1. Eine globale Variable `prlgoal` vom Typ `proof` wird mit der aktuellen Beweissequenz, also mit ∀x:ℕ.∃y:ℕ. $y^2 \leq x \wedge x < (y+1)^2$ assoziiert. Bisher bestehende Teilziele und Regeln eines eventuell schon vorhandenen Teilbaumes werden ignoriert.

[15] Es gibt Überlegungen, durch eine separate graphische Darstellung des Beweisbaums die Bewegungen zu beschleunigen und unbewiesene Teilziele leichter überschaubar zu machen.

[16] Ein eventuell vorher vorhandenes Beweisziel und frühere Regeln gehen hierdurch verloren.

2. Die Regel `all_i` und wird zusammen mit ihren (in diesem Fall nicht vorhandenen) Argumenten mit Hilfe von `refine` in eine Taktik umgewandelt und auf die Variable `prlgoal` angewandt, was zu einer (möglicherweise leeren, hier einelementigen) Liste von Teilzielen und einer Validierung führt.
3. Die Validierung wird auf die Teilziele angewandt und erzeugt einen Beweisbaum der Tiefe 1.
4. Der Beweisbaum wird in den Beweis des aktuellen Theorems integriert, wobei insbesondere der Name der Regel in das Regelfeld übernommen wird.
5. Der Inhalt des Beweiseditorfensters (also Status und Teilzieldarstellung) wird neu berechnet und angezeigt (ist die Regel nicht anwendbar, so wird der Status auf *fehlerhaft* gesetzt und eine Fehlermeldung ausgegeben).

Nach Eingabe von `all_i` erhalten wir somit die folgende Darstellung des resultierenden Beweisknotens.

```
EDIT THM intsqrt
 # top
 ⊢ ∀x:ℕ.∃y:ℕ. y²≤x ∧ x<(y+1)²

 BY all_i

 1# 1. x:ℕ
    ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
```

Um nun den nächsten Schritt ausführen zu können, müssen wir uns in den nächsttieferen Beweisknoten bewegen, indem wir das Feld des ersten (und einzigen) Teilziels selektieren. Wir erhalten

```
EDIT THM intsqrt
 # top 1
 1# 1. x:ℕ
    ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²

 BY <refinement rule>
```

und können nun schrittweise und auf ähnliche Art wie zuvor den Beweis zuende führen.

Es sei angemerkt, daß sich auch bei einer interaktiven Unterstützung eines formalen Beweises eine gewisse Vorausplanung des Beweisgangs empfiehlt, da das System einem nur die lästige Schreibarbeit und die Korrektheitsüberprüfung, nicht aber die Beweisideen abnehmen kann. Eine genaue Beschreibung des aktuellen Beweiseditors von NuPRL findet man in [Jackson, 1993b, Kapitel 7]. Verbesserungsvorschläge werden gerne entgegengenommen.

Extraktionsmechanismus

In den Beweiseditor integriert ist ein Extraktionsmechanismus, mit dem die implizit in einem Beweis enthaltenen Programme extrahiert werden können, sobald der Beweis vollständig vorliegt. Dabei werden die Extraktterme der einzelnen im Beweis enthaltenen Regeln schrittweise gemäß der rekursiven Definition auf Seite 120 zusammengesetzt. Der gesamte Extraktterm wird zusammen mit dem Beweis im Theorem-Objekt abgelegt. Im ML Top Loop kann hierauf dann mittels `extract_of_thm_object` *name* zugegriffen werden.

Der Beweiseditor und der zugehörige Programmgenerator basieren auf den allgemeinen Grundkonzepten des *Cornell Program Synthesizer Generator* [Teitelbaum & Reps, 1981, Reps & Teitelbaum, 1984] und speziellen Extraktionsmechanismen, die in [Bates, 1981, Sasaki, 1986, Stansifer, 1985] dokumentiert sind.

4.1.7 Definitionsmechanismus

Bereits in Abschnitt 2.1.5 auf Seite 14 haben wir über die Notwendigkeit gesprochen, einen formalen Kalkül durch definitorische Abkürzungen konservativ erweitern zu können. Definitorische Abkürzungen sind ein essentieller Bestandteil aller mathematischen Theorien, denn sie erlauben es, lange und komplexe Ausdrücke durch

eine kurze, prägnante Notation zu ersetzen und somit die Darstellung der Aussagen der Theorie verständlich zu halten. Es ist offensichtlich, daß ein gutes Beweisentwicklungssystem diese Vorgehensweise unterstützen muß, also einen *Definitionsmechanismus* benötigt, welcher erlaubt, konservative Erweiterungen der zugrundeliegenden Theorie auf elegante Weise einzuführen.

Ein sehr einfacher Mechanismus (der auch in früheren Versionen von NuPRL benutzt wurde) ist die Verwendung von *Textmacros*, bei denen ein langer formaler Text auf dem Bildschirm durch ein abkürzendes Macro repräsentiert wird. Dies hat aber den Nachteil, daß das System zwischen dem abkürzenden Text und der ausführlichen Form nicht unterscheiden kann. Der Gedankengang der Abstraktion, also die Einführung eines neuen Begriffs, wird dadurch nicht adäquat erfaßt. Sinnvoller ist daher, einen *Abstraktionsmechanismus* zu entwickeln, mit dem neue Terme der formalen Sprache deklariert werden, die nur durch Auffalten (die `fold`-Regel) in den Ausdruck überführt werden können, den sie abkürzen.[17] Entsprechend unserem Prinzip 3.2.4 auf Seite 103 wollen wir hierbei die einheitliche Term-Syntax verwenden und die Darstellung dieser Terme auf dem Bildschirm separat definieren.

4.1.7.1 Abstraktion

Innerhalb eines Beweisentwicklungssystems werden konservative Erweiterungen der Theorie durch Abstraktionsobjekte der Bibliothek eigeführt. Diese enthalten Definitionen der Form

$$lhs \texttt{ == } rhs,$$

wobei *lhs* den neu deklarierten Term beschreibt und *rhs* den Ausdruck, der durch *lhs* abgekürzt werden soll. Beide Seiten sind Termschemata mit eventuell frei vorkommenden (Meta-)variablen, die implizit allquantifiziert sind. Derartige Definitionen haben wir im vorigen Kapitel an vielen Stellen benutzt wie zum Beispiel bei der Einführung von Logik-Operatoren in Definition 3.3.3 auf Seite 134:

$$\begin{array}{ll} \textsf{and}\{\}(A;B) & \texttt{==}\ A\times B \\ \textsf{exists}\{\}(T;x.P) & \texttt{==}\ x{:}T\times P \\ \mathbb{P}i & \texttt{==}\ \mathsf{U}_i \end{array}$$

Hier sind A, B, T und P Platzhalter (oder *Metavariablen*) für Terme, x Platzhalter für eine Variable, die in P frei vorkommen darf und i Platzhalter für einen Parameter. Beim Auffalten einer konkreten Instanz der linken Seite werden diese Platzhalter durch Substitutionen an konkrete Terme oder Parameter gebunden und entsprechend auf der rechten Seite ersetzt. Um diesen Mechanismus zu realisieren, reichen gewöhnliche Substitutionen erster Stufe und der zugehörige Matching-Algorithmus jedoch nicht mehr aus, wie das folgende Beispiel zeigt.

Beispiel 4.1.3

Wenn wir versuchen, eine Substitution zu finden, die das Schema $\exists x{:}T.P$ in die spezielle Instanz $\exists \texttt{y:}\mathbb{N}\texttt{.y<5}$ überführt, so werden wir feststellen, daß die Substitution $[\texttt{y,y<5,}\mathbb{N}\ /x, P, T]$ hierfür nicht ausreicht, da ihre Anwendung auf das Schema gemäß Definition 3.2.7 (Seite 105) zu einer Umbenennung der Variablen `y`, also zu einem Term der Gestalt $\exists\texttt{y':}\mathbb{N}\texttt{.y<5}$ führen würde, in dem der gewünschte Zusammenhang zwischen der quantifizierten Variablen und der freien Variablen des Terms nicht mehr besteht. Auf die Umbenennungsvorschrift kann aber nicht verzichtet werden, da ansonsten auch andere Variablen unbeabsichtigt in den Bindungsbereich eines Quantors geraten können.

Ein Blick auf die interne Darstellung der in diesem Beispiel benutzten Terme zeigt, wo das Problem liegt. In $\textsf{exists}\{\}(T;x.P)$ repräsentiert die Metavariable P einen Term, in die Bindungsvariable x frei vorkommen kann. P ist in Wirklichkeit also eine Metavariable *zweiter Stufe* der Stelligkeit 1. Sie darf nicht durch einen einfachen Term substituiert werden, sondern nur durch einen Term, in dem ein Platzhalter für einen anderen Term vorkommt, der dann durch x zu ersetzen ist. Derartige Terme nennt man *Terme zweiter Stufe*. Sie entsprechen den gebundenen Termen im Sinne von Definition 3.2.5.

[17] Der Unterschied ist vergleichbar mit dem Unterschied zwischen `lettype` und `abstype` in ML.

Das Substitutionsverfahren ist also komplizierter als bei Substitutionen erster Stufe. Eine Substitution zweiter Stufe hat die Gestalt $[x_1, ..., x_m.t_{x_1,...,x_m}/P]$, wobei P eine Variable zweiter Stufe und $x_1, ..., x_m.t_{x_1,...,x_m}$ ein Term zweiter Stufe ist. Die Anwendung dieser Substitution auf eine konkrete Instanz $P[a_1, ..., a_n]$ der Variablen P liefert den Term $t_{a_1,...,a_n}$. Die konkreten freien Variablen innerhalb der Instanz P ersetzen also die Platzhalter $x_1, ..., x_m$ in y. Wenn wir also eine Substitution suchen, die $\exists x{:}T\,.\,P$ in $\exists$`y:`$\mathbb{N}$`.y<5` überführt, so benötigen wir die Substitution zweiter Stufe

$$[\texttt{y.y<5,}\ \mathbb{N}\ /P, T].$$

Die Anwendung dieser Substitution auf $\exists x{:}T\,.\,P$ führt dann zu dem Term $\exists x{:}\mathbb{N}\,.\,x$`<5`, der α-konvertibel zu dem gewünschten Term ist.[18]

Alle anderen Platzhalter, also Metavariablen erster Stufe und Platzhalter für Parameter sind unproblematisch. Sie können mit einem Verfahren behandelt werden, welches Matching und Substitutionsanwendung erster Stufe entspricht. Die Details des Abstraktionsmechanismus von NuPRL, insbesondere die konkreten Methoden zur Kennzeichnung von Variablen zweiter Stufe und von Meta-Parametern findet man in [Jackson, 1993b, Kapitel 5] beschrieben.

4.1.7.2 Termdarstellung

In Ergänzung des Abstraktionsmechanismus sorgt ein spezieller Display-Mechanismus für eine elegante und leicht lesbare Darstellung von mathematischen Texten auf dem Bildschirm. Neben der textlichen Präsentation eines Terms in einer nahezu beliebigen Syntax kann man mit diesem Mechanismus eine Reihe von anderen Kontrollmöglichkeiten über die Darstellung ausüben, wie zum Beispiel die folgenden:

- Formatierung des Textes in einem Fenster in Abhängigkeit von der Größe dieses Fensters.
- Automatische Klammerung von Termen niedrigerer Priorität, wenn diese als Teilterme eines anderen Terms auftreten. So wird **add**{}(**mul**{}(4,5),6) als 4*5+6 , **mul**{}(4,**add**{}(5,6)) aber als 4*(5+6) dargestellt, ohne daß der Benutzer dies erzwingen muß.
- Automatische Iteration von Operatoren wie z.B. die Verwendung von λ`x y.x+y` anstelle von λ`x.`λ`y.x+y`.
- Automatische Anwendung sonstiger Verkürzungen wie z.B. `4=5` anstelle von `4=5` $\in \mathbb{Z}$, sobald diese anwendbar sind.

Dies erhöht die Eleganz und Flexibilität der Darstellung formaler Texte und erleichtert somit den Übergang von informal präsentierten mathematischen Theorien zu der in Beweissystemen verwendeten Form.

Diese Mechanismen wurden vor allem durch die Hypertext-Technik möglich gemacht und sind noch im Stadium der Weiterentwicklung. Den gegenwärtigen Stand findet man in [Jackson, 1993b, Kapitel 6].

4.1.8 Der Programmevaluator

Die Terme der intuitionistischen Typentheorie beschreiben im Endeffekt eine funktionale Programmiersprache deren Berechnungsvorschriften wir im vorigen Kapitel ausführlich besprochen haben. Neben der Möglichkeit, mit dem Beweissystem über das Resultat dieser Berechnungen zu *schließen*, bietet NuPRL auch einen Mechanismus an, Programme – seien sie nun aus Theoremen extrahiert oder direkt geschrieben – auch innerhalb des Systems laufen zu lassen und praktisch auszutesten. Hierzu wird die Funktion `evaluate_term` eingesetzt, die den ohnehin in das System integrierten Auswertungsmechanismus aus Abbildung 3.3 (Seite 108) benutzt.

[18] In NuPRL ist der Standardalgorithmus für die Anwendung von Substitutionen zweiter Stufe zugunsten der besseren Lesbarkeit so abgewandelt worden, daß die tatsächlichen Namen der bindenden Variablen erhalten bleiben.

4.1.9 Korrektheit der Implementierung

Die Korrektheit von Beweisen, die mit Hilfe eines interaktiven Beweissystems geführt werden, kann wegen der abstrakten Definition des Datentyps proof in Abschnitt 4.1.2 relativ leicht sichergestellt werden. Veränderungen von Beweisen sind ausschließlich mit Hilfe der Funktion `refine` möglich, die wiederum auf die vordefinierten Inferenzregeln zurückgreifen muß. Unbefugte oder irrtümliche Manipulationen von Beweisen sind somit gänzlich ausgeschlossen. Die Zuverlässigkeit des interaktiven Beweissystems hängt somit ausschließlich von der korrekten Implementierung der Funktion `refine` und der einzelnen Inferenzregeln (und natürlich von der korrekten Arbeitsweise der Implementierungssprache ML und des Betriebsystems) ab.

Für die Implementierung von Inferenzregeln bietet sich eine schematische Darstellung an, die sich an die im vorigen Kapitel gegebene textliche Repräsentation von Regeln anlehnt und Metavariablen und -parameter gesondert kennzeichnet. Auf diese Art ist eine Übereinstimmung der implementierten Regeln mit dem formalen Theorie leicht zu überprüfen.

Die einzige Funktion, die wirklich verifiziert werden muß, ist die Funktion `refine`. Sie muß im wesentlichen ein Regelschema (samt Extraktterm) für ein gegebenes Beweisziel korrekt instantiieren und hieraus die Nachfolgerknoten und die Validierung generieren. Ihre Korrektheit folgt somit aus der Korrektheit der eingebauten Matching- und Substitutionsalgorithmen, die ebenfalls nicht schwer zu beweisen ist.

4.2 Taktiken – programmierte Beweisführung

Mit den bisher vorgestellten Komponenten des Beweisentwicklungssystems sind wir in der Lage, formale Beweise interaktiv zu entwickeln und hieraus Programme zu extrahieren. Als Benutzer des Systems brauchen wir nur unsere Theoreme zu formulieren und zum Beweis die jeweiligen Regeln anzugeben. Die Ausführung der Regeln, die Korrektheitsüberprüfung und vor allem die Schreibarbeit bei der Erzeugung der Teilziele bleibt dem System überlassen. Hierdurch wird die Entwicklung formaler Beweise mit garantierter Korrektheit bereits erheblich erleichtert.

Nichtsdestotrotz zeigt sich schon bei relativ einfachen Problemstellungen, daß eine formale Beweisführung immer noch zu kompliziert ist, da die Beweise zu viele Details enthalten, die sie unübersichtlich werden lassen. Aus diesem Grunde ist es notwendig, eine Rechnerunterstützung anzubieten, die über die Möglichkeiten interaktiver Beweissysteme hinausgeht, und Techniken bereitzustellen, mit denen die Beweisführung zumindest zum Teil automatisiert und übersichtlicher gestaltet werden kann. Hierzu könnte man nun das Inferenzsystem um fest eingebaute Beweisprozeduren ergänzen, die in derLage sind, Teilprobleme vollautomatisch zu lösen. In einigen Anwendungsbereichen (siehe Abschnitt 4.3) ist dies sicherlich ein sinnvoller Weg. Wegen der großen Ausdruckskraft der Typentheorie und der Vielfalt der Anwendungsmöglichkeiten ist diese Vorgehensweise jedoch nicht flexibel genug, da sie größere Eingriffe in das Beweisentwicklungssystem verlangen würde, wann immer eine neuartige Problemstellung damit bearbeitet werden soll.

Aus diesem Grunde ist es wichtig, Mechanismen bereitzustellen, die den Benutzern des Systems ermöglichen, das Inferenzsystem um eigene Beweisstrategien zu ergänzen und damit zu experimentieren, und dabei gleichzeitig die größtmögliche Sicherheit gegenüber fehlerhaften Beweisen zu bieten. Diese Idee der *benutzerdefinierten Erweiterung* von Beweissystemen, die erstmals innerhalb des LCF Projektes [Gordon *et.al.*, 1979] aufkam, läßt sich auf einfache Weise realisieren, wenn die Metasprache des Systems als Programmiersprache – in unserem Fall ML – formalisiert ist, da in diesem Fall ein Benutzer den Aufruf von Beweisregeln und die Bestimmung der nötigen Parameter *programmieren* kann. Erlaubt man einem Benutzer also, metasprachliche Programme in Beweisen zu verwenden, so erhält man die gewünschte Flexibilität in der Beweisführung, wobei die Darstellung der Datentypen proof und rule dafür sorgt, daß keine fehlerhaften Beweise entstehen können.

Durch einen Zugriff auf die Metasprache wird ein Benutzer in die Lage versetzt, Beweise im Voraus zu *planen* und entsprechend zu programmieren, anstatt sie rein interaktiv auszuführen. Darüber hinaus kann er aber auch Beweisstrategien entwickeln, die versuchen, einen Beweis automatisch zu finden, und somit

die Beweisführung von Teilproblemen entlasten, deren Beweis naheliegend ist aber voller mühsamer Details steckt. Die Programmierung auf der Metaebene des Kalküls ermöglicht also eine *taktische* Vorgehensweise in Ergänzung zu der interaktiven Beweisführung. *Taktiken* – also Metaprogramme, die auf Beweisen operieren – können eingesetzt werden, um nach Beweisen zu *suchen*, bestehende Beweise zu *modifizieren*, uninteressante Beweis*details zu verstecken*, komplexe Beweise zu *strukturieren* und – zusammen mit dem Abstraktionsmechanismus – die formale Sprache und das Inferenzsystem des zugrundeliegenden Kalküls um *benutzerdefinierte Konzepte* beliebig zu erweitern. Das Prinzip des *taktischen Theorembeweisens* liefert einen einfachen Weg, die Präzision von Computern mit dem Einfallsreichtum des menschlichen Benutzers zu koppeln und so einen einfachen Proof Checker in ein mächtiges Werkzeug für den Beweis von Theoremen, die Konstruktion von Programmen und die Entwicklung formaler mathematischer Theorien zu verwandeln.

In diesem Abschnitt wollen wir nun die wesentlichen Aspekte einer Realisierung des Taktik-Konzeptes am Beispiel der wichtigsten NuPRL Taktiken diskutieren.

4.2.1 Grundkonzepte des taktischen Beweisens

Das Konzept der taktischen Beweisführung basiert im wesentlichen auf der Methode des heuristischen Problemlösens, die schon von den altgriechischen Mathematikern eingesetzt, von Polya [Polya, 1945] systematisiert und erstmalig im Logic Theorist von Newell, Shaw und Simon [Newell *et.al.*, 1963] programmiert wurde. In dieser Denkweise ist ein Problem (oder Beweisziel) nichts anderes als "eine Menge möglicher Lösungskandidaten zusammen mit einem Testverfahren, welches überprüft ob ein gegebenes Element dieser Menge tatsächlich eine Lösung für das Problem ist" [Minsky, 1963]. In LCF [Gordon *et.al.*, 1979] wurde dieser Gedanke noch verallgemeinert und das Testverfahren durch den Begriff des *Erreichens* (englisch: *achievement*) ersetzt, der eine mögliche Beziehung zwischen einem Ziel (*goal*) und einem (Lösungs-)*Ereignis* (*event*) herstellt. In diesem Sinne können "viele Situationen des Problemlösens als spezielle Instanzen der drei Konzepte Ziel, Ereignis und Erreichen" verstanden werden.

Diese allgemeine Sicht auf Problemlöseprozesse erlaubt es, universelle Techniken zu entwickeln, die sich in allen Bereichen des Problemlösens – also nicht nur innerhalb eines festen Kalküls – einsetzen lassen und mittlerweile in vielen Systemen (siehe Fußnote 3 auf Seite 182) Anwendung gefunden haben. Taktiken sind dabei nichts anderes als eine Formulierung der Idee einer zielorientierten (top-down) Heuristik in dieser Denkweise. Eine Taktik zerlegt ein Beweis*ziel* G in eine endliche Liste von Teilzielen $G_1, \ldots, G_n$ sowie eine Validierung v. Wenn nun die *Ereignisse* e_i die Teilziele g_i *erreichen*, dann dient die Validierung v dazu, ein Ereignis $e = v(e_1, \ldots, e_n)$ zu konstruieren, welches das ursprüngliche Ziel G erfüllt. Bei dieser Vorgehensweise wird also im Top-Down Verfahren nach einer Lösung des Problems gesucht und diese dann bottom-up (zu einem erreichenden Ereignis) zusammengesetzt.

In der Sprache von Beweiskalkülen können wir ein Ziel als ein Theorem (also eine Sequenz) ansehen und ein Ereignis als seinen Beweis, welcher eine Evidenz für die Gültigkeit des Theorems sein soll. Beweise dürfen dabei unvollständig sein und Sequenzen können als degenerierte unvollständige Beweise angesehen[19] werden. Dieses Verständnis führt dann genau zu den in Abschnitt 4.1.2 angegebenen Datentypen. Der Begriff des Erreichens ist nun recht einfach umzusetzen: ein Beweis p erreicht ein Ziel G, wenn die Sequenz in seiner Wurzel genau die Sequenz G (also die Wurzelsequenz des degenerierten Initialbeweises) ist.[20]

Eine Taktik löst also einen Teil der Problemstellung und hinterläßt Teilziele, die sie nicht vollständig beweisen konnte und eine Validierung, welche Beweise für diese Teilziele in einen Beweis des Ausgangsziels umwandelt. Wenn diese Validierung nur auf vollständige Beweise angewandt wird – insbesondere also, wenn überhaupt keine Teilziele übrigbleiben, dann entsteht ein vollständiger Beweis des ursprünglichen Ziels.

[19] Wenn Taktiken in einem interaktiven System aufgerufen werden sollen, müssen sie auch unvollständige Beweise generieren dürfen, ohne fehlzuschlagen. Die Identifizierung von Sequenzen und Beweisen wird besonders bei Transformationstaktiken wichtig.

[20] In der NuPRL-Implementierung des Taktik-Konzeptes wird sichergestellt, daß Taktiken, wenn sie überhaupt anwendbar sind, nur Beweise erzeugen können, deren Wurzelsequenz das Ausgangsziel ist. Auf diese Art braucht der Begriff des Erreichens nicht gesondert überprüft zu werden.

Der Vorteil dieser etwas kompliziert anmutenden Vorgehensweise, die uns bereits in Abschnitt 4.1.2 im Zusammenhang mit der Implementierung von Beweisregeln begegnet ist, wird deutlich, wenn wir Taktiken zusammensetzen wollen. Da eine Taktik eine explizite Liste von Teilzielen generiert, können wir andere Taktiken einfach hierauf anwenden und somit das Beweisziel schrittweise zerlegen. Der Aufbau des eigentlichen Beweises wird dabei automatisch durch die entsprechenden Validierungen durchgeführt. Dies macht es sehr einfach, spezialisierte Taktiken zu programmieren, die nur in wenigen Fällen Anwendung finden (siehe Abschnitt 4.2.5) und Hilfsmittel für eine leichte Komposition von Taktiken bereitzustellen.

Es gibt zwei grundsätzliche Klassen von Taktiken: *Verfeinerungstaktiken* und *Transformationstaktiken*.

4.2.2 Verfeinerungstaktiken

Verfeinerungstaktiken können in erster Näherung als *abgeleitete Inferenzregeln* betrachtet werden. Sie werden angewandt, indem im Beweiseditor anstelle einer elementaren Kalkülregel der Name einer Taktik eingegeben wird. Dieser Name wird auch als Regelname im Editorfenster erscheinen, wenn die Taktik erfolgreich ausgeführt werden kann. Als Teilziele erscheinen dann diejenigen Ziele, die von der Taktik nicht vollständig bewiesen werden konnten. Alle zwischenzeitlich berechneten Ziele werden zwar (zugunsten einer effizienten Extraktion) im Beweis gespeichert, bleiben aber für den Anwender des Systems unsichtbar.

Beispiel 4.2.1

Ein typisches Beispiel für eine Verfeinerungstaktik ist die Taktik `cases`, welche eine Fallunterscheidung ausführt. Dies geschieht normalerweise dadurch, daß man zuerst eine Disjunktion $A \vee B$ über die Schnittregel einführt und dann diese Disjunktion eliminiert, um die Fälle A und B einzeln betrachten zu können. Im Beweiseditor würde man also die folgenden Schritte ausführen

EDIT THM cases
top $\vdash T$
BY cut 1 $A \vee B$
1# $\vdash A \vee B$
2# 1. $A \vee B \ \vdash T$

und danach

EDIT THM cases
top 2 1. $A \vee B \ \vdash T$
BY or_e 1
1# 1. $A \ \vdash T$
2# 1. $B \ \vdash T$

Eigentlich sind diese beiden Schritte jedoch eine Einheit und müßten zusammengefaßt werden. Dies genau geschieht durch die Taktik `cases`, die wir in Beispiel 4.2.4 auf Seite 204 ausprogrammieren werden. Sie liefert in einem Schritt folgendes Resultat.

EDIT THM cases
top $\vdash T$
BY cases A B
1# $\vdash A \vee B$
2# 1. $A \ \vdash T$
3# 1. $B \ \vdash T$

Der Zwischenschritt wird hierbei zwar wie oben ausgeführt, bleibt aber unsichtbar.

Im Prinzip führt der Beweiseditor beim Aufruf einer Verfeinerungstaktik dieselben Schritte aus wie bei der Anwendung einer elementaren Beweisregel (vergleiche unsere Beschreibung in Beispiel 4.1.2 auf Seite 194).

1. Die globale Variable `prlgoal` wird mit der aktuellen Beweissequenz assoziiert. Bisher bestehende Ziele und Regeln eines eventuell schon vorhandenen Teilbaumes werden ignoriert.

2. Die Taktik wird auf die Variable `prlgoal` angewandt, was zu einer (möglicherweise leeren) Liste von Teilzielen und einer Validierung führt.

3. Die Validierung wird auf die Teilziele angewandt und erzeugt einen Beweisbaum.

4. Der Beweisbaum wird zusammen mit dem Namen der Taktik als Verfeinerungsregel des aktuellen Beweises eingetragen. Die von der Taktik generierten Teilziele werden die Teilziele des Verfeinerungsschrittes.

5. Der Inhalt des Beweiseditorfensters wird neu berechnet und angezeigt, wobei nur der Name der Taktik als Verfeinerungsregel erscheint. Ist die Taktik nicht anwendbar, so wird der Status auf *fehlerhaft* gesetzt und eine Fehlermeldung ausgegeben.

Man beachte, daß der eigentliche Beweis erst durch die Validierung aufgebaut wird, nachdem alle Teilziele berechnet wurden. Dadurch entfällt die Notwendigkeit, einen Teil des erzeugten Beweises wieder rückgängig zu machen, wenn die Taktik nach vielen Schritten im Endeffekt fehlschlagen sollte.

4.2.3 Transformationstaktiken

Transformationstaktiken sind von ihrem Typ her identisch mit Verfeinerungstaktiken, unterscheiden sich von diesen aber durch die Art der Anwendung. Ihr Zweck ist nicht die Verfeinerung eines Beweisziels sondern die Transformation eines bestehenden Beweises in einen anderen. Daher betrachten sie üblicherweise auch nicht nur das Beweisziel, sondern den gesamten Beweisbaum, der in dem aktuellen Knoten des Beweises beginnt. Aus Benutzersicht erzeugen sie auch nicht nur neue Teilziele sondern einen ganzen Beweisbaum. Nach erfolgreicher Anwendung erscheint ihr Name normalerweise nicht im entstandenen Beweis. Transformationstaktiken werden üblicherweise dazu eingesetzt, Beweise sichtbar zu vervollständigen, zu expandieren, oder Beweise zu konstruieren, die in einem gewissen Sinn analog zu bestehenden Beweisen sind.

Beispiel 4.2.2

Ein typisches Beispiel für die Anwendung von Transformationstaktiken ist das Taktik-Paar `Mark` und `Copy`, die dazu benutzt werden können, das gleiche Argument an mehreren Stellen eines Beweises anzuwenden, ohne hierzu ein Lemma zu formulieren. Hierzu wird in einem ersten Schritt der in einem Knoten beginnende Beweis mit `Mark` ‘*name*‘ in einer globalen Variablen abgespeichert. Anschließend bewegt man sich zu dem Beweisknoten (ggf. auch in einem anderen Theorem-Objekt), in dem dasselbe Argument benutzt werden soll und benutzt `Copy` ‘*name*‘, um diesen Beweis erneut auszuführen. Dabei müssen alle Regeln des gespeicherten Beweises ohne Abänderung anwendbar sein, um eine analoge Kopie des Beweises zu erzeugen.

Eine Transformationstaktik wird innerhalb des Beweiseditors durch Aufruf des Termeditors mithilfe eines speziellen Befehls generiert. Bei ihrer Ausführung werden folgende internen Schritte durchgeführt.

1. Die Variable `prlgoal` wird mit dem gesamten Beweis unterhalb der aktuellen Beweissequenz assoziiert.

2. Die Taktik wird auf `prlgoal` angewandt und liefert eine Liste von Teilzielen und eine Validierung.

3. Die Validierung wird auf die Teilziele angewandt und erzeugt einen Beweisbaum.

4. Der gesamte erzeugte Beweisbaum wird in den Beweis des aktuellen Theorems anstelle des vorherigen Teilbeweises integriert.

5. Der Inhalt des Beweiseditorfensters wird neu berechnet und angezeigt. Ist die Taktik nicht anwendbar, so bleibt der Beweis unverändert und eine Fehlermeldung wird ausgegeben.

Eine Transformationstaktik schreibt insbesondere auch die Namen aller angewandten Regeln in die entsprechenden Regel-Felder des Beweiseditors und zeigt dem Anwender somit alle Details des ausgeführten Beweises. Daher ist es durchaus auch sinnvoll, Taktiken, die ursprünglich als Verfeinerungstaktiken geschrieben wurden, als Transformationstaktiken ausführen zu lassen. In diesem Fall wird nicht, wie sonst üblich, nur das Endergebnis der Beweisführung angezeigt, sondern alle einzelnen Schritte des Beweises sichtbar gemacht. Wir wollen diesen Unterschied an einem Beispiel erläutern.

Beispiel 4.2.3

Die Taktik `simple_prover`, deren Implementierung wir in Abbildung 4.4 auf Seite 207 beschreiben werden, ist in der Lage, einfache logische Schlüsse automatisch auszuführen, solange hierzu keine komplexen Entscheidungen zu treffen sind wie etwa die Angabe eines Terms für die Ausflösung eines Quantors, oder die Auswahl einer zu beweisenden Alternative in einer Disjunktion. Von ihrer Konzeption her ist sie eine typische Verfeinerungstaktik. Sie kann zum Beispiel benutzt werden, um das Ziel

$\forall$x:IN. $\forall$y:IN. x<y $\wedge$ ($\neg$x<y $\vee$ 4=x) $\Rightarrow$ 4=x

vollständig zu beweisen. Um sie als Verfeinerungstaktik anzuwenden, müssen wir sie als Regel des entsprechenden Beweiszieles angeben:

```
EDIT THM or_test
 # top
 ⊢ ∀x:IN. ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY <refinement rule>
```

```
EDIT Rule of or_test

 simple_prover
```

Nachdem wir das Editorfenster für den Regelnamen geschlossen haben, wird die Taktik `simple_prover` als Verfeinerungstaktik ausgeführt und liefert in einem Schritt den vollständigen Beweis.

```
EDIT THM or_test
 * top
 ⊢ ∀x:IN. ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY simple_prover
```

An diesem Beweis kann man nur erkennen, daß die Taktik `simple_prover` in der Lage war, den Beweis alleine auszuführen. Solange man sich für die Details dieses Beweises nicht interessiert, ist diese Anwendungsform die beste Vorgehensweise, zumal sie immer noch den vollständigen Extraktterm (λx.λy.... liefert. Es ist allerdings auch möglich, `simple_prover` so ablaufen zu lassen, daß man jeden einzelnen Schritt, den diese Taktik ausgeführt hat, zu sehen bekommt.[21] Dies geschieht durch Aufruf des Editors für Transformationstaktiken

```
EDIT THM or_test
 # top
 ⊢ ∀x:IN. ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY <refinement rule>
```

```
EDIT Transformation Tactic

 simple_prover
```

Nachdem das Editorfenster geschlossen ist, wird `simple_prover` als Transformationstaktik ausgeführt. Dies liefert ebenfalls einen vollständigen Beweis, zeigt im Fenster aber nur den ersten Schritt an.

```
EDIT THM or_test
 * top
 ⊢ ∀x:IN. ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY all_i

 1* 1. x:IN
    ⊢ ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
```

Den vollständigen Beweis kann man nun beim Durchlaufen des Beweises mit dem Beweiseditor ansehen oder mit der Transformationstaktik `PrintTexFile` *filename* in der vertrauten Kurzform (siehe Abbildung 4.2) als LaTeX-File darstellen lassen.

[21] Durch eine spezielle Funktion `Run`, welche Taktiken in elementare Regeln umwandelt, kann man allerdings verhindern, daß eine Taktik im Transformationsmodus aufgelöst wird. Dies ist sinnvoll, wenn man eine Taktik wie `simple_prover` nur in ihre Hauptbestandteile (die logischen "Regeln") zerlegen lassen will.

```
⊢ ∀x:ℕ. ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
BY all_i
 \
  x:ℕ ⊢ ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
  BY all_i
   \
    x:ℕ, y:ℕ  ⊢ x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
    BY implies_i
     \
      x:ℕ, y:ℕ,  x<y ∧ (¬x<y ∨ 4=x) ⊢ 4=x
      BY and_e 3
       \
        x:ℕ, y:ℕ, x<y, ¬x<y ∨ 4=x  ⊢  4=x
        BY or_e 4
        |\
        | x:ℕ, y:ℕ, x<y, ¬x<y ⊢ 4=x
        | BY not_e 4
        |  \
        |   x:ℕ, y:ℕ, x<y, ¬x<y ⊢ x<y
        |   BY hypothesis 3
         \
          x:ℕ, y:ℕ, x<y, 4=x  ⊢ 4=x
          BY hypothesis 4
```

Abbildung 4.2: Von der Taktik `simple_prover` generierter Beweis

4.2.4 Programmierung von Taktiken

Wir wollen nun untersuchen, wie einfache Taktiken in ML geschrieben werden können, ohne daß man hierzu alle Details des NuPRL Systems verstehen muß.

Die Grundlage aller Taktiken ist die Funktion `refine`, mit der man die elementaren Regeln des NuPRL Systems in Taktiken umwandeln kann. Da alle im vorigen Kapitel vorgestellten Regeln der intuitionistischen Typentheorie bereits mittels `refine` in "Regel-Taktiken" umgewandelt wurden, ist eine explizite Verwendung der Funktion `refine` (und die damit verbundene Handhabung der Argumente einer Regel) für den normalen Anwender von NuPRL nicht erforderlich.[22] Er kann stattdessen davon ausgehen, daß die Inferenzregeln der Typentheorie als Taktiken vorliegen, und auf dieser Basis dann Spezialtaktiken zusammensetzen, die für seine Anwendungen besonders geeignet sind.

Auch hierfür ist es normalerweise nicht erforderlich, in die Details der Programmiersprache ML einzusteigen. Stattdessen können Anwender von NuPRL eine Reihe von vordefinierten *tacticals* benutzen, um bestehende Taktiken in neue Taktiken umzuwandeln. Tacticals sind nichts anderes als ML Funktionen des Typs `tactic -> tactic`, deren besonderer Zweck darin liegt, die imperative Denkweise bei der Anwendung von Regeln und Taktiken in einer funktionalen Programmiersprache auszudrücken.

Die häufigste Art, Taktiken zu kombinieren, ist die *Hintereinanderausführung*: um ein Ziel zu beweisen, wendet man eine gewisse Regel an, dann eine zweite auf die entstehenden Teilziele, dann eine dritte auf deren Resultate usw. Dieser Effekt kann mit dem Tactical `THEN` erreicht werden, welches als Funktion in *Infix-*

[22] Bei dieser Umwandlung wird insbesondere die automatische Umbenennung von Variablennamen durchgeführt, die in der Regel nicht direkt enthalten ist. Stattdessen verlangt die Regel die Angabe des Variablennamens als Parameter. Mithilfe einer Funktion `new : tok -> proof -> tok`, welche überprüft, ob eine bestimmte Variable bereits im Beweis deklariert ist und diese dann ggf. umbenennt, kann man die Regel-Taktik `lambdaI` dann z.B. wie folgt implementieren.

```
let lambdaI pf =
    let [id,S;(),T] = bound_terms_of_term (conclusion pf)
    in
      refine (make_primitive_rule `lambdaFormation` [make_level_expression_argument level; new id pf]) pf
;;
```

Da das Schema der Umwandlung bei allen Regeln gleich ist, enthält die tatsächliche Implementierung aller Regeltaktiken einige Hilfsfunktionen, welche die eben beschriebene Umwandlung eleganter durchführen.

- t_1 `THEN` t_2: *"Wende t_2 auf alle von t_1 erzeugten Teilziele an"*
  t `THENL` $[t_1; t_2; ..t_n]$: *"Wende t_i auf das i-te von t erzeugte Teilziel an"*
- t_1 `ORELSE` t_2: *"Wende t_1 an. Falls dies fehlschlägt, wende t_2 an".*
- `Repeat` t: *"Wiederhole die Taktik t bis sie fehlschlägt"*
- `Complete` t: *"Wende t nur an, wenn hierdurch der Beweis vollständig wird"*
- `Progress` t: *"Wende t nur an, wenn ein Fortschritt erzielt wird"*
- `Try` t: *"Wende t an; falls dies fehlschlägt, lasse das Ziel unverändert"*

Abbildung 4.3: Wichtige vordefinierte Tacticals

Notation vordefiniert ist. Eine Variante von `THEN` ist das Tactical `THENL`, die eine individuellere Handhabung der entstandenen Teilziele ermöglicht, indem man zu jedem der entstehenden Teilziele eine eigene Regel, insgesamt also eine Liste von Taktiken angibt. Neben der Programmierung neuer Taktiken unterstützen diese beiden Tacticals besonders auch Beweise, in denen man den Effekt der nächsten Schritte vorausplanen und zu einem Schritt zusammenfassen will. Da man hierbei oft auch nur einige der Teilziele automatisch weiterverarbeiten will (z.B. nur Wohlgeformtheitsziele), wurde eine spezielle Taktik `Id` vordefiniert, die ein Teilziel unverändert läßt. Ihre Implementierung ist denkbar einfach, da nur die Typstruktur verändert werden muß.

```
let Id (pf:proof) = [pf],hd;;
```

Beispiel 4.2.4 (Kombination von Taktiken durch Hintereinanderausführung)

Ein typisches Beispiel für eine Taktik, die nur durch Hinterausführung von Taktiken (Regeln) programmiert werden kann, ist die in Beispiel 4.2.1 auf Seite 200 angesprochene Taktik `cases`. Sie besteht darin, erst eine Disjunktion per `cut` einzuführen und dann das zweite Teilziel mit `or_e` zu zerlegen:

```
let cases A B =
          cut (-1) (make_or_term A B)
   THENL [Id ; or_e (-1)]
;;
```

Für ein mehr experimentelles Vorgehen ist das (Infix-)Tactical `ORELSE` da. Es ermöglicht, die Anwendung einer Taktik t_1 zu versuchen und eine zweite Taktik t_2 zu starten, wenn die Anwendung der ersten fehlschlägt. Dies wird insbesondere dann eingesetzt, wenn man sich über die genaue Struktur der zu erwartenden Ziele nicht ganz im klaren ist, sondern nur weiß, das eine der angegebenen Taktiken anwendbar sein wird. `ORELSE` ist somit für das weitestgehend automatische Suchen nach Beweisen ein entscheidendes Hilfsmittel.

Das Tactical `Repeat` wendet eine Taktik t solange an, bis sie fehlschlägt. Die Ausnahmen, die durch das Fehlschlagen der Taktik t auf einigen Unterzielen entstehen, werden dabei abgefangen. Hiermit kann man zum Beispiel eine einfache Taktik `repeatedIntro` schreiben, welche logische Einführungsregeln solange anwendet, bis das eigentlich interessante Teilziel offengelegt ist.

```
let repeatedIntro =
   Repeat (       all_i
           ORELSE imp_i
           ORELSE not_i
           ORELSE and_i
          )
;;
```

Diese Taktik übernimmt also die langwierigen trivialen Schritte eines Beweises, die ohnehin naheliegend sind.

Die wichtigsten weiteren einfachen Tacticals, die sich als nützlich für die Erstellung neuer Taktiken herausgestellt haben, sind `Complete`, `Progress` und `Try`. Abbildung 4.3 faßt ihre Bedeutung kurz zusammen. Weitere interessante Tacticals sind in [Jackson, 1993b, Kapitel 8.3] zusammengestellt. Die Implementierung dieser Tacticals ist nicht sehr schwierig, wie das folgende Beispiel zeigt.

Beispiel 4.2.5

Die Tacticals `THENL` und `ORELSE` müssen als Infix-Operatoren vereinbart werden (wozu die Funktion `ml_curried_infix` bereitsteht) und werden aus Gründen der Übersichtlichkeit groß geschrieben.

Die Programmierung von `THENL` ist nichts anderes als ein sorgfältiges Zusammensetzen von Teilzielen und Validierungsfunktionen. t `THENL` `[`$t_1; t_2; ..t_n$`]` muß als Teilziele die Taktiken t_i auf die Resultate der Anwendung von t anwenden, wozu eine elementare Listenfunktion `map_apply` benutzt werden kann. Das Resultat ist eine Liste von Listen von Beweiszielen, die "flach" gemacht werden muß. Die Validierungen der t_i finden eine solche flache Liste vor und müssen nun auf die jeweils passende Teilliste angewandt werden um eine Liste von Beweisen zu erzeugen, auf die dann die Validierung von t angewandt wird.

`ORELSE` wird mit dem Ausnahmebehandlungsmechanismus von ML programmiert. Hierbei ist allerdings zu beachten, daß nur die *Anwendung* einer Taktik eine Ausnahme erzeugt, nicht aber die Taktik selbst. Statt t_1 `?` t_2 muß man daher `\pf.` t_1 `pf ?` t_2 `pf` schreiben. Für die Programmierung von `Complete` muß man nur testen, ob nach der Anwendung der Taktik noch zu beweisende Teilziele vorhanden sind. In diesem Fall muß eine Ausnahme ausgelöst werden.

```
ml_curried_infix `THENL` ;;
ml_curried_infix `ORELSE` ;;

let $THENL (tac: tactic) (tac_list : tactic list) (pf:proof) =
  let subgoals, val = tac pf
  in
     if not length tac_list = length subgoals
        then fail
        else let subgoalLists, valList = map_apply tac_list subgoals
             in
                (flatten subgoalLists),
                \proofs. val ( (mapshape (map length subgoalLists) valList) proofs)
;;
let $ORELSE (t1:tactic) (t2:tactic) pf =  t1 pf ? t2 pf ;;

let Complete (tac:tactic) (pf:proof) = let subgoals, val = tac pf
                                       in
                                          if null subgoals
                                             then subgoals, val
                                             else fail
                                       ;;
```

Die Programmierung der anderen in Abbildung 4.3 genannten Tacticals ist eine einfache Übung.

Natürlich steht es einem Anwender auch frei, Taktiken durch wesentlich komplexere ML-Programme zu erzeugen als nur durch die Anwendung von Tacticals. Letztere erleichtern eine übersichtliche Programmierung jedoch ungemein, so daß auch trickreichere Taktiken zu einem Teil auf Tacticals zurückgreifen werden. Auch hierzu wollen wir ein Beispiel geben.

Beispiel 4.2.6

Die Taktik `hypcheck` überprüft, ob ein Beweisziel unmittelbar aus einer der Hypothesen folgt. Dies kann geschehen, indem man die Regel `hypothesis` auf alle Hypothesen des Beweises anwendet. Da die bisher angegebenen Tacticals hierfür nicht weiterhelfen, muß hierfür eine rekursive Taktik geschrieben werden.

```
let hypcheck (pf:proof)  =
  let n = length (hypotheses pf)
  in
      letrec TryHyp pos =
        if pos<=n  then hypothesis pos ORELSE TryHyp (pos+1)
                   else Id
      in
         TryHyp 1 pf
;;
```

Dieses Durchprobieren aller Hypothesen auf Anwendbarkeit einer Taktik ist auch in anderen Fällen sinnvoll. Aus diesem Grund lohnt es sich, die obige Implementierung zu einem neuen Tactical `TryAllHyps (tac: int -> tactic) (pf:proof)` zu verallgemeinern, welches anstelle der Regel `hypothesis` eine beliebige Taktik anwendet. Dieses Tactical findet zum Beispiel bei der Programmierung der Taktik `simple_prover` in Abbildung 4.4 mehrfach Verwendung.

Nur mit den vordefinierten Tacticals, den (umgewandelten) NuPRL Regeln und wenigen vordefinierten Taktiken kann ein Anwender des NuPRL Systems bereits bemerkenswert leistungsfähige Taktiken programmieren. Ein sehr interessantes Beispiel hierfür ist die Taktik `simple_prover`, die in der Lage ist, viele einfache logische Schlüsse selbständig auszuführen.

Beispiel 4.2.7

Die Taktik `simple_prover`, deren ML-Implementierung in Abbildung 4.4 angegeben ist, spiegelt die in Abschnitt 2.2.9 gegebenen Richtlinien für das Führen logischer Beweise wieder, welche die Prioritäten der Regeln nach den Kosten ihrer Anwendung sortiert.

Zuerst wird versucht, Taktiken anzuwenden, die keinerlei Teilziele erzeugen und somit das Beweisziel abschließen oder sofort fehlschlagen. Die Elimination von Konjunktionen und Existenzquantoren sorgt nur für eine feinere Auflösung der Hypothesenliste und ist somit gefahrlos (schlimmstenfalls hätte die Hypothese in einem *späteren* Teilziel als ganzes verwendet werden können und muß dann wieder zusammengebaut werden). Die Einführung von Allquantor, Implikation, Negation und Konjunktion ist ebenfalls nur mit geringen Kosten verbunden. Eine Auflösung einer Disjunktion in einer Hypothese erzeugt zwei Teilziele und verdoppelt die Beweislast (was teuer ist, wenn die Disjunktion keine Rolle spielt). Rückwärtsschließen über Negationen und Implikationen wird nur mit einer gewissen Vorsicht einzusetzen sein.

Die Einführung von Existenzquantoren und die Elimination universell quantifizierter Formeln erfordert eine gewisse Vorausschau, welcher Term zur Instantiierung der Variablen benötigt wird, und wird deshalb nicht mit aufgenommen. Aus einem ähnlichen Grund wird auf die Regeln `or_i1` und `or_i2` verzichtet, da diese den Beweis in eine falsche Richtung führen können und nur bewußt eingesetzt werden sollten.

Auch die Programmierung von echten Transformationstaktiken ist nicht sehr aufwendig. Wir wollen dies am Beispiel der Taktiken `Mark` und `Copy` illustrieren, die dazu benutzt werden können, Teilbeweise zu sichern und zu kopieren.

Beispiel 4.2.8

Die Taktik `Mark` legt eine direkte Kopie des aktuellen Teilbeweises unter einem angegebenen Namen in einer Tabelle `saved_proofs` ab, die als globale Variable vom Typ `(tok#proof) list` deklariert ist. Neue Einträge in diese Liste geschehen mit der Funktion `add_saved_proof` und mit `get_saved_proof` kann man den unter einem Namen abgelegten Teilbeweis wieder auffinden.

Da das Ablegen der Kopie alleine keine Taktik darstellt, muß zugunsten der Typkorrektheit im Anschluß daran die leere Taktik `Id` angewandt werden. Hierzu benutzt `Mark` ein imperatives Merkmal – die sequentielle Auswertung von Funktionen.

`Copy` durchläuft den abgesicherten Beweis rekursiv und benutzt die jeweilige `refinement` Komponente, um den Namen der Regel des Beweises zu erzeugen. Da dies fehlschlagen wird, wenn der abgelegte Teilbeweis unvollständig war, muß vorher abgefragt werden, ob überhaupt eine `refinement` Komponente existiert. Ist dies nicht der Fall, so ist die Taktik `Id` aufzurufen.

```
let Mark name pf = add_saved_proof name pf; Id pf ;;

letrec copy_pattern pattern pf =
  if is_refined pattern
    then Try ( refine (refinement pattern) THENL (map copy_pattern (children pattern)) ) pf
    else Id pf
;;
let Copy name  = copy_pattern (get_saved_proof name)
```

```
%-------------------------------------------------------------------------+
| PREDEFINED TACTICALS                                                    |
|                                                                         |
|  TryAllHyps: (int -> tactic) -> tactic[23]                              |
|             Try to apply a given tactic with to the hypotheses of the proof|
|                                                                         |
|  Chain: (* -> tactic) -> tactic -> tactic                               |
|             Try to apply a given tactic repeatedly to all the hypotheses |
|             of the proof or apply a given basetactic. Chaining is limited|
|             to a certain number of steps and must complete a proof.     |
|                                                                         |
|  Run:       convert a tactic into a rule to make it primitive. This     |
|             prevents transformation tactics from showing unwanted details.|
|                                                                         |
+-------------------------------------------------------------------------+
|                                                                         |
|   hypcheck:          check whether the goal follows from a hypothesis   |
|   contradiction:     try to find a contradictory hypothesis             |
|   conjunctionE:      eliminate all conjunctions in hypotheses           |
|   disjunctionE:      eliminate all disjunctions in hypotheses           |
|   impChain:          chain several hypotheses in order to completely    |
|                      prove the goal                                     |
|                                                                         |
|   simple_prover:     A tactic trying to apply simple steps in increasing|
|                      order of costs                                     |
|                                                                         |
+-------------------------------------------------------------------------%
```

```
let hypcheck          = TryAllHyps hypothesis             ;;
let contradiction     = TryAllHyps false_e                ;;
let conjunctionE      = TryAllHyps and_e                  ;;
let existentialE      = TryAllHyps ex_e                   ;;
let disjunctionE      = TryAllHyps or_e                   ;;
let impChain          = Chain impE hypcheck               ;;
let notChain          = TryAllHyps not_e    THEN impChain ;;

let nondangerousI pf  = let termkind = opid (conclusion pf)
                        in
                           if member termkind [‘all‘; ‘not‘; ‘implies‘;
                                               ‘rev_implies‘; ‘iff‘; ‘and‘]
                              then Run (termkind ^ ‘_i‘) pf
                              else fail
                        ;;

let simple_prover    = Repeat
                           (       hypcheck
                            ORELSE contradiction
                            ORELSE conjunctionE
                            ORELSE existentialE
                            ORELSE nondangerousI
                            ORELSE disjunctionE
                            ORELSE notChain
                            ORELSE impChain
                           );;
```

Abbildung 4.4: Implementierung der Taktik `simple_prover`

Die tatsächliche Implementierung von `simple_prover` wurde mittlerweile noch weiter verfeinert und im Hinblick auf Effizienz optimiert. Das Tactical `TryALlHyps` nimmt nun zum Beispiel eine Vorselektierung vor, bei der Hypothesen ignoriert werden, auf welche die genannte Taktik prinzipiell nicht anwendbar ist. Diese Selektion ist effizienter als eine fehlschlagende Taktik abzufangen.

4.2.5 Korrektheit taktisch geführter Beweise

Wir haben bisher an vielen Beispielen die Vielfalt der Einsatzmöglichkeiten von Taktiken im automatischen Beweisen illustriert. Da einem Benutzer von taktisch gesteuerten Beweisentwicklungssystemen zur Programmierung von Taktiken der gesamte Sprachumfang einer Turing-mächtigen Programmiersprache (in unserem Fall der von ML) zur Verfügung steht, gibt es praktisch keinerlei Restriktionen beim Schreiben von Taktiken. Dies wirft natürlich die Frage auf, wie zuverlässig taktisch geführte Beweise bei derartigen Freiheitsgraden noch sein können. Solange das Beweissystem rein interaktiv war, folgte die Zuverlässigkeit eines Beweisers aus der korrekten Implementierung der Funktion `refine` und der Korrektheit der einzelnen Regeln. Es ist aber auch bekannt, daß bei *automatischen* Theorembeweisern die gesamte Beweisprozedur verifiziert werden muß, wenn man garantieren will, daß ein maschinell geführter Beweis auch tatsächlich korrekt ist. Ist dies nun für Beweistaktiken ebenfalls nötig?

Zum Glück kann diese Frage eindeutig mit "nein" beantwortet werden, da der Freiheitsgrad beim Programmieren von Taktiken in einem Punkt doch eingeschränkt ist, nämlich im Bezug auf die Möglichkeiten, einen Beweis zu manipulieren. Da Beweise gemäß Abschnitt 4.1.2 Elemente eines *abstrakten* Datentyps sind, können sie ausschließlich mithilfe der Funktion `refine` verändert werden. Im Endeffekt müssen daher alle Taktiken auf die Funktion `refine` und die vordefinierten Regeln des Beweissystems zurückgreifen. Eine andere Möglichkeit, Beweise zu manipulieren, gibt es nicht. Somit wird auf eine sehr einfache Art der Implementierung sichergestellt, daß jeder Beweis, der durch eine Taktik – sei es nun eine Verfeinerungstaktik oder eine Transformationstaktik – erzeugt wird, auch korrekt ist.

Satz 4.2.9

> *Das Resultat einer Taktik-Anwendung auf ein Beweisziel ist immer ein gültiger Beweis des zugrundeliegenden logischen Kalküls.*[24]

Dieser Satz erlaubt dem Anwender eines taktisch gestützten Beweisssystems, das Inferenzsystem des zugrundeliegenden Kalküls nach Belieben um eigene Beweisverfahren zu ergänzen, *ohne sich dabei Gedanken über die Korrektheit seiner Implementierungen zu machen.* Die Tatsache, daß Taktiken im Endeffekt auf den Regeln des Kalküls aufbauen, macht die benutzerdefinierten Erweiterungen automatisch konsistent mit dem Rest der Theorie. Das Schlimmste, was passieren könnte, wäre daß eine Taktik kein Resultat liefert. Da aber jedes durch Taktiken erzielte Ergebnis in jedem Falle korrekt ist, unterstützen Taktiken in besonderem Maße das Experimentieren mit neuen Ideen und Strategien in einer Art, die keinerlei Eingriffe in das eigentliche Beweissystem verlangt. In einem gewissen Sinne sind Taktiken also *konservative Erweiterungen* des Systems der Inferenzregeln und bieten zusammen mit Abstraktionen einen sicheren Weg, die praktische Ausdruckskraft des Kalküls beliebig zu erweitern.

4.2.6 Erfahrungen im praktischen Umgang mit Taktiken

Der Taktik-Mechanismus hat sich im Laufe der Jahre als ein sehr erfolgreiches Hilfsmittel zur Einbettung einfacher Beweisstrategien in ein interaktives Beweisentwicklungssystem erwiesen. Viele Anwender des NuPRL Systems haben eine Reihe universeller (siehe vor allem [Howe, 1988] und [Jackson, 1993b, Kapitel 8]) und anwendungsspezifischer Taktiken geschrieben und dazu benutzt, um bestimmte mathematische Teilgebiete formal zu verifizieren [Howe, 1986, Howe, 1987, Howe, 1988, Kreitz, 1986, Basin, 1989, de la Tour & Kreitz, 1992]. Der Taktik-Mechanismus hat hierbei ermöglicht, in wenigen Tagen die gleichen Strategien zu realisieren, deren Implementierung in anderen Systemen mehrere Wochen intensive Arbeit gekostet hatte, weil Eingriffe in das eigentliche Beweissystem vorgenommen werden mußten. Taktiken sind leichter zu verstehen, da sie nahezu direkt auf der Ebene der Objekttheorie operieren, und können somit auch leichter modifiziert und zusammengesetzt werden als direktere Implementierungen von Strategien.

[24] Dies setzt natürlich voraus, daß die Regeln des Kalküls und die Funktion `refine` korrekt implementiert wurden. Wenn dies aber nicht der Fall wäre, dann wäre das gesamte System nutzlos.

Besonders wichtig ist, daß Taktiken bei aller Flexibilität ein hohes Maß an Sicherheit bieten und sich somit für Experimente mit neuen strategischen Ideen besonders eignen. Insbesondere öffnen sie eine Möglichkeit, alle bekannten Verfahren des automatischen Beweisens und der Programmsynthese in *ein einziges Beweissystem* zu integrieren. Diese Verfahren können nämlich als eine Art *Beweisplaner* verstanden werden, deren Resultate dann zur Steuerung der Beweisregeln verwandt werden kann.[25] Der wachsende Erfolg taktischer Beweissysteme (siehe Fußnote 3 auf Seite 182), insbesondere der des universellen generischen Systems ISABELLE demonstriert in besonderem Maße die praktischen Vorteile des Taktik-Konzeptes gegenüber Systemen mit festeingebauten Strategien.

Im Prinzip gibt es keine Grenzen für das, was man mit Taktiken programmieren kann. Die Versuchung, Taktiken zu schreiben, die möglichst alle Arbeiten automatisch durchführen, ist daher sehr groß. Dennoch sollte man sich darauf beschränken, Taktiken zu programmieren, deren Effekte überschaubar und kontrollierbar[26] sind, da sehr universelle Taktiken oft Irrwege einschlagen und Beweise erheblich größer werden lassen, als dies der Fall sein müßte. Es ist ratsam, stattdessen kleine und effiziente Taktiken mit einem sehr begrenzten Anwendungsbereich zu schreiben und diese gezielt einzusetzen. Hierbei ist jedoch auf eine sinnvolle und leicht verständliche Namensgebung zu achten, da ansonsten eine unüberschaubare Fülle von Taktiken die Auswahl einer guten Taktik in einer konkreten Situation erschwert.

Bei komplexen Anwendungen kann die Anwendung von Taktiken recht ineffizient werden, da sie alle elementaren Beweisschritte einzeln ausführen müssen. In diesem Falle erweist es sich als sinnvoll, die wichtigsten Eigenschaften der betrachteten Objekte zunächst als Lemmata (Theoreme der Bibliothek) zu verifizieren und dann in Taktiken auf diese Lemmata zurückzugreifen. Damit reduziert sich die Beweislast zur Laufzeit der Taktik auf eine Instantiierung universell-quantifizierter Variablen des Lemmas, während die Hauptlast ein für alle Mal bei dem Beweis des Lemmas durchgeführt wurde. Diese Vorgehensweise steigert nicht nur die Effizienz der Beweisführung sondern auch die Verständlichkeit der erzeugten Beweise, da diese nun in großen konzeptionellen Schritten strukturiert werden können.[27] Somit kann der Taktik Mechanismus, gekoppelt mit Abstraktionen und Theoremen der Bibliothek dazu benutzt werden, das Niveau des logischen Schließens schrittweise von den Grundbestandteilen des zugrundeliegenden Kalküls auf die typische Denkweise der zu bearbeitenden Probleme anzuheben und somit die formal korrekte Lösung komplexerer Anwendungen erheblich zu erleichtern.

4.3 Entscheidungsprozeduren – automatische Beweisführung

Im formalen Schließen tauchen relativ oft Teilprobleme auf, die im Prinzip leicht zu beweisen sind, weil sie auf bekannten mathematischen Erkenntnissen beruhen. Dennoch ist ihr formaler Beweis selbst beim Einsatz von Taktiken oft sehr aufwendig obwohl er keinerlei algorithmische Bedeutung besitzt (also im wesentlichen Axiom als Extraktterm liefert). Das Hauptinteresse bei der Bearbeitung solcher Probleme liegt also darin, herauszufinden, ob die aufgestellte Behauptung wahr ist oder nicht, während die einzelnen Beweisschritte keine signifikante Bedeutung[28] besitzen.

Es ist daher durchaus sinnvoll, für manche Problemstellungen schnelle Algorithmen zu entwerfen, welche diese in einer Weise entscheiden, die für einen normalen Benutzer nicht unbedingt einsichtig ist. Derartige *Entscheidungsprozeduren* beruhen meist auf einer maschinennahen Charakterisierung für die Gültigkeit der

[25] Dieses Konzept wird sehr erfolgreich von dem System OYSTER [Bundy, 1989, Bundy *et.al.*, 1990] verfolgt.

[26] Nicht alle in Taktiken, die derzeit in NuPRL integriert sind, erfüllen diese Bedingungen. Zugunsten einer einfacheren Handhabung enthält das System einige Taktiken, die Wohlgeformtheitsziele möglichst vollständig abhandeln. Wenn diese allerdings fehlschlagen, findet der Benutzer oftmals recht unverständliche Teilziele vor.

[27] Die Taktiken zur Instantiierung von Lemmata in [Jackson, 1993b, Kapitel 8.5] und die in [Jackson, 1993b, Kapitel 8.8]) beschriebenen Taktiken zur automatischen Konversion von Beweiszielen mit Gleichheitslemmata unterstützen diese Methodik.

[28] Sie tragen weder zum Extraktterm des Gesamtbeweises bei noch liefern sie neue mathematische Einsichten, da es sich um ein längst erforschtes Teilgebiet handelt.

Behauptung, die durch eine grundlegenden allgemeinen Analyse der Problemstellung zustandegekommen ist[29] und sich leichter überprüfen läßt, als die ursprüngliche Behauptung. Nach außen hin erscheinen sie als eine einzige ausgeklügelte Inferenzregel, die Axiom als Extraktterm liefert, aber ansonsten nicht mehr durch ein einfaches Schema beschrieben werden kann.

Es ist offensichtlich, daß eine solche Entscheidungsprozedur nur dann zu einem Beweissystem hinzugenommen werden darf, wenn ihre Konsistenz mit den anderen Regeln der zugrundeliegenden Theorie bewiesen werden kann. Da die Typentheorie und viele darin enthaltene Teiltheorien (wie die Prädikatenlogik oder das Induktionsbeweisen) nachweislich unentscheidbar sind, müssen wir uns auf Entscheidungsprozeduren für Teiltheorien beschränken, die bekanntermaßen entscheidbar sind und eine wichtige Rolle in der Praxis des mathematischen Schließens spielen. Da Entscheidungsprozeduren aufgrund ihrer Verwendung innerhalb von interaktiven Beweissystemen sehr schnell arbeiten sollen, muß außerdem sehr leicht feststellbar sein, ob die Prozedur überhaupt anwendbar ist und welche Hypothesen für sie relevant sind. Dies schränkt die Menge der Teiltheorien, für die es sinnvoll ist, Entscheidungsprozeduren zu entwerfen, relativ stark ein.

In diesem Abschnitt werden wir zwei Entscheidungsprozeduren vorstellen, die erfolgreich zur Typentheorie von NuPRL hinzugefügt werden konnten und daran illustrieren, welche Schritte bei der Entwicklung einer Entscheidungsprozedur eine Rolle spielen.

4.3.1 Eine Entscheidungsprozedur für elementare Arithmetik

Die Erfahrung im Umgang mit Beweissystemen hat gezeigt, daß praktische Arbeiten ohne eine arithmetische Entscheidungsprozedur so gut wie undurchführbar sind. Mangels einer solchen Prozedur ist das LCF Projekt [Gordon *et.al.*, 1979], welches sich das Schließen über Berechnungen zum Ziel gesetzt hatte, ebenso in seinen Anfängen steckengeblieben wie das Projekt AUTOMATH [Bruijn, 1980, van Benthem Jutting, 1977], in dem mathematische Beweise formalisiert und automatisch überprüft werden sollten: es dauerte über 5 Jahre intensiver Arbeit bis man die Ringaxiome und einige andere elementare Eigenschaften der reellen Zahlen beweisen konnte, nur weil für jeden dieser Beweise eine extrem große Anzahl von Beweissschritten nötig war.

Warum nun ist eine Entscheidungsprozedur die Arithmetik so bedeutend? Im wesentlichen kann man vier Ursachen dafür finden.

1. In fast allen praktisch relevanten Beweisen spielt arithmetisches Schließen eine Rolle.
2. Ein Großteil der arithmetischen Schlüsse ist absolut trivial, was die gewonnenen Erkenntnisse angeht.
3. Ein und dieselbe arithmetische Aussage kann in einer unglaublichen Vielfalt von Erscheinungsformen auftauchen. So sind zum Beispiel `x+1<y, 0<t ⊢ (x+1)*t < y*t` und `x<y, 0<t ⊢ x*t < y*t` syntaktisch völlig verschiedene Behauptungen obwohl die zweite Aussage eine einfache Variante der ersten ist.
4. Der formale Beweis einer simplen arithmetischen Aussage mit Hilfe der elementaren Inferenzregeln der ganzen Zahlen ist keineswegs so einfach, wie es auf den ersten Blick erscheinen mag. So ist die Aussage

   Wenn drei ganze Zahlen sich jeweils um maximal 1 unterscheiden, dann sind zwei von ihnen gleich

   intuitiv besehen absolut einsichtig, kann formal aber nur relativ aufwendig bewiesen werden.

Es gibt also gute Gründe dafür, eine Entscheidungsprozedur für die Arithmetik zu entwerfen, die derartige Probleme in *einem* Inferenzschritt lösen kann und nach außen wie eine einfache Inferenzregel erscheint. Bevor wir dies tun können, müssen wir allerdings die theoretischen Grenzen einer solchen Entscheidungsprozedur untersuchen und eine maschinennahe Charakterisierung für die Gültigkeit arithmetischer Aussagen aufstellen

[29] So kann man die Gültigkeit einer arithmetischen Aussage auf ein graphentheoretisches Problem zurückführen (siehe Abschnitt 4.3.1, und die Gültigkeit einer logischen Aussage durch eine Matrixcharakterisierung [Bibel, 1983, Bibel, 1987] beschreiben. Auch für geometrische Probleme gibt es eine sehr effiziente Charakterisierung [Wu, 1986, Chou & Gao, 1990].

und als korrekt nachweisen. Auch der Algorithmus selbst muß schließlich als korrekt nachgewiesen werden, da wir sonst keinerlei Veranlassung mehr hätten, einem mechanisch geführten Beweis zu vertrauen. Wir werden in diesem Abschnitt eine relativ kurze und wenig formale Beschreibung der Entscheidungsprozedur `arith` geben, die in das NuPRL-System integriert ist und verweisen auf [Constable *et.al.*, 1982][30] für weitere Details.

Welche Art von Problemen kann man mit `arith` lösen und welche nicht? Wir wollen hierzu ein paar Beispiele geben.

Beispiel 4.3.1

Die folgenden Problemstellungen können wie folgt mit `arith` gelöst werden.

Γ, i: 0<x, Δ	$\vdash$ 0<x+2	**by** arith
Γ, i: 0<x, Δ	$\vdash$ 0<x*x	**by** arith i * i
Γ, i: x+y≤z, Γ', j: y≥1, Δ	$\vdash$ x<z	**by** arith i - j
Γ, i: x≤y, Γ', j: x≠y, Δ	$\vdash$ x<y	**by** arith
Γ	$\vdash$ x-5 < x+10	**by** arith
Γ, i: x<x*x, Γ', j: x≠0 , Δ	$\vdash$ x≥2 $\vee$ x<0	**by** arith i ÷ j
Γ, i: x<y, Γ', j: 0<z, Δ	$\vdash$ x*z<y*z	**by** arith i * i
Γ, i: x+y>z, Γ', j: 2*x≥z, Δ	$\vdash$ 3*x+y ≥ 2*z-1	**by** arith i + i

All diese Probleme wären ohne `arith` nur sehr aufwendig zu beweisen und zeigen die praktischen Vorteile dieser Prozedur. Es gibt jedoch auch Grenzen für die Einsatzmöglichkeiten arithmetischer Entscheidungsprozeduren, da nicht alle arithmetischen Probleme entscheidbar sind.

Beispiel 4.3.2 (Das 10. Hilbertsche Problem)

Ist f eine *beliebige* berechenbare Funktion auf den ganzen Zahlen, so gibt es keine Möglichkeit zu entscheiden, ob f eine Nullstelle besitzt oder nicht. Das Problem

$\exists x_1,\ldots,x_n{:}\mathbb{Z}.\ f(x_1,\ldots,x_n)=0$

ist nicht durch ein universelles Verfahren (das keine Spezialkenntnisse über f besitzt) zu entscheiden.

Dieses Beispiel zeigt, daß wir eine Entscheidungsprozedur für die gesamte Arithmetik nicht konstruieren können. Wir können also bestenfalls erwarten, eine *eingeschränkte Theorie der Arithmetik* entscheiden zu können, die aber immer noch die meisten realen Probleme erfassen kann. Arithmetische Probleme, die mit den Mitteln dieser eingeschränkten Theorie nicht mehr beschrieben werden können, müssen auf andere Weise (z.B. durch eine Taktik) behandelt werden. Dies ist aus praktischer Hinsicht aber kein gravierender Nachteil, da wir vor allem ja die trivialen Fälle durch eine Entscheidungsprozedur lösen wollen und nicht etwa anstreben, einen universellen Beweiser zu konstruieren. Wir werden im folgenden also eine eingeschränkte Theorie der Arithmetik ohne Induktion aufstellen, die mit Sicherheit entscheidbar ist. Für diese Theorie werden wir eine Charakterisierung der Gültigkeit aufstellen und auf dieser Basis dann eine Entscheidungsprozedur angeben.

Abbildung 4.5 beschreibt die komplette Syntax und Semantik einer eingeschränkten arithmetischen Theorie, die wir im folgenden als *Theorie der elementaren Arithmetik* $\mathcal{A}$ bezeichnet. Zur Vereinfachung haben wir die Theorie als *quantorenfreie Arithmetik* formuliert und gehen davon aus, daß alle vorkommenden Variablen all-quantifiziert und vom Typ $\mathbb{Z}$ sind.[31] Es ist leicht einzusehen, daß $\mathcal{A}$ eine Teiltheorie der intuitionistischen Typentheorie ist und somit von NuPRL verarbeitet werden kann.

[30] Die `arith`-Prozedur ist 1976 im Zusammenhang mit dem PL/CV System entstanden und erfolgreich in einer früheren Version von PRL (λ-PRL) eingesetzt worden, die noch nicht auf Typentheorie basierte. Eine ausführliche Beschreibung und einen Korrektheitsbeweis liefert der Artikel "*An algorithm for checking PL/CV arithmetic inferences*" von Tat-hung Chan, der im Appendix von [Constable *et.al.*, 1982] wiedergegeben ist. Die Einbettung in NuPRL verlangte eine Reihe kleinerer Anpassungen, die in diesem Artikel nicht erwähnt sind, aber keine signifikante Rolle spielen.

Man beachte, daß alle Beweise, die mit `arith` gelöst werden können, im Prinzip auch mit den normalen Regeln der Typentheorie bewiesen werden könnten, wobei man natürlich spezielle Regeln für Konstantenarithmetik und alle anderen explizit definierten Ausdrücke benötigen würde (die ja nicht mehr auf Nachfolger und Induktion abgestützt sind). Die Hinzunahme von `arith` anstelle eines "konventionellen" Regelsatzes geschieht aus rein pragmatischen Gründen und verändert die Typentheorie als solche überhaupt nicht.

[31] In der `arith`-Prozedur wirkt sich das so aus, daß alle symbolischen Ausdrücke, die bei der Verarbeitung wie Variablen behandelt werden, nach erfolgreicher Ausführung von `arith` als vom Typ $\mathbb{Z}$ nachgewiesen werden müssen.

Formale Sprache: Die formale Sprache von $\mathcal{A}$ besteht aus elementar-arithmetischen Formeln:

- *Terme* sind NuPRL-Terme, die nur aus ganzzahligen Konstanten, Variablen und den Operatoren `+, -, *` aufgebaut sind.
- *Atomare Formeln* sind Terme der Form $t_1 \rho t_2$, wobei t_1 und t_2 Terme sind und ρ einer der arithmetischen Vergleichsoperatoren $=, \neq, <, >, \leq, \geq$ ist.
- *elementar-arithmetische Formeln* sind alle Ausdrücke, die sich aus atomaren Formeln und den aussagenlogischen Konnektiven $\neg$, $\wedge$, $\vee$ und $\Rightarrow$ aufbauen lassen.

Semantik: Die Semantik von $\mathcal{A}$ wird durch (eingeschränkte) Gleichheitsaxiome und die gewöhnlichen Axiome der Zahlentheorie charakterisiert. Im einzelnen sind dies:

Gleichheitsaxiome. Für alle ganzen Zahlen `x,y,z` gilt:

1. `x=x` (Reflexivität)
2. `x=y` $\Rightarrow$ `y=x` (Symmetrie)
3. `x=y` $\wedge$ `y=z` $\Rightarrow$ `x=z` (Transitivität)
4. `x=y` $\wedge$ `x` ρ `z` $\Rightarrow$ `y` ρ `z` `x=y` $\wedge$ `z` ρ `x` $\Rightarrow$ `z` ρ `y` (eingeschränkte Substitutivität)[a]

Axiome der Konstantenarithmetik wie `1+1=2, 2+1=3, 3+1=4, ...`[b]

Ringaxiome der ganzen Zahlen. Für alle ganzen Zahlen `x,y,z` gilt:

1. `x+y=y+x` `x*y=y*x` (Kommutativgesetze)
2. `(x+y)+z=x+(y+z)` `(x*y)*z=x*(y*z)` (Assoziativgesetze)
3. `x*(y+z)=(x*z)+(y*z)` (Distributivgesetz)
4. `x+0=x` `x*1=x` (Neutrale Elemente)
5. `x+(-x)=0` (Inverses Element der Addition)
6. `x-y=x+(-y)` (Definition der Subtraktion)

Axiome der diskreten linearen Ordnung. Für alle ganzen Zahlen `x,y,z` gilt:

1. $\neg$`(x<x)` (Irreflexivität)
2. `x<y` $\vee$ `x=y` $\vee$ `y<x` (Trichotomie)
3. `x<y` $\wedge$ `y<z` $\Rightarrow$ `x<z` (Transitivität)
4. $\neg$`(x<y` $\wedge$ `y<x+1)` (Diskretheit)

Definition von Ordnungsrelationen und Ungleichheiten. [c] Für alle ganzen Zahlen `x,y,z` gilt:

1. `x`$\neq$`y` $\Leftrightarrow$ $\neg$`(x=y)`
2. `x`$>$`y` $\Leftrightarrow$ `y<x`
3. `x`$\leq$`y` $\Leftrightarrow$ `x<y` $\vee$ `x=y`
4. `x`$\geq$`y` $\Leftrightarrow$ `y<x` $\vee$ `x=y`

Monotonieaxiome. Für alle ganzen Zahlen `x,y,z,w` gilt:

1. `x`$\geq$`y` $\wedge$ `z`$\geq$`w` $\Rightarrow$ `x+z`$\geq$`y+w` (Addition)
2. `x`$\geq$`y` $\wedge$ `z`$\leq$`w` $\Rightarrow$ `x-z`$\geq$`y-w` (Subtraktion)
3. `x`$\geq$`0` $\wedge$ `y`$\geq$`z` $\Rightarrow$ `x*y`$\geq$`x*z` (Multiplikation)
4. `x`$>$`0` $\wedge$ `x*y`$\geq$`x*z` $\Rightarrow$ `y`$\geq$`z` (Faktorisierung)

Sind `z` und `w` Konstanten, dann werden die Monotonoieaxiome der Addition und Subtraktion auch als *triviale Monotonien* bezeichnet.

Abbildung 4.5: Die Theorie $\mathcal{A}$ der elementaren Arithmetik

[a] Es sind also nur *geschlossene* Substitutionen zugelassen: aus `x=z` und `x`$\neq$`x*y` folgt `z`$\neq$`x*y`, nicht aber `z`$\neq$`z*y`.

[b] Entsprechende Gesetze mit größeren Konstanten wie `4+9=13` und `4*9=36` folgen hieraus mit den Ringaxiomen.

[c] In der NuPRL-Version von `arith` werden anstelle der Relationen $\neq$, $\leq$, $\geq$, `>` die entsprechenden rechten Seiten der Definition 3.4.5 auf Seite 145 verarbeitet.

Addition

	z>w	z≥w	z=w	z≠w
x>y	x+z≥y+w+2	x+z≥y+w+1	x+z≥y+w+1 x+w≥y+z+1	-----
x≥y	x+z≥y+w+1	x+z≥y+w	x+z≥y+w x+w≥y+z	-----
x=y	x+z≥y+w+1 y+z≥x+w+1	x+z≥y+w y+z≥x+w	x+z=y+w x+w=y+z	x+z≠y+w x+w≠y+z
x≠y	-----	-----	x+z≠y+w x+w≠y+z	-----

Multiplikation

	y≥z	y>z	y=z	y≠z
x>0	x*y≥x*z	x*y>x*z	x*y=x*z	x*y≠x*z
x≥0	xy*≥x*z	x*y≥x*z	x*y=x*z	-----
x=0	x*y=x*z x*y=0	x*y=x*z x*y=0	x*y=x*z x*y=0	x*y=x*z x*y=0
x≤0	x*y≤x*z	x*y≤x*z	x*y=x*z	-----
x<0	x*y≤x*z	x*y<x*z	x*y=x*z	x*y≠x*z
x≠0	-----	x*y≠x*z	x*y=x*z	x*y≠x*z

Subtraktion

	z>w	z≥w	z=w	z≠w
x>y	x-w≥y-z+2	x-w≥y-z+1	x-w≥y-z+1 x-z≥y-w+1	-----
x≥y	x-w≥y-z+1	x-w≥y-z	x-w≥y-z x-z≥y-w	-----
x=y	x-w≥y-z+1 y-w≥x-z+1	x-w≥y-z y-w≥x-z	x-w=y-z y-w=x-z	x-w≠y-z x-z≠y-w
x≠y	-----	-----	x-w≠y-z x-z≠y-w	-----

Faktorisierung

	x*y>x*z	x*y≥x*z	x*y=x*z	x*y≠x*z
x>0	y>z	y≥z	y=z	y≠z
x<0	y<z	y≤z	y=z	y≠z
x≠0	y≠z	-----	y=z	y≠z

Abbildung 4.6: Varianten der Monotoniegesetze von $\mathcal{A}$

Weiterhin gilt, daß die klassische und konstruktive Interpretation der elementar-arithmetischen Formeln übereinstimmen, da nur einfache arithmetische Operatoren (+, -, *), die elementaren Vergleichsoperatoren $=, \neq, <, >, \leq, \geq$ und die aussagenlogischen Konnektive ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) zugelassen sind. Durch eine Induktion über die Termstruktur kann man daher beweisen, daß alle atomaren Formeln in $\mathcal{A}$ entscheidbar sind. Hieraus folgt wiederum die Entscheidbarkeit aller elementar-arithmetische Formeln, da Entscheidbarkeit von Formeln unter $\neg$, $\wedge$, $\vee$ und $\Rightarrow$ abgeschlossen ist. Insgesamt gilt also der folgende Satz.

Satz 4.3.3

Die Theorie der elementaren Arithmetik $\mathcal{A}$ ist entscheidbar.

Der (formal etwas aufwendige) Beweis von Satz 4.3.3 liefert im Prinzip bereits ein Verfahren, mit dem die Gültigkeit aller elementar-arithmetischen Aussagen entschieden werden kann, das allerdings sehr ineffizient ist. Um eine wirklich effiziente Entscheidungsprozedur für elementar-arithmetischen Aussagen zu entwickeln, ist es nötig, viele Varianten der Axiome von $\mathcal{A}$ und die üblichen Berechnungsverfahren für die Auswertung elementar-arithmetischer Terme direkt in den Entscheidungsalgorithmus einzubetten. Diese Varianten (modulo der Ringaxiome) sind in den Tabellen von Abbildung 4.6 aufgeführt, deren Zeilen und Spalten durch *Termschemata* indiziert sind. Jeder Tabelleneintrag beschreibt die Schlußfolgerungen, die sich aus den in den Indizes dargestellten Hypothesen ergeben.

Eine weitere wesentliche Vereinfachung ergibt sich daraus, daß aufgrund der Entscheidbarkeit von $\mathcal{A}$ jede elementar-arithmetische Formel auf eine konjunktive Normalform gebracht werden kann.

Lemma 4.3.4 (Konjunktive Normalform elementar-arithmetischer Formeln)

Zu jeder elementar-arithmetischen Formel F gibt es eine äquivalente elementar-arithmetische Formel G der Gestalt $(G_{11} \vee \ldots \vee G_{1n_1}) \wedge \ldots \wedge (G_{m1} \vee \ldots \vee G_{1n_m})$, wobei alle G_{ij} atomare Formeln in $\mathcal{A}$ sind.

Beweis: Aufgrund der Entscheidbarkeit von $\mathcal{A}$ kann jede elementar-arithmetische Formel mit dem aus der klassischen Aussagenlogik bekannten Verfahren in eine Formel der Gestalt $(F_{11} \vee \ldots \vee F_{1n_1}) \wedge \ldots \wedge (F_{m1} \vee \ldots \vee F_{1n_m})$ umgewandelt werden, wobei die F_{ij} atomare Formeln oder negierte atomare Formeln sind. Negierte atomare Formeln lassen sich wiederum durch Verwendung der 'negierten' Vergleichsoperatoren in atomare Formeln umwandeln (also z.B. $\neg$(x≤y) in x>y). □

Da nun eine Formel der Gestalt $(G_{11} \vee \ldots \vee G_{1n_1}) \ \wedge \ \ldots \wedge \ (G_{m1} \vee \ldots \vee G_{1n_m})$ genau dann gültig ist, wenn jede einzelne Klausel dieser Formel allgemeingültig ist, können wir jede dieser Klauseln separat behandeln. Es reicht daher aus, daß die Entscheidungsprozedur in der Lage ist, Beweisziele der Form

$$\Gamma \vdash G_1 \vee \ldots \vee G_n$$

zu verarbeiten, wobei die G_i elementar-arithmetische Formeln sind und der Sonderfall $n = 0$ bedeutet, daß die Konklusion Λ ist (da keine Alternative gegeben wird).

Im folgenden wollen wir die Entscheidungsprozedur `arith` schrittweise anhand eines Beispiels entwickeln und anschließend die hierbei verwendeten Erkenntnisse als Satz festhalten. Wir betrachten das Beweisziel

Γ, i :x+y>z, Γ', j :2*x≥z, Δ ⊢ 3*x+y≥2*z-1

In einem ersten vorverarbeitenden Schritt[32] können neue Hypothesen ergänzt werden, die sich aus bestehenden Hypothesen aufgrund der Monotoniegesetze ergeben. Dieser Schritt muß durch den Benutzer gesteuert werden, der angibt, wie die neue Hypothese durch arithmetische Operationen aus den bereits bestehenden zu erzeugen ist. Dabei kann man zwei Hypothesen aufaddieren, subtrahieren, multiplizieren und dividieren (beschränkt auf Faktorisierung), sofern hierdurch Schlußfolgerungen aufgrund der Monotoniegesetze in Abbildung 4.6 gezogen werden können. Hypothesen, die sich aufgrund *trivialer Monotonien* aus anderen Hypothesen ergeben, brauchen nicht explizit generiert zu werden, da dies im Verlaufe von `arith` automatisch geschieht.

Definition 4.3.5

Eine triviale Monotonie ist die Anwendung eines Monotonieaxioms für Addition oder Subtraktion auf zwei Prämissen, von denen eine die Gestalt $n \,\rho\, m$ hat, wobei m und n ganzzahlige Konstanten und ρ ein arithmetischer Vergleichsoperator ist. Jede andere Anwendung von Monotonieaxiomen ist nichttrivial.

Intuitiv betrachtet ist die Anwendung einer trivialen Monotonie also das Aufaddieren von Konstanten auf beiden Seiten eines elementaren arithmetischen Vergleichs, sofern danach noch ein Vergleich gezogen werden kann. So können wir zum Beispiel aus `x=y` folgern, daß `x+2≠y+4` gilt, wissen aber nichts über das Verhältnis von `x+2` and `y+4`, falls `x≠y` gilt.

Beispiel 4.3.6

1. Im Falle des Beweisziels Γ, i :x+y>z, Γ', j :2*x≥z, Δ ⊢ 3*x+y≥2*z-1 gibt es zunächst nur nichttriviale Monotonien, von denen wir die Addition der Hypothesen i und j ausnutzen wollen. Auf diese Art erhalten wir eine Hypothese, in der – nach Vereinfachungen – links `3*x+y` und rechts `2*z` vorkommen wird. Um dies zu erreichen, muß die Entscheidungsprozedur `arith` mit dem Parameter i+j aufgerufen werden. Entsprechend der Additionstabelle in Abbildung 4.6 erhalten wir somit als Ausgangspunkt für das eigentliche Entscheidungsverfahren

   Γ, i :x+y>z, Γ', j :2*x≥z, Δ, x+y+2*x≥z+z+1 ⊢ 3*x+y≥2*z-1

2. Da in der elementaren Arithmetik klassisches und intuitionistisches Schließen übereinstimmt, ist es legitim, auf eine *negative Darstellung des Problems* überzugehen, d.h. anzunehmen, daß die Konklusion falsch wäre und hieraus einen Widerspruch zu folgern. Dieser Wechsel der Darstellung ist ein Standardtrick im Theorembeweisen, denn er reduziert das ursprüngliche Problem auf die Frage, eine sich selbst widersprechende Menge von Hypothesen zu finden. Dieser Schritt, dessen Rechtfertigung in Theorem 4.3.7.1 gegeben wird, führt zu der Sequenz

   Γ, x+y>z, Γ', 2*x≥z, Δ, x+y+2*x≥z+z+1, ¬(3*x+y≥2*z-1) ⊢ Λ

3. Hypothesen, die keine elementar-arithmetischen Formeln enthalten, können problemlos ignoriert werden, da sie auf keinen Fall zu einem (arithmetischen) Beweis beitragen werden. Dies führt dazu, daß Γ, Γ' und Δ aus der Hypothesenliste verschwinden.

[32] In der derzeitigen Implementierung ist dieser Vorverarbeitungsschritt nicht in der eigentlichen `arith`-Prozedur enthalten sondern muß separat durchgeführt werden.

4. Im nächsten Schritt werden die Komparanden (d.h. die Terme links und rechts von einem Vergleichsoperator) jeder Hypothese auf eine Normalform gebracht. Hierzu verwendet man die Standarddarstellung von Polynomen als Summen von Produkten $a*x_1^n*x_2^n\ldots+\ldots$. Diese symbolische Normalisierung ist eine einfache arithmetische Umformung, welche die Gültigkeit einer Sequenz nicht beeinflußt, aber dafür sorgt, daß semantisch gleiche Terme auch die gleiche syntaktische Gestalt erhalten. In unserm Beispiel führt dies zu

   $\texttt{x+y}>\texttt{z},\ 2*\texttt{x}\geq\texttt{z},\ 3*\texttt{x+y}\geq 1+2*\texttt{z},\ \neg(3*\texttt{x+y}\geq(-1)+2*\texttt{z}) \ \vdash\ \Lambda$

5. Nun werden alle Komparanden in *monadische lineare Polynome* transformiert, also in Ausdrücke der Form $\texttt{c + }u_i$, wobei u_i eine (neue) Variable ist. Dies bedeutet, daß ab jetzt alle nichtkonstanten Komponenten eines Terms wie eine Variable behandelt werden, wobei gleiche Komponenten natürlich durch die gleiche Variable repräsentiert werden. Dieser Schritt basiert auf Theorem 4.3.7.2 (was relativ mühsam zu beweisen ist) und führt zu

   $\texttt{u}_0>\texttt{z},\ \texttt{u}_1\geq\texttt{z},\ \texttt{u}_2\geq 1+\texttt{u}_3\ \neg(\texttt{u}_2\geq(-1)+\texttt{u}_3) \ \vdash\ \Lambda$

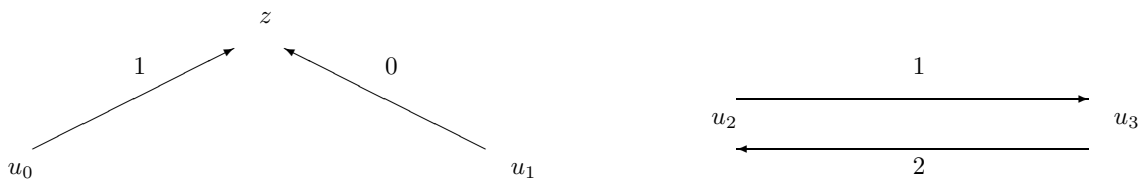
   Hierbei wurde $\texttt{u}_0\equiv\texttt{x+y},\ \ \texttt{u}_1\equiv 2*\texttt{x},\ \ \texttt{u}_2\equiv 3*\texttt{x+y}$ und $\texttt{u}_3\equiv 2*\texttt{z}$ gewählt.

6. Die nächste Normalisierung betrifft die Vergleichsoperatoren. Alle Hypothesen werden transformiert in Formeln der Gestalt $t_1\geq t_2$, wobei die linke Seite t_1 entweder eine Variable $\texttt{u}_i$ oder die Konstante `0` ist und t_2 ein monadisches lineares Polynom. Dies läßt sich leicht durch Verwendung von Konversionstabellen[33] erreichen, kann allerdings dazu führen, daß neue Disjunktionen atomarer Formeln entstehen und somit mehrere Alternativen betrachtet werden müssen. In unserem Fall lautet das Resultat

   $\texttt{u}_0\geq\texttt{z+1},\ \texttt{u}_1\geq\texttt{z},\ \texttt{u}_2\geq 1+\texttt{u}_3\ \texttt{u}_3\geq\texttt{u}_2+2 \ \vdash\ \Lambda$

   Wenn nach diesem Schritt Disjunktionen atomarer Formeln in den Hypothesen auftreten, so ist es nötig, diese zu zerlegen und jeden dieser Fälle separat zu betrachten. Im schlimmsten Fall kann dies dazu führen, daß im nächsten Schritt exponentiell viele Alternativen (gemessen an der Anzahl der ursprünglich vorhandenen Ungleichheiten) untersucht werden müssen.[34]

7. Im vorletzten Schritt wird die Sequenz in einen Graphen umgeformt, welcher die Ordnungsrelation zwischen den Variablen beschreibt. Jeder Knoten dieses Graphen repräsentiert eine Variable oder eine Konstante und eine Kante $u_i \xrightarrow{c} u_j$ repräsentiert die Ungleichung $\texttt{u}_i\geq\texttt{u}_j+c$. In unserem Beispiel führt dies zu folgendem Graphen



   Dieser Graph besitzt (gemäß Theorem 4.3.7.3) einen positiven Zyklus, wenn die ursprüngliche Formelmenge widersprüchlich war. Mit einem graphentheoretischen Standardalgorithmus (siehe zum Beispiel [Constable *et.al.*, 1982, Seite 241–242]) läßt sich nun leicht überprüfen, ob dies der Fall ist.

   In obigem Graphen gibt es einen Zyklus mit den Knoten $\texttt{u}_2$ und $\texttt{u}_3$. Dies bedeutet, daß die Hypothesenmenge des letzten Schrittes widersprüchlich und somit die ursprüngliche Sequenz allgemeingültig war. Die Prozedur `arith` schließt damit die Überprüfung der Formel erfolgreich ab und hinterläßt als Teilziele, daß `x, y` und `z` als Elemente von $\mathbb{Z}$ nachzuweisen sind.

[33] Typische Konversionen, die hierbei zum Tragen kommen werden, sind zum Beispiel $\neg x\geq y \Leftrightarrow x<y,\ \ x>y \Leftrightarrow x\geq y+1,\ x=y \Leftrightarrow x\geq y\wedge y\geq x,\ \ x\neq y \Leftrightarrow x\geq y+1\vee y\geq x+1$, etc. sowie das beidseitige Aufaddieren von Konstanten, um negative Konstanten in Polynomen zu eliminieren.

[34] In der Praxis ist diese miserable Komplexitätsschranke allerdings nicht so bedeutend, da `arith` in den meisten Fällen nur eine relativ kleine Anzahl arithmetischer Hypothesen verarbeiten muß und von diesen nur sehr wenige auch Ungleichheiten sind, die zu einer Aufblähung in Alternativen führen. Schlüsse, die ausschließlich auf Ungleichheiten beruhen, sind auch für den Menschen keineswegs trivial.

Gegeben sei ein Beweisziel der Gestalt $\Gamma \vdash G_1 \vee \ldots \vee G_n$ (Λ, falls $n = 0$) wobei die G_i elementar-arithmetische Formeln sind und Γ beliebige Hypothesen enthält.
Der Algorithmus für die Regel `arith` i *op* j, wobei *op* entweder `+, -, *` oder $\div$ ist, geht wie folgt vor.

1. Führe die geforderten Monotonieschritte mit den Hypothesen i und j aus und erzeuge eine neue Hypothese gemäß den Einträgen in den Tabellen aus Abbildung 4.6 aus.
2. Transformiere die resultierende Sequenz $\Gamma' \vdash G_1 \vee \ldots \vee G_n$ in $\Gamma, \neg G_1, \ldots, \neg G_n \vdash \Lambda$.
3. Zerlege alle Konjunktionen entsprechend der Regel `and_e` in neue Einzelhypothesen.
4. Transformiere alle Ungleichungen der Form $x \neq y$ in die Disjunktion $x \geq y+1 \vee y \geq x+1$
5. Zerlege alle Disjunktionen in den Hypothesen entsprechend der Regel `or_e`. Alle entstehenden Beweisziele werden im folgenden separat betrachtet und müssen zum Erfolg führen.
6. Entferne alle Hypothesen, die keine atomaren elementar-arithmetischen Formeln sind.
7. Ersetze Teilterme der Komparanden, welche nicht aus `+, -, *` oder ganzzahligen Konstanten aufgebaut sind, durch (neue) Variablen, wobei gleiche Terme durch die gleiche Variable ersetzt werden.
8. Transformiere alle Komparanden einer Hypothese in die Standarddarstellung von Polynomen.
9. Transformiere alle Komparanden in monadische lineare Polynome `c +` u_i, indem jedes nichtkonstante Polynom p durch eine (neue) Variable u_i ersetzt wird.
10. Konvertiere alle Hypothesen in Ungleichungen der Gestalt $t_1 \geq t_2$, wobei t_1 eine Variable $\mathtt{u}_i$ oder die Konstante `0` und t_2 ein monadisches lineares Polynom ist.
11. Erzeuge den Ordnungsgraphen der entstandenen Formelmenge. Erzeuge Knoten für jede Variable und Konstante und eine Kante $u_i \xrightarrow{c} u_j$ für die Ungleichung $\mathtt{u}_i \geq \mathtt{u}_j + c$.
12. Teste, ob der Ordnungsgraph einen positiven Zyklus hat oder nicht. Im Erfolgsfall generiere Wohlgeformtheitsziele für jeden durch eine Variable repräsentierten Teilterm. Andernfalls erzeuge eine Ausnahme mit einer entsprechenden Fehlermeldung.

Abbildung 4.7: Die Entscheidungsprozedur `arith`

Über die eben beschriebenen Fähigkeiten hinaus ist die arithmetische Entscheidungsprozedur des NuPRL Systems in der Lage, auch nichtelementare arithmetische Ausdrücke zu verarbeiten, also arithmetische Vergleiche zu handhaben, deren Komparanden nicht nur aus `+, -, *` aufgebaut sind. Derartige Teilausdrücke werden von `arith` wie atomare Terme behandelt, die nicht weiter analysiert werden. Da `arith` keinerlei Wohlgeformtheitsüberprüfungen durchführt, müssen alle atomaren Terme – seien es nun Konstanten, Variablen oder komplexere nichtelementare arithmetische Ausdrücke – nach einer erfolgreichen Überprüfung der Gültigkeit einer Sequenz als Elemente von $\mathbb{Z}$ nachgewiesen werden. Entsprechende Behauptungen werden als Teilziele der `arith`-Regel generiert.

Abbildung 4.7 beschreibt das allgemeine Verfahren, welches beim Aufruf der `arith`-Regel ausgeführt wird. Zugunsten einer effizienteren Verarbeitung wird die Relation $\neq$ schon relativ früh aufgelöst und Disjunktionen frühzeitig als alternative Fälle behandelt. Es wird gerechtfertigt durch die Erkentnisse des folgenden Satzes.

Satz 4.3.7

1. *Sind $G_1, \ldots, G_n$ entscheidbare Aussagen so ist die Sequenz $\Gamma \vdash G_1 \vee \ldots \vee G_n$ genau dann allgemeingültig, wenn $\Gamma, \neg G_1, \ldots, \neg G_n \vdash \Lambda$ gültig ist.*
2. *Es seien $e_1, \ldots, e_k$ elementar-arithmetische Terme und $u_1, \ldots, u_k$ Variablen.*
   Eine Menge $F_1[e_1, .., e_k / u_1, .., u_k], \ldots, F_n[e_1, .., e_k / u_1, .., u_k]$ von (instantiierten) elementar-arithmetischen Formeln ist genau dann widersprüchlich, wenn die Menge $F_1, \ldots, F_n$ widersprüchlich ist.
3. *Es sei $\Gamma = v_1 \geq u_1 + c_1, \ldots, v_n \geq u_n + c_n$ eine Menge von atomaren arithmetischen Formeln, wobei die v_i und u_i Variablen (oder die Konstante `0`) und die c_i nichtnegative Konstanten sind, und $\mathcal{G}$ der Graph, welcher die Ordnungsrelation zwischen den Variablen von Γ beschreibt.*
   Dann ist Γ genau dann widersprüchlich, wenn $\mathcal{G}$ einen positiven Zyklus besitzt.

Es sei angemerkt, daß in der derzeitigen NuPRL Implementierung von `arith` der erste Schritt – die Monotonieoperation – nicht enthalten ist. Ausschließlich die trivialen Monotonien kommen im Verlaufe der im Schritt 10 beschriebenen Konversion zum Tragen. Es sind allerdings einige Erweiterungen von `arith` als Taktiken implementiert, die Definitionen von Teiltypen von $\mathbb{Z}$ auflösen und die entsprechenden Bereichsinformationen ebenfalls verarbeiten können. Weitere Informationen hierzu findet man in [Jackson, 1993b, Kapitel 8.7].

4.3.2 Schließen über Gleichheit

Schließen über Gleichheit ist das Problem zu verifizieren, daß eine Gleichheit in der Konklusion eines Beweisziels aus einer Reihe bekannter Gleichheiten in den Hypothesen folgt, also daß zum Beispiel

`f(f(a,b),b) = a` aus `f(a,b) = a`

oder

`g(a) = a` aus `g(g(g(a))) = a` und `g(g(g(g(a)))) = a`

folgt. Nahezu alle Problem, die in der Praxis auftauchen, und insbesondere ein formales Schließen über Programme und die von ihnen berechneten Werte verlangt ein solches Schließen über Gleichheiten.

Wir hatten in Abschnitt 2.2.7 bereits angesprochen, daß Schließen über Gleichheiten im wesentlichen aus einer Anwendung der (in Abbildung 3.7 auf Seite 115 formalisierten) Regeln der Reflexivität, Symmetrie, Transitivität und der Substitution besteht. Formale Beweise, die sich jedoch ausschließlich auf diese Regeln stützen, sind im Allgemeinfall sehr aufwendig, obwohl die dahinterstehenden Einsichten sehr gering sind. So ist für einen Menschen sehr leicht einzusehen, warum die obengenannten Gleichheitsschlüsse wirklich gültig sind, während der formale Beweis mehrere trickreiche Substitutionen enthält. Da mittlerweile bekannt ist, daß die spezielle Theorie der Gleichheit – die nur aus den Axiomen Reflexivität, Symmetrie, Transitivität und Substitution besteht – entscheidbar ist [Ackermann, 1954][35] ist es sinnvoll, eine Entscheidungsprozedur zum Schließen über Gleichheiten zu entwickeln, welche ein vielfaches Anwenden der elementaren Gleichheitsregeln *in einem einzigen Schritt* durchführt, und diese als Inferenzregel in das Beweisentwicklungssystem mit aufzunehmen.

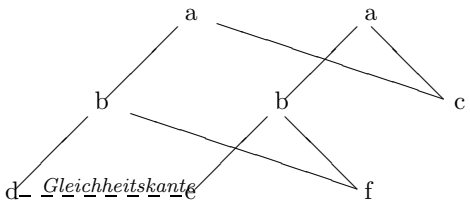
Es gibt eine Reihe guter Algorithmen, die für diesen Zweck eingesetzt werden können. Im folgenden werden wir ein Verfahren vorstellen, welches auf Arbeiten von Greg Nelson and Derek Oppen [Nelson & Oppen, 1979, Nelson & Oppen, 1980] basiert und als Kern der `equality`-Regel (siehe Abbildung 3.28 auf Seite 175) implementiert wurde. Die Schlüsselidee ist hierbei, in einem Graphen die transitive Hülle einer Relation zu konstruieren und hieraus dann abzuleiten, ob zwei Elemente in der gewünschten Relation zueinander stehen. Im Prinzip ist dieses Verfahren nicht nur zur Behandlung von Gleichheitsfragen geeignet, sondern kann ebenfalls dazu benutzt werden, um über Listenstrukturen und ähnliche Problemstellungen zu schließen. Wir wollen den Algorithmus zunächst an einem einfachen Beispiel erklären.

Beispiel 4.3.8

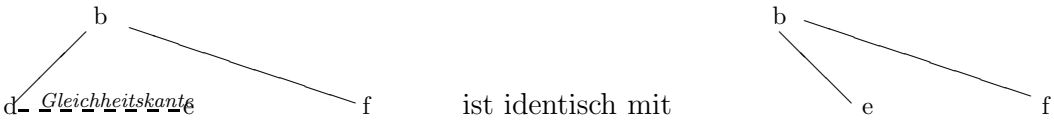
Wir wollen zeigen, daß `a(b(d,f),c) = a(b(e,f),c)` aus der Gleichheit `d=e` folgt, wobei `a,b,c,d,e` und `f` beliebige Terme sind.

Dazu repräsentieren wir Term-Ausdrücke in der üblichen Baumdarstellung und die Gleichheit von Term-Ausdrücken als Gleichheitskanten ("*equality links*") zwischen den Knoten dieser Bäume, wobei identische Teilausdrücke als gemeinsame Teilbäume der verschiedenen Termbäume behandelt werden. Für die obigen Formeln bedeutet dies, daß `c` und `f` gemeinsame Knoten sind, während `d` und `e` durch eine Gleichheitskante verbunden werden.

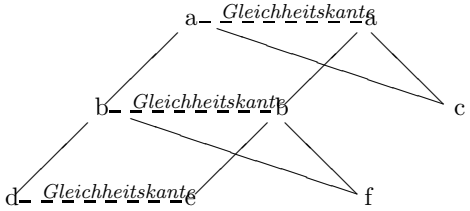
[35] Diese Form des Gleichheitsschließens darf nicht verwechselt werden mit dem aus dem automatischen Beweisen bekannten Problem, logische Schlüsse unter der Verwendung von Gleichheiten zu ziehen. Letzteres ist erheblich komplizierter, da hier Schließen über Gleichheiten und prädikatenlogisches Schließen, das bekanntlich alleine schon unentscheidbar ist, miteinander gekoppelt werden. Für derartige Fragestellungen sind daher wesentlich aufwendigere Verfahren wie Paramodulation [Robinson & Wos, 1969] oder modifizierte logische Verfahren wie E-Resolution [Morris, 1969, Anderson, 1970] oder Gleichheitskonnektionen [Bibel, 1987, Bibel, 1992] nötig.



Der Beweis, daß zwei Ausdrücke gleich sind, ist nun dasselbe wie der Beweis, daß eine Gleichheitskante zwischen den Wurzelknoten ihrer Baumdarstellung konstruiert werden kann. Im Algorithmus geschieht dieser Nachweis dadurch, daß Knoten, die mit einer Gleichheitskante verbunden sind, miteinander identifiziert werden, was im Endeffekt dasselbe ist, als ob die Knoten zusammengelegt worden wären. In unserem Fall bedeutet dies, daß `d` mit `e` identifiziert wird.



Im nächsten Schritt suchen wir nun nach Teilbäumen, in denen `d` oder `e` vorkommen und die in allen anderen Knoten identisch sind. Zwischen den Wurzelknoten dieser Teilbäumen kann nun eine Gleichheitskante konstruiert werden. Wir können also `b(d,f)` mit `b(e,f)` verbinden und im Anschluß daran `a(b(d,f),c)` with `a(b(e,f),c)`, wodurch die Gleichheit der beiden Terme nachgewiesen ist.



Diese Lösung des Problems ist erheblich eleganter und effizienter als ein Beweis, der sich nur auf die Regeln der Reflexivität, Symmetrie, Transitivität und der Substitution stützen kann.

Der Algorithmus konstruiert also Gleichheitskanten zwischen Teilbäumen, bis das Beweisziel bewiesen ist oder alle verbundenen Teilbäume ohne weiteren Fortschritt bearbeitet wurden. Aus technischer Sicht ist diese Methode zum Schließen über Gleichheiten verwandt mit Verfahren zur Bestimmung gemeinsamer Teilausdrücke in optimierenden Compilern.[36] Aus mathematischer Sicht bedeutet die Konstruktion von Gleichheitskanten dasselbe wie die Berechnung der transitiven Hülle (auch *Kongruenzabschluß*, englisch *congruence closure*) der Gleichheitsrelation innerhalb des Termgraphen. Um den Algorithmus präzise beschreiben zu können, führen wir ein paar graphentheoretische Begriffe ein.

Definition 4.3.9

Es sei $G = (V, E)$ ein gerichteter Graph mit markierten Knoten und R eine Relation auf V.

1. *$l(v)$ bezeichnet die Markierung (label) des Knoten v in G*
2. *$\delta(v)$ bezeichnet die Anzahl der von v ausgehenden Kanten.*
3. *Für $1 \leq i \leq \delta(v)$ bezeichnet $v[i]$ den i-ten Nachfolgerknoten von v.*[37]
4. *u ist ein Vorgänger von v, wenn $v = u[i]$ für ein i ist.*
5. *Zwei Knoten u und v sind kongruent unter R, wenn $l(u) = l(v)$, $\delta(u) = \delta(v)$ und $(u[i], v[i]) \in R$ für alle $1 \leq i \leq \delta(u)$ gilt.*
6. *R ist abgeschlossen unter Kongruenzen, wenn für alle Knoten u und v, die unter R kongruent sind, die Relation $(u, v) \in R$ gilt.*
7. *Der Kongruenzabschluß R^* von R ist die eindeutige minimale Erweiterung von R, die abgeschlossen unter Kongruenzen und eine Äquivalenzrelation ist.*

Gegeben sei ein gerichteter Graph $G = (V, E)$, eine unter Kongruenzen abgeschlossene Äquivalenzrelation R und zwei Knoten u, v aus V.

1. Wenn u und v in der gleichen Äquivalenzklasse von R liegen, dann ist R der Kongruenzabschluß von $R \cup \{(u, v)\}$. Lasse R unverändert und beende die Berechnung.
2. Andernfalls sei P_u die Menge aller Vorgänger von Knoten aus der Äquivalenzklasse von u und P_v die Menge aller Vorgänger von Knoten aus der Äquivalenzklasse von v.
3. Modifiziere R durch Verschmelzung der Äquivalenzklassen von u und v.
4. Wiederhole für alle $x \in P_u$ und $y \in P_v$

   Falls die Äquivalenzklassen von x und y verschieden sind, aber x und y kongruent sind, so modifiziere R durch Aufruf von MERGE(R,x,y). Andernfalls lasse R unverändert.

Ausgabe sei die modifizierte Relation R.

Abbildung 4.8: Der MERGE Algorithmus

Der Kongruenzabschluß einer Menge von Gleichheiten s_1=t_1, . . . , s_n=t_n läßt sich mit Hilfe einiger Standardalgorithmen auf Graphen relativ einfach berechnen. Der Kongruenzabschluß der *Identitätsrelation*, bei der Knoten nur in Relation mit sich selbst sind, ist wieder die Identitätsrelation und mit Hilfe des Algorithmus MERGE, den wir in Abbildung 4.8 angegeben haben, können wir schrittweise Gleichungen zu dieser Relation hinzunehmen und jeweils den Kongruenzabschluß berechnen. Aus Effizienzgründen stellen wir dabei eine Äquivalenzrelation durch die entsprechende Menge der Äquivalenzklassen dar. Die Eigenschaften von MERGE beschreibt der folgende Satz.

Satz 4.3.10

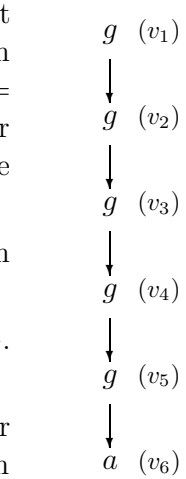
Es sei $G = (V, E)$ ein gerichteter Graph mit markierten Knoten, R eine Äquivalenzrelation auf V, die unter Kongruenzen abgeschlossen ist, und u, v beliebige Knoten aus V.

Dann berechnet MERGE(R,u,v)[38] *den Kongruenzabschluß der Relation $R \cup \{(u, v)\}$.*

Wir wollen die Arbeitsweise des MERGE-Algorithmus an einem Beispiel erläutern.

Beispiel 4.3.11

Wir wollen den Kongruenzabschluß von `g(g(g(a)))=a` und `g(g(g(g(g(a)))))=a` berechnen. Da alle vorkommenden Terme Teilterme von `g(g(g(g(g(a)))))` sind, genügt es, die Baumdarstellung dieses Terms zu konstruieren und alle Kongruenzen in diesem Graphen zu bestimmen. Wir beginnen mit der Äquivalenzrelation $R = \{\{v_1, v_6\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}\}$ (also `g(g(g(g(g(a)))))=a`), die offensichtlich unter Kongruenzen abgeschlossen ist, nehmen als neue Knoten u=v_3 (d.h. `g(g(g(a)))`) sowie v=v_6 (`a`) und rufen MERGE(R,u,v) auf.



Die Vorgänger von u ist v_2 und der von v ist v_5. Da der zu v äquivalente Knoten v_1 keinen Vorgänger besitzt erhalten wir $P_u = \{v_2\}$ und $P_v = \{v_5\}$.

Nun verschmelzen wir die Äquivalenzklassen von u und v und erhalten $\{\{v_1, v_3, v_6\}, \ldots\}$.

Aus den Klassen P_u bzw. P_v wählen wir nun $x = v_2$ und $y = v_5$. Da die Nachfolger von x und y äquivalent sind, sind x und y kongruent, gehören aber zu verschiedenen Äquivalenzklassen. Wir rufen also MERGE(R,v_2,v_5) auf.

[36] Hierzu siehe [Nelson & Oppen, 1980] und die Dissertation von Scott Johnson [Johnson, 1983].

[37] Es ist zulässig, daß mehrere von v ausgehende Kanten auf den gleichen Knoten zeigen, also daß $v[i] = v[j]$ für $i \neq j$ gilt.

[38] Man beachte, daß R Eingabe- *und* Ausgabeparameter von MERGE ist und hierbei verändert wird.

Gegeben seien Hypothesen $s_1{=}t_1, \ldots, s_n{=}t_n$ und eine Konklusion $s{=}t$.

1. Konstruiere den Graphen G, der aus den Baumdarstellungen der Terme $s, s_1, \ldots, s_n, t, t_1, \ldots, t_n$ besteht, wobei identische Teilausdrücke jeweils durch einen Teilbaum dargestellt werden.

   Wähle R als Identitätsrelation auf den Knoten von G

2. Wende schrittweise die Prozedur MERGE(R,$\tau(s_i)$,$\tau(t_i)$) für $1 \leq i \leq n$ an.

3. Wenn $\tau(s)$ in der gleichen Äquivalenzklasse wie $\tau(t)$ liegt, dann sind s und t gleich.

   Andernfalls folgt $s{=}t$ nicht aus $s_1{=}t_1, \ldots, s_n{=}t_n$

Dabei bezeichne $\tau(u)$ den Wurzelknoten der Baumdarstellung des Terms u in G.

Abbildung 4.9: Entscheidungsalgorithmus zum Schließen über Gleichheiten

Dies führt dazu, daß nun in R die Äquivalenzklasse $\{v_2, v_5\}$ gebildet wird und v_1 und v_4 untersucht werden, die sich nun ebenfalls als kongruente Elemente verschiedener Äquivalenzklassen herausstellen.

Beim rekursiven Aufruf von MERGE(R,v_1,v_4) werden nun die Klassen von v_1 und v_4 zu $\{v_1, v_3, v_4, v_6\}$ verschmolzen und die Vorgänger von $\{v_1, v_3, v_6\}$ bzw. von v_4 betrachtet.

Unter diesen sind v_2 und v_3 kongruente Elemente verschiedener Klassen, was zu MERGE(R,v_2,v_3) führt und $\{v_1, v_2, v_3, v_4, v_6\}$ liefert. Die nachfolgende Betrachtung von v_5 und v_3 liefert über MERGE(R,v_5,v_3) schließlich die Äquivalenzklasse $\{v_1, v_2, v_3, v_4, v_5, v_6\}$.

Da nun alle Knoten äquivalent sind stoppt der Algorithmus und liefert die Relation $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ als Kongruenzabschluß der Ausgangsgleichungen. Alle Teilterme von `g(g(g(g(g(a)))))` sind äquivalent.

Quantorenfreies Schließen über Gleichheiten mit uninterpretierten Funktionssymbolen und Variablen kann nun relativ leicht auf das Problem der Bestimmung des Kongruenzabschlusses der Gleichheitsrelation reduziiert werden. Um zu entscheiden, ob $s{=}t$ aus den Hypothesen $s_1{=}t_1, \ldots, s_n{=}t_n$ folgt, genügt es, schrittweise mit dem MERGE-Algorithmus den Kongruenzabschluß der Gleichheiten $s_1{=}t_1, \ldots, s_n{=}t_n$ zu berechnen und dann zu testen, ob s und t in der gleichen Äquivalenzklasse liegen. Dieser Algorithmus, den wir in Abbildung 4.9 beschrieben haben, ist die Grundlage der `equality`-Regel von NuPRL.[39]

4.3.3 Andere Entscheidungsverfahren

Neben den hier vorgestellten Entscheidungsverfahren für Arithmetik und Schließen über Gleichheiten gibt es auf dem Gebiet des automatischen Beweisens noch weitere Entscheidungsprozeduren für gewisse mathematische Teilgebiete. So läßt sich zum Beispiel die Bildung des Kongruenzabschlusses auch für daß Schließen über *Listenstrukturen* und ähnlich gelagerte Problemstellungen einsetzen.[40] Für bestimmte *geometrische Probleme* gibt es effiziente Entscheidungsverfahren [Wu, 1986, Chou & Gao, 1990] genauso wie für die (klassische) *quantorenfreie (Aussagen-)logik mit uninterpretierten Funktionssymbolen* [Davis & Putnam, 1960, Prawitz, 1960].

[39] Man beachte hierbei, daß die Entscheidungsprozedur – aufgrund der einheitlichen Syntax von Termen – in der Lage ist, beliebige Terme zu verarbeiten und nicht etwa nur solche, die aus Funktionsapplikationen aufgebaut sind. Die Rolle der Funktionssymbole wird hierbei von den Operatoren (siehe Definition 3.2.5 auf Seite 104) übernommen. Allerdings befaßt sich der Entscheidungsalgorithmus ausschließlich mit der Frage der Gleichheit von Termen und kann deshalb keine Teilziele behandeln, in denen neben der reinen Gleichheitsüberprüfung auch noch Typbestimmungen nötig sind. Dies schränkt die Einsatzmöglichkeiten der Prozedur `equality` innerhalb von typentheoretischen Beweisen wieder etwas ein. `equality` kann nur auf den Wurzelknoten eines Terms operieren und darf nicht in Teilterme hineingehen, ohne deren Typ zu bestimmen. Ohne eine zusätzliche Unterstützung durch Taktiken sind die Fähigkeiten der `equality`-Regel von NuPRL daher etwas schwächer als die des allgemeinen Algorithmus.

[40] Diese Prozeduren wurden – ebenso wie die unten angesprochene Integration von Entscheidungsprozeduren – in Vorläufern des NuPRL Systems, die noch nicht auf Typentheorie basierten, erfolgreich eingesetzt. Eine Erweiterung auf typentheoretische Konzepte wurde bisher noch nicht durchgeführt, da der Taktik-Ansatz für diese Zwecke vielversprechender erscheint.

Auch für die *klassische Prädikatenlogik* gibt es eine Vielfalt von Beweisverfahren, die üblicherweise eine effiziente *Suchstrategie* [Robinson, 1965, Andrews, 1971, Bibel, 1987] beinhalten. Diese Verfahren bergen allerdings immer die Gefahr in sich, viel Rechenzeit und Speicherplatz zu verbrauchen und dennoch erfolglos zu arbeiten, also keine Entscheidung zu liefern. Aufgrund der Unentscheidbarkeit der Prädikatenlogik können sie daher nur dazu dienen, Beweise zu finden, wenn es welche gibt. *Entscheidungsprozeduren für die Prädikatenlogik kann es jedoch nicht geben.*

Üblicherweise liefert eine Entscheidungsprozedur wie `arith` oder `equality` einen vollständigen Beweis oder sie schlägt fehl, weil die Konklusion innerhalb der eingeschränkten Theorie dieser Prozedur nicht gültig ist. Es gibt jedoch eine Möglichkeit, mehrere Entscheidungsprozeduren miteinander zu kombinieren, so daß sie Informationen über die entdeckten Relationen untereinander austauschen[41] und insgesamt eine größere Theorie bearbeiten können. Durch eine derartige *Kooperation von Entscheidungsprozeduren* kann die Leistungsfähigkeit des Beweissystems erheblich gesteigert werden. Ein Algorithmus, mit dem man verschiedene Entscheidungsprozeduren für quantorenfreie Theorien miteinander kombinieren kann, ist in [Nelson & Oppen, 1979] beschrieben und erfolgreich in λ-PRL (einem Vorläufer von ν-PRL) eingesetzt worden.

4.3.4 Grenzen der Anwendungsmöglichkeiten von Entscheidungsprozeduren

In Beweissystemen, deren zugrundeliegende Logik weniger ausdruckstark ist als die intuitonistische Typentheorie, kann eine Kooperation verschiedener festeingebauter Entscheidungsprozeduren für Arithmetik, Gleichheit, Listenstrukturen, quantorenfreie Logik etc. sehr erfolgreich eingesetzt werden. In Systemen mit einer reichhaltigeren Typstruktur entstehen hierbei jedoch Probleme mit der Typisierung der auftauchenden Teilterme[42] und deshalb können nur wenige dieser Entscheidungsprozeduren sauber in die Theorie eingebettet werden. Aus diesem Grunde stößt das Konzept der Entscheidungsprozeduren als Mittel zur Automatisierung des logischen Schließens sehr schnell an seine Grenzen, denn es wird immer wieder Anwendungsgebiete für logisch-formales Schließen geben, die nicht mehr in einer entscheidbaren Theorie formuliert werden können.

Aber auch aus praktischen Gesichtspunkten macht es wenig Sinn, ein Beweissystem mit einer fest eingebauten Beweisstrategie ständig um Entscheidungsprozeduren zu erweitern, wann immer ein Benutzer eine Problemstellung findet, die von dem bestehenden System nicht mehr gelöst werden kann.[43] Denn hierzu ist immer ein Eingriff in das eigentliche Beweissystem notwendig, was dazu führt, daß dieses im Laufe der Zeit immer weniger verständlich wird und eine ursprünglich vorhandene saubere Systemstruktur irgendwann verloren geht. Zudem müßte jede Beweisprozedur vor ihrer Integration in das System als korrekt und konsistent mit dem Rest des Systems nachgewiesen werden, was zeitaufwendig und üblicherweise kaum durchführbar ist und somit das Vertrauen in die Zuverlässigkeit des Gesamtsystems erheblich belastet.

Entscheidungsprozeduren sind also hilfreich, um Probleme einer kleinen Anzahl wohlverstandener Schlüsseltheorien automatisch zu lösen. Als allgemeine Vorgehensweise, die dazu geeignet ist, das logische Schließen in allen Bereichen der Mathematik und Programmierung zu automatisieren, können sie jedoch nicht angesehen werden. Hierfür ist das Konzept der Taktiken erheblich besser geeignet, da es bei geringen Effizienzverlusten erheblich mehr Flexibilität und Sicherheit bei der Implementierung von Beweisstrategien liefert.

---

[41] Diese Technik nennt man *equality propagation.*

[42] Da die Typzugehörigkeitsrelation unter anderem auch die volle Prädikatenlogik, die Rekursionstheorie und ein Teilmengenkonzept beinhaltet, ist Typisierbarkeit im allgemeinen unentscheidbar. Das wesentliche Problem ist dabei die Bestimmung der Definitionsbereiche von Funktionen. Soll zum Beispiel `f(0)` $\in$ `T` bewiesen werden, so ist es oft nicht möglich, den Typ von `f` zu bestimmen, der für den Nachweis `f(0)` $\in$ `T` erforderlich ist. Ist `f` auf der ganzen Menge $\mathbb{Z}$ definiert, oder nur auf einer Teilmenge davon, welche die Null enthält? Hierfür gibt es keine algorithmische Lösung, da sonst das Halteproblem entscheidbar wäre.

Eine ausführliche Diskussion dieser Problematik findet man in [Harper, 1985].

[43] Diese Vorgehensweise findet man leider im automatischen Theorembeweisen immer noch relativ häufig vor, da diese sich zum Ziel gesetzt haben, alle praktischen Probleme *vollautomatisch*, also ohne Interaktion mit einem Benutzer zu lösen.

4.4 Diskussion

Wir haben uns in diesem Kapitel mit den wesentlichen Techniken für eine Implementierung zuverlässiger interaktiver Beweissysteme und den prinzipiellen Möglichkeiten der Automatisierung der Beweisführung befaßt. Dabei ging es uns weniger um die Fähigkeiten *konkreter Strategien*, mit denen Beweise geführt und und Programme konstruiert werden können, als um die Techniken, mit denen solche Strategien auf eine zuverlässige und effiziente Weise innerhalb eines Beweisentwicklungssystems realisiert werden können.

Grundsätzlich empfiehlt es sich, Beweissysteme für ausdruckstarke formale Theorien wie die intuitionistische Typentheorie, als *interaktiv gesteuerte Systeme* anzulegen, da vollautomatische Systeme aufgrund der vielen Unentscheidbarkeiten ausdruckstarker Formalismen nicht sinnvoll sind. Stattdessen bietet es sich an, eine Kooperation mit einem Benutzer anzustreben, dem hierfür ein verständliches Interface zu der formalen, intern verarbeiteten Theorie und gewisse Möglichkeiten zur Automatisierung der Beweisführung angeboten werden müssen. Für wohlverstandene entscheidbare Teiltheorien des zugrundeliegenden formalen Kalküls sind *Entscheidungsprozeduren* ein sehr wichtiges Hilfsmittel, um einen Benutzer bei der Interaktion von einer Fülle trivialer Schritte zu entlasten. Das Konzept der *Taktiken* bietet wiederum die Möglichkeit einer benutzerdefinierten Erweiterung des Inferenzsystems, die eine sehr flexible und dennoch absolut sichere Einbettung beliebiger Beweisstrategien (bei verhältnismäßig geringen Effizienzverlusten gegenüber einem direkten – ungesicherten – Eingriff in das eigentliche Beweissystem) in das System ermöglicht.

Damit haben wir nun die Grundlagen für die Implementierung eines flexiblen *universellen Basissystems* geschaffen, mit dem Schlußfolgerungen über alle Bereiche der Mathematik und Programmierung formal gezogen und nach Belieben automatisiert werden können. Dieses kann nun durch ausgefeiltere Strategien, die mit den in diesem Kapitel besprochenen Techniken implementiert werden, zu einem leistungsfähigen semiautomatischen Inferenzsystem für ein beliebiges Anwendungsgebiet – wie etwa die Verifikation mathematischer Teiltheorien oder die rechnergestützte Entwicklung von Software – ausgebaut werden, ohne daß hierzu an dem eigentlichen System noch etwas verändert werden muß. Ein Großteil der weiteren Forschungen auf dem Gebiet der Automatisierung von Logik und Programmierung wird sich daher mit der *Anwendung* und der *praktischen Nutzbarmachung* dieses Systems befassen.

Andere Forschungsrichtungen zielen auf Techniken für elegantere Handhabung der Theorie als solche. Hierzu gehören einerseits Arbeiten an einer Verbesserung der in das System integrierten Mechanismen wie Abstraktion, Display und vor allem die Strukturierung der mathematischen Bibliothek, andererseits aber auch die Erforschung von mathematische Schlußfolgerungsmethoden, die von Menschen benutzt werden, bisher aber von der Theorie nicht erfaßt werden konnten. Hierzu zählt vor allem die Fähigkeit, Schlußfolgerungen über das Beweisen als solches zu ziehen, also das sogenannte *Meta-Schließen*. Dies bedeutet zum Beispiel, Aussagen über das Resultat einer Taktik-Anwendung zu treffen [Knoblock, 1987, Constable & Howe, 1990a] oder *Analogien* zwischen bestimmten Problemstellungen für eine Beweiskonstruktion auszunutzen [de la Tour & Kreitz, 1992]. Diese Forschungsrichtung wird besonders interessant, wenn man versucht, das Meta-Schließen durch *Reflektion* wieder in die eigentliche Theorie zu integrieren, also innerhalb der Theorie Schlüsse über Beweise der Theorie zu ziehen [Allen *et.al.*, 1990, Giunchiglia & Smaill, 1989] .[44] Hierdurch könnten Beweise noch effizienter, kürzer und eleganter gestaltet werden und, da sie sich noch mehr an eine menschliche Denkweise anlehnen, dazu genutzt werden, die Kooperation zwischen Mensch und Computersystem weiter verbessern.

4.5 Ergänzende Literatur

Das 1986 erschienene Buch über das Beweisentwicklungssystem NuPRL [Constable *et.al.*, 1986] kann als Referenzbuch für dieses Kapitel angesehen werden, obwohl es in manchen Implementierungsdetails mittlerweile nicht mehr ganz auf dem neuesten Stand ist. Die wichtigsten Änderungen und Ergänzungen sind in drei

[44] Da die Metasprache von NuPRL bereits als mathematische Programmiersprache ML formuliert wurde, ist eine weitere Formalisierung der Metatheorie und ihrer Gesetze prinzipiell möglich.

technischen Manuals [Jackson, 1993a, Jackson, 1993b, Jackson, 1993c] dokumentiert, die derzeit allerdings – genauso wie das NuPRL System selbst – ständig ergänzt werden.

Die Grundideen des taktischen Beweisens werden in [Gordon *et.al.*, 1979] ausführlich diskutiert. Die Ausarbeitung dieses Konzepts für typtentheoretische Beweiser wird in [Constable *et.al.*, 1985] beschrieben. Wertvolle Hinweise findet man auch in den Beschreibungen der in Fußnote 3 auf Seite 182 erwähnten Systeme.

Vertiefende Informationen zu den hier vorgestellten Entscheidungsprozeduren liefert [Constable *et.al.*, 1982], dessen Anhang "*An algorithm for checking PL/CV arithmetic inferences*" von Tat-hung Chan) besonders zu empfehlen ist. Die Arbeiten von Nelson und Oppen [Nelson & Oppen, 1979, Nelson & Oppen, 1980] liefern ebenfalls viele lohnenswerte Informationen.

Kapitel 5

Automatisierte Softwareentwicklung

In den vorherigen Kapiteln haben wir die Grundlagen für die Implementierung eines universellen Basisinferenzsystems geschaffen, mit dem Schlußfolgerungen über alle Bereiche der Mathematik und Programmierung formal gezogen und zum Großteil automatisiert werden können. Alle allgemeinen Techniken sind von ihrer Konzeption her vorgestellt und im Hinblick auf ihre prinzipiellen Möglichkeiten und Grenzen ausführlich diskutiert worden.

Im Anfangskapitel hatten wir als Motivation für die Entwicklung von formalen Kalkülen, Inferenzsystemen und Beweisstrategien besonders die rechnergestützte Softwareentwicklung hervorgehoben. Wir wollen nun auf dieses Ziel zurückkommen und uns mit der praktischen Anwendbarkeit von Inferenzsystemen als Unterstützung bei der Entwicklung garantiert korrekter Software befassen. Wir wollen vor allem Techniken vorstellen, mit denen das allgemeine Beweisentwicklungssystem zu einem leistungsfähigen semi-automatischen System für die Synthese von Programmen aus formalen Spezifikationen ausgebaut werden kann.

Viele dieser Techniken sind in den letzten Jahrzehnten zunächst unabhängig von formalen logischen Kalkülen entstanden[1] und ausgetestet worden. Wir werden sie daher zunächst losgelöst von einer typentheoretischen Formalisierung – wohl aber angepaßt an die verwendete Notation – besprechen und erst im Anschluß daran diskutieren, auf welche Art die bekannten Synthesestrategien in den allgemeinen logischen Formalismus integriert[2] werden können, der die Korrektheit einer Synthese sicherstellen kann.

Programmsynthese geht davon aus, daß die Beschreibung des zu lösenden Problems bereits als eine Spezifikation vorliegt, die in einer präzisen formalen Sprache formuliert ist. Bevor man sich also mit den einzelnen Techniken und Verfahren der Programmsynthese beschäftigen kann, muß man wissen, auf welche Arten sich eine Formalisierung informal gegebener Begriffe und Zusammenhänge realisieren läßt. Die Formalisierung mathematischer Theorien, die wir in Abschnitt 5.1 exemplarisch besprechen werden, bildet die Grundlage einer präzisen Beschreibung von Programmierproblemen und ebenso aller Verfahren, die aus dieser Beschreibung einen korrektes Programm generieren sollen. Während sich Abschnitt 5.1 auf die Formalisierung verschiedener Anwendungsbereiche konzentriert, werden wir in Abschnitt 5.2 die formalen Grundbegriffe vorstellen, die für eine einheitliche Betrachtung verschiedenster Programmsyntheseverfahren erforderlich sind.

Die folgenden drei Abschnitte sind dann den den verschiedenen Methoden zur Synthese von Programmen aus formalen Spezifikationen gewidmet. Abschnitt 5.3 betrachtet die Verfahren, die auf dem bereits

[1] Die Literatur zur Programmsynthese ist extrem umfangreich und es ist nahezu unmöglich, einen Überblick über alle Verfahren zu behalten. Die wichtigsten dieser Verfahren sind in [Green, 1969, Burstall & Darlington, 1977, Manna & Waldinger, 1979, Manna & Waldinger, 1980, Bibel, 1980, Gries, 1981, Hogger, 1981, Bibel & Hörnig, 1984, Dershowitz, 1985, Bates & Constable, 1985, Franova, 1985, Smith, 1985b, Heisel, 1989, Neugebauer *et.al.*, 1989, Smith & Lowry, 1990, Galmiche, 1990, Fribourg, 1990, Bibel, 1991, Franova & Kodratoff, 1991, Lowry, 1991, Smith, 1991a] beschrieben worden. Die Unterschiede zwischen diesen Verfahren liegen vor allem in den konkreten Schwerpunkten, der Effizienz ihrer Implementierung und der Art der notwendigen Benutzerinteraktion begründet.

[2] Die Integration von Synthesestrategien in einen allgemeinen logischen Formalismus ist derzeit ein aktuelles Forschungsgebiet. Die hier präsentierten Wege beschreiben die derzeit in Untersuchung befindlichen Möglichkeiten und sind keineswegs vollends ausgereift. Für eine Vertiefung im Rahmen von Studien- und Diplomarbeiten ist dieses Thema daher besonders geeignet.

$\mathbb{B}$, true, false	Data type of boolean expressions, explicit truth values
$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$	Boolean connectives
$\forall$x$\in$S.p, $\exists$x$\in$S.p	Limited boolean quantifiers (on finite sets and sequences)
if p then a else b	Conditional
Seq(α)	Data type of finite sequences over members of α
null?, $\in$, $\sqsubseteq$	Decision procedures: emptiness, membership, prefix
[], [a], [i..j], [a_1...a_n]	Empty/ singleton sequence, subrange, literal sequence former
a.L, L·a	prepend a, append a to L
[f(x)|x$\in$L$\wedge$p(x)], |L|, L[i]	General sequence former, length of L, i-th element,
domain(L), range(L)	The sets {1..|L|} and {L[i]|i$\in$domain(L)}
nodups(L)	Decision procedure: all the L[i] are distinct (no duplicates)
Set(α)	Data type of *finite* sets over members of α
empty?, $\in$, $\subseteq$	Decision procedures: emptiness, membership, subset
$\emptyset$, {a}, {i..j}, {a_1...a_n}	Empty set, singleton set, integer subset, literal set former
S+a, S-a	element addition, element deletion
{f(x)|x$\in$S$\wedge$p(x)}, |S|	General set former, cardinality
S$\cup$T, S$\cap$T, S\T	Union, intersection, set difference
$\bigcup$FAMILY, $\bigcap$FAMILY	Union, intersection of a family of sets

Abbildung 5.1: Erweitertes Vokabular

bekannten Prinzip "Beweise als Programme" basieren. Abschnitt 5.4 bespricht dann Syntheseverfahren auf der Basis äquivalenzerhaltender Formeltransformationen, wobei wir besonders auch auf das LOPS-Verfahren [Bibel, 1980] eingehen werden. Abschnitt 5.5 schließlich behandelt Verfahren, die auf einer Instantiierung schematischer Algorithmen beruhen, und illustriert diese am Beispiel dreier Algorithmenentwurfsstrategien, die in das KIDS System des Kestrel Instituts in Palo Alto [Smith, 1990, Smith, 1991a] – aus praktischer Hinsicht derzeit das erfolgreichste aller Systeme – integriert sind. Die Möglichkeiten einer nachträglichen Optimierung synthetisch erzeugter Algorithmen besprechen wir dann in Abschnitt 5.6.

Die hier vorgestellten Techniken sind bisher nur wenig systematisiert und vereinheitlicht worden, da sie größtenteils unabhängig voneinander entstanden sind. Der interessierte Leser ist daher meist auf ein intensives Studium einer Vielfalt von Originalarbeiten angewiesen. Mit diesem Kapitel soll versucht werden, die Thematik in einer möglichst einheitlichen Weise darzustellen. Wegen der großen Bandbreite von Denkweisen, die hinter den einzelnen Verfahren stehen, können wir jedoch nicht allen Ansätzen gleichermaßen gerecht werden und sind sogar gezwungen, manche von ihnen gänzlich zu ignorieren.

ACHTUNG: *Dieser Teil des Skriptes ist bis auf weiteres noch in unvollständigem Zustand. Die Abschnitte 5.1, 5.2, und 5.3 beschreiben nur das Notwendigste, was für ein Verständnis der Hauptteile von Bedeutung ist. Eine endgültige Version wird etwas ausführlicher auf die in der Vorlesung angesprochenen Details eingehen.*

5.1 Verifizierte Implementierung mathematischer Theorien

Dieser Abschnitt befaßt sich vor allem mit den konservativen Erweiterungen der Typentheorie, die für eine Formalisierung von Grundbegriffen der wichtigsten Anwendungsbereiche erforderlich sind, welche in der Programmierung immer wieder vorkommen. Neben der Einführung von Vokabular (siehe Abbildung 5.1) geht es hierbei vor allem darum, die grundlegensten Gesetze dieser Begriffe als verifizierte Lemmata der NuPRL-Bibliothek zu formalisieren.

Notwendigerweise muß dabei über eine Systematik beim Aufbau mathematischer Theorien gesprochen werden (vor allem auch darüber, welche Lemmata fundamental sind und welche nicht) und über die technischen Hilfsmittel, wie diese Systematik in einem formalen System umgesetzt werden kann. Hauptsächlich geht es dabei um eine Unterstrukturierung des Wissens in Theorien *und um die Anordnung von Lemmata in der Wissensbank – zum Beispiel durch eine Systematische Vergabe von Lemma-Namen entsprechend der Theorie, dem Hauptoperator und dem Nebenoperator, die in dem Lemma vorkommen. Diese Systematik, die im Detail derzeit noch erforscht wird, liegt der Anordnung der in Anhang B zusammengestellten Definitionen und Lemmata zugrunde.*

5.2 Grundkonzepte der Programmsynthese

In diesem Abschnitt werden vor allem die Grundbegriffe vorgestellt, die für eine einheitliche Betrachtung verschiedenster Programmsyntheseverfahren erforderlich sind. Hierzu werden zunächst die einzelnen Phasen einer systematischen Programmentwicklung angesprochen, um zu verdeutlichen, welche Begriffe unbedingt erforderlich sind.

Eine Präzisierung der Begriffe "formale Spezifikation, Programm, Korrektheit und Erfüllbarkeit von Spezifikationen" als Grundlage jeglicher weiteren Verarbeitung im Syntheseprozeß schließt sich an und wird an einem Beispiel illustriert. Anschließend wird ein kurzer Überblick über Synthesekonzepte und historisch relevante Verfahren gegeben, der es erleichtert, die späteren Kapitel einzuordnen.

Definition 5.2.1 (Spezifikationen, Programme und Korrektheit)

1. *Eine (formale) Spezifikation spec ist ein Tupel (D,R,I,O), wobei*
   - *D ein Datentyp ist (der Eingabebereich oder Domain),*
   - *R ein Datentyp (der Ausgabebereich oder Range),*
   - *I ein Prädikat über D (die Input-Bedingung an zulässige Eingaben) und*
   - *O ein Prädikat über D×R die Output-Bedingung an erlaubte Ausgaben)*
2. *Ein (formales) Programm p besteht aus einer Spezifikation spec=(D,R,I,O) und einer berechenbaren Funktion body: D↛R (dem Programmkörper).*
3. *Ein Programm p=((D,R,I,O), body) heißt korrekt, falls für alle zulässige Eingaben x∈D mit I(x) die Bedingung O(x,body(x)) erfüllt ist.*

Da es sich in beiden Fällen um formale Konzepte handelt, müssen formale Sprachen zur Beschreibung der Komponenten von Spezifikationen und Programmen benutzt werden. Man unterscheidet im allgemeinen die *Spezifikationssprache*, mit der Ein- und Ausgabebereich, sowie Ein- und Ausgabebedingung formuliert werden von der *Programmiersprache*, in welcher der Programmkörper beschrieben wird. Bei der Erstellung automatisierter Programmsynthesesysteme hat es sich jedoch als sinnvoll herausgestellt, in beiden Fällen dieselbe Sprache zu verwenden oder, genauer gesagt, die Programmiersprache als (berechenbaren) Teil der Spezifikationssprache[3] anzusehen.

Da die Tupelschreibweise *((D,R,I,O), body)* im allgemeinen nur sehr schwer zu lesen ist, verwenden wir in konkreten Fällen meist eine Notation, die zusätzlich einige Schlüsselworter zur syntaktischen Auftrennung zwischen den Komponenten und die Abstraktionsvariablen enthält. Da mengenwertige Funktionen bei der Synthese eine große Rolle spielen werden, geben wir auch hierfür eine besondere Notation an.

Definition 5.2.2 (Syntaktisch aufbereitete formale Notationen)

1. *Eine formale Spezifikation spec=*(*D*, *R*, $\lambda x\,.\,I[x]$, $\lambda x, y\,.\,O[x,y]$)[4] *eines Programms f wird auch durch die Notation*

   `FUNCTION` f`(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$

   beschrieben. Wenn $I[x]$=`true` *ist, darf der* `WHERE`*-Teil auch entfallen.*

2. *Die Notation*

   `FUNCTION` f`(`x`:`D`):Set(`R`)` `WHERE` $I[x]$ `RETURNS` $\{\, y \mid O[x,y] \,\}$

   steht als Abkürzung für die Spezifikation

   `FUNCTION` f`(`x`:`D`):Set(`R`)` `WHERE` $I[x]$ `RETURNS S SUCH THAT S =` $\{\, y{:}R \mid O[x,y] \,\}$

[3] Bei einer Integration in die Typentheorie ist dies automatisch gewährleistet. Die Spezifikationssprache geht nur in dem Sinne über die Programmiersprache hinaus, daß hier auch unentscheidbare Prädikate (also Funktionen mit Bildbereich $\mathbb{P}_i$) zugelassen sind, wo die Programmiersprache boole'sche Funktionen erwartet. Dieser Unterschied wird allerdings nur selten ausgenutzt.

[4] Es sei daran erinnert, daß $O[x,y]$ Platzhalter für einen beliebigen Ausdruck ist, in dem x und y frei vorkommen dürfen.

3. *Ein Programm* p=($(D, R, , \lambda x.I[x], \lambda x, y.O[x, y])$, `letrec` $f(x)$=$body[f, x]$) *wird beschrieben durch die Notation*

   `FUNCTION` f`(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x, y] = body[f, x]$

Damit ist das Ziel der Programmsynthese klar. Es geht darum zu einer gegebenen formalen Spezifikation *spec*=`(D.R,I,O)` eine berechenbare Funktion `body` zu bestimmmen, so daß insgesamt das Programm

`FUNCTION` f`(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y) = body(x)`

korrekt ist. Da wir später zeigen wollen, wie man Programmsyntheseverfahren in den formalen Rahmen der Typentheorie integrieren können, müssen wir dieses Ziel als Ziel eines konstruktiven Beweises ausdrücken. Dies ist nicht schwer, da "es ist `body` zu bestimmmen" mit einem Nachweis der Existenz der Funktion `body` gleichgesetzt werden kann. Wir sagen in diesem Fall, daß die Sezifikation *erfüllbar* (oder *synthetisierbar*) ist.

Definition 5.2.3 (Synthetisierbarkeit von Spezifikationen)

Eine formale Spezifikation spec=`(D.R,I,O)` *heißt erfüllbar (synthetisierbar), wenn es eine Funktion* `body:D`$\not\rightarrow$`R` *gibt, die das Programm p*=(*spec*,`body`) *korrekt werden läßt.*

Wir wollen das Vorgehen bei der formalen Spezifikation eines Problems an einem Beispiel illustrieren.

Beispiel 5.2.4 (Costas-Arrays: Problemstellung)

In [Costas, 1984] wurde erstmals eine Klasse von Permutationen beschrieben, die für die Erzeugung leicht wiederzuerkennender Radar- und Sonarsignale besonders gut geeignet sind. Seitdem sind eine Reihe der kombinatorischen Eigenschaften dieser sogenannten *Costas-Arrays* untersucht worden, aber eine allgemeine Konstruktionsmethode konnte noch nicht angegeben werden. Das Problem der Aufzählung von Costas-Arrays muß daher durch Suchalgorithmen gelöst werden. Ein effizientes (d.h. nicht-exponentielles) Verfahren konnte erstmals 4 Jahre nach der Beschreibung der Costas-Arrays in [Silverman *et.al.*, 1988] gegeben werden. Das Problem ist mithin schwierig genug, um sich als Testbeispiel für Syntheseverfahren zu eignen. Andererseits erlaubt die mathematische Beschreibung eine relativ einfache Formalisierung des Problems.

Ein *Costas Array der Ordnung n* ist eine Permutation `p` von $\{1..n\}$ in deren Differenzentafel keine Zeile doppelt vorkommende Elemente besitzt. Dabei ergibt sich die erste Zeile der Differenzentafel aus den Differenzen benachbarter Elemente, die zweite aus den Differenzen der Elemente mit Abstand 2 usw. Ein Costas Array der Ordnung 6 und seine Differenzentafel ist unten als Beispiel gegeben.

2	**4**	**1**	**6**	**5**	**3**	p
-2	**3**	**-5**	**1**	**2**		Zeile 1
1	**-2**	**-4**	**3**			Zeile 2
-4	**-1**	**-2**				Zeile 3
-3	**1**					Zeile 4
-1						Zeile 5
						Zeile 6

Das *Costas Arrays Problem* ist die Frage nach einem effizienten Algorithmus, der bei Eingabe einer natürlichen Zahl n *alle* Costas Arrays der Ordnung n bestimmt.

Um dieses Ziel formal zu spezifizieren, müssen wir zunächst alle Begiffe formalisieren, die in der Problemstellung genannt sind, aber noch nicht zum erweiterten Standardvokabular. Anschließend sollten die wichtigsten Gesetze dieser Konzepte als Lemmata formuliert werden, um für die Verwendung während einer Synthese bereitzustehen (sie müssten ansonsten *während* der Synthese hergeleitet werden). Parallel dazu kann die Problemstellung durch eine formale Spezifikation beschrieben werden.

– Die Konzepte *Permutation* und *Differenzentafel* gehören bisher nicht zum formalisierten Vokabular. Eine Liste L ist Permutation einer Menge S, wenn sie alle Elemente von S enthält, aber keine doppelten Vorkommen. Die Differenzentafel einer Liste L schreiben wir zeilenweise auf und verwenden hierzu die Bezeichnung `dtrow(`L`,`j`)` (difference table of L, row j). Eine solche Zeile besteht aus der Differenz der Elemente von L mit Abstand j.

$$\texttt{perm}(L,S) \quad \equiv \quad \texttt{nodups}(L) \ \wedge \ \texttt{range}(L)=S$$
$$\texttt{dtrow}(L,j) \quad \equiv \quad [L[\texttt{i}]-L[\texttt{i}+j] \mid \texttt{i} \in [1..|L|-j]\,]$$

– Die wichtigsten mathematischen Gesetze der Permutationen ergeben sich nach Auffalten der Definition aus den Gesetzen von `nodups` und `range` und brauchen nicht mehr aufgeführt zu werden.

Für `dtrow` stellen wir – wie immer – vor allem distributive Eigenschaften bezüglich Kombination mit anderen Standardoperationen auf endlichen Folgen auf und erhalten die folgenden Lemmata.

$\forall$L,L':Seq($\mathbb{Z}$).$\forall$i:$\mathbb{Z}$.$\forall$j:$\mathbb{N}$.
1. dtrow([],j) = []
2. j$\leq$|L| $\Rightarrow$ dtrow(i.L,j) = (i-L[j]).dtrow(L,j)
3. j$\neq$0 $\Rightarrow$ dtrow([i],j) = []
4. L$\sqsubseteq$L' $\Rightarrow$ dtrow(L,j) $\sqsubseteq$ dtrow(L',j)
5. j$\geq$|L| $\Rightarrow$ dtrow(L,j) = []
6. j$\leq$|L| $\Rightarrow$ dtrow(L$\cdot$i,j) = dtrow(L,j)$\cdot$(L[|L|+1-j]-i)

– Damit sind alle Begriffe geklärt und wir können die formale Spezifikation aufstellen: das Programm soll bei Eingabe von $n \in \mathbb{Z}$ alle Permutationen p$\in$Seq($\mathbb{Z}$) von {1..n} berechnen, deren Differenzentafeln in keiner Zeile j doppelte Elemente haben. Dies macht nur dann Sinn, wenn $n \geq 1$ ist.

Da Zeilen in der Differenzentafel von p per Definition endliche Folgen sind, können wir die Funktion `nodups` verwenden. Zudem reicht es, Zeilenindizes zwischen 1 und n bzw. aus `domain(p)` zu betrachten, da bei größeren Indizes `dtrow(p,`j`) = []` ist. Das erlaubt die Verwendung des beschränkten All-Quantors und wir erhalten als Ausgabebedingung

$$\texttt{perm(p,\{1..n\})} \ \wedge \ \forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))}$$

Damit haben wir alle Komponenten der Problemspezifikation bestimmt, setzen diese in das Beschreibungsschema für mengenwertige Programmspezifikationen aus Definition 5.2.2 ein und erhalten die folgende formale Spezifikation des Costas Arrays Problems.

```
FUNCTION Costas (n:ℤ):Seq(ℤ)  WHERE n≥1
  RETURNS { p: | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
```

5.3 Programmentwicklung durch Beweisführung

Die grundsätzliche Denkweise des Prinzips "Beweise als Programme" ist eines der fundamentalen Konzepte der intuitionistischen Typentheorie. Wir hatten in Abschnitt 3.4 bereits ausführlich darüber gesprochen.

In diesem Abschnitt soll der Zusammenhang zu den eben eingeführten Konzepten der formalen Spezifikation gezogen werden, um eine Vergleichbarkeit zu anderen Syntheseparadigmen zu erreichen. Konkrete Synthesestrategien auf der Basis des Prinzips "Beweise als Programme" haben üblicherweise einen Induktionsbeweiser als Kernbestandteil. Wir werden dies nur kurz am Beispiel des Oyster/Clam *Systems illustrieren und dann im wesentlichen auf Literatur verweisen, da eine ausführliche Behandlung von Induktionsbeweisern ein Vorlesungsthema für sich ist.*

Zum Schluß sprechen wir über einige Probleme einer "reinen" Synthese von Programmen durch Beweisführung, wobei vor allem das niedrige Niveau der Inferenzschritte und die Schwierigkeiten bei der Konstruktion allgemeinrekursiver Programme zur Sprache kommen.

5.4 Programmentwicklung durch Transformationen

Während das Paradigma "Beweise als Programme" sich im wesentlichen an der Frage nach einer verifizierten Korrektheit von Programmen orientierte, ist eine Steigerung der Effizienz von Programmen der ursprüngliche Hintergrund von transformationsbasierten Verfahren. Dies liegt im wesentlichen daran, daß es für viele Problemstellungen durchaus sehr leicht ist, eine korrekte prototypische Lösung zu erstellen, die sich aufgrund ihrer *abstrakten Beschreibungsform* auch sehr leicht verifizieren und modifizieren läßt. Diese Eleganz erkauft man sich jedoch oft mit einer relativ großen Ineffizienz.

Aus diesem Grunde hat man sich schon sehr früh mit der Frage befaßt, wie man Programme systematisch in effizientere Programme mit gleichem extensionalen Verhalten *transformieren*[5] kann [Darlington, 1975, Burstall & Darlington, 1977, Clark & Sickel, 1977]. Die Fortsetzung dieser Idee auf die Synthese von Programmen aus formalen Spezifikationen [Manna & Waldinger, 1975, Darlington, 1975, Manna & Waldinger, 1979] basiert vor allem auf dem Gedanken, daß eine Spezifikation eigentlich auch als ein sehr ineffizientes – genauer gesagt, nicht ausführbares – Programm betrachtet werden kann, das nun dahingehend verbessert werden muß, daß alle nichtausführbaren Bestandteile durch effizientere ersetzt werden müssen, also durch Ausdrücke, die von einem Computer berechnet werden konnten.

Seit dem Aufkommen des Programmierens in der Sprache der Prädikatenlogik – also der Denkweise der Programmiersprache Prolog – lassen sich Programmtransformationen mehr oder weniger mit Transformationen logischer Formeln identifizieren[6] und sind somit für logische Inferenzsysteme, *Rewrite-Techniken* und Theorembeweiser-basierte Verfahren handhabbar geworden. Viele Verfahren haben aus diesem Grunde auch eine logische Programmierprache als Zielsprache und habe sich zur Aufgabe gesetzt, die Problemspezifikation in eine den Hornklauseln verwandte Formel umzuwandeln, die dann direkt in ein Prolog-Programm übertragen werden kann. Im Gegensatz zur Konstruktion von Programmen durch Beweise läßt sich das Ziel einer Synthese jedoch nicht so deutlich fixieren. Es wird vielmehr durch die internen Vorgaben der Strategien definiert, welche die Anwendung von Transformationen als *Vorwärts*inferenzen steuern.

Aufgrund der Dominanz logik-basierter Verfahren bei der Synthese durch Transformationen und zugunsten einer besseren Vergleichbarkeit mit den anderen Syntheseparadigmen wollen im folgenden eine logische Beschreibungsform für transformationsbasierte Verfahren verwenden. Alle andersartigen Ansätze lassen sich ohne Probleme in diese Form übersetzen.

5.4.1 Synthese durch Transformation logischer Formeln

Aus logischer Sicht besteht eine Synthese durch Transformationen im wesentlichen daraus, daß die Ausgabebedingung einer Spezifikation als ein *neues Prädikat* aufgefaßt wird, dessen Eigenschaften nun mit Hilfe von (bedingten) Äquivalenztransformationen der Ausgabebedingung hergeleitet werden sollen, bis diese Eigenschaften konkret genug sind, um eine rekursive Beschreibung des neuen Prädikates anzugeben, in der nur noch "auswertbare" Teilformeln auftreten. Interpretiert man das neue Prädikat nun als den Kopf eines Logikprogramms so kann die rekursive Beschreibung als der entsprechende Programmkörper angesehen werden, der nun im Sinne der Logik-Programmierung auswertbar ist.

[5] Programmtransformationssysteme mit dem Ziel der Optimierung von Programmen [Burstall & Darlington, 1977, Broy & Pepper, 1981, Partsch & R., 1983, Krieg-Brückner *et.al.*, 1986, Bauer & others, 1988, Krieg-Brückner, 1989] sind mittlerweile ein eigenes Forschungsgebiet geworden, auf dem sehr viel wertvolle Arbeit geleistet wurde. Manche von ihnen verstehen sich durchaus auch als Programmentwicklungssysteme, die den Weg von ineffizienten zu effizienten Spezifikationen von Algorithmen unterstützen. Auch wenn es aus praktischer Sicht durchaus legitim ist, direkt mit einer prototypische Lösung anstelle einer formalen deskriptiven Spezifikation zu beginnen, findet eine Synthese im eigentlichen Sinne hierbei jedoch nicht statt.

[6] Nichtsdestotrotz variieren die individuellen Formalismen sehr stark, da viele Ansätze auch auf der Basis einer funktionalen Sichtweise entwickelt wurden. Die wichtigsten Verfahren sind beschrieben in [Manna & Waldinger, 1975, Hogger, 1978, Manna & Waldinger, 1979, Barstow, 1979, Bibel, 1980, Hogger, 1981, Kant & Barstow, 1981, Dershowitz, 1985, Balzer, 1985, Sato & Tamaki, 1989, Lau & Prestwich, 1990].

Die Vorgehensweise bei der Synthese eines durch D, R, I und O spezifierten Algorithmus ist also die folgende.

1. Zunächst wird ein neues Prädikat `F` über $D \times R$ definiert durch die Formel

   $$\forall \texttt{x}{:}D.\forall \texttt{y}{:}R.\ \ I[x] \quad \Rightarrow \quad \texttt{F(x,y)} \ \Leftrightarrow\ O[x,y]^{7}$$

   Diese Formel hat die äüßere Gestalt einer bedingten Äquivalenz und wird für spätere Inferenzen der Wissensbank des Systems (temporär) hinzugefügt.

2. Unter Verwendung aller aus der Wissensbank bekannten Äquivalenzen – einschließlich der soeben definierten – wird die obige Formel nun so lange transformiert, bis eine Formel der Gestalt

   $$\forall \texttt{x}{:}D.\forall \texttt{y}{:}R.\ \ I[x] \quad \Rightarrow \quad \texttt{F(x,y)} \ \Leftrightarrow\ O^{*}[F,x,y]$$

   erreicht wurde, wobei O^* nur aus erfüllbaren Prädikaten und rekursiven Vorkommen von `F` bestehen darf. Zugunsten einer besseren Übersichtlichkeit darf O^* hierbei auch in mehrere Teilprobleme zerlegt worden sein, die durch separate, neu eingeführte Prädikate beschrieben werden. Am Ende des Transformationsprozesses müssen für deren Definitionsformeln allerdings die gleichen Bedingungen gelten, so daß diese Zerlegung wirklich rein "kosmetischer" Natur ist.

   Welche Regeln angewandt werden müssen, um dieses Ziel zu erreichen, ist zunächst nicht weiter festgelegt.[8] Dies ist eher eine Frage der konkreten Synthesestrategie, welche die anzuwendenden Regeln nach einer internen Heuristik auswählt.

3. Zur Erzeugung eines Programms aus der erreichten Zielformel gibt es nun zwei Möglichkeiten. Man kann die Formel mehr oder weniger direkt *als Logik-Programm interpretieren* und muß in diesem Falle nur die Syntax entsprechend anpassen. Es besteht aber auch die Möglichkeit, durch Anwendung von *Programmformationsregeln*, die logische Konnektive in Programmstrukturen übersetzen, imperative oder funktionale Programme zu erzeugen.

Bei dieser Art der Synthese spielen sogenannte *äquivalenzerhaltende Transformationen* eine zentrale Rolle. Diese basieren im wesentlichen auf bedingten Äquivalenzen und Gleichheiten der Gestalt

$$\forall \texttt{z:T}.\ \ P[z] \ \Rightarrow\ \ Q[z] \ \Leftrightarrow\ Q'[z] \qquad \text{bzw.} \qquad \forall \texttt{z:T}.\ \ P[z] \ \Rightarrow\ \ t[z] \ \texttt{=}\ t'[z],$$

die von nahezu allen Lemmata des Anhang B eingehalten wird. Dabei werden diese Äquivalenzen und Gleichheiten im allgemeinen als gerichtete *Rewrite-Regeln* behandelt, die meist von links nach rechts gelesen werden. Sie ersetzen in einer Formel jedes Vorkommen einer Teilformel $Q[z]$ durch $Q'[z]$ (bzw. eines Terms $t[z]$ durch $t'[z]$, sofern im Kontext dieser Teilformel die Bedingung $P[z]$ nachgewiesen[9] werden kann. Dabei bestehen die Äquivalenzen und Gleichheiten, auf die das Inferenzsystem zugreifen kann, aus den *vorgefertigten Lemmata* der Wissensbank, den *Definitionen neu erzeugter Prädikate*, elementaren *logischen Tautologien* und Abstraktionen, sowie aus dynamisch erzeugten Kombinationen dieser Äquivalenzen.

Da die Leistungsfähigkeit von Syntheseverfahren mittels Äquivalenztransformationen wesentlich von den eingesetzten Strategien abhängt, richtet sich die derzeitige Forschung vor allem auf effiziente Rewrite-Techniken und Heuristiken, sowie auf den Entwurf leistungsfähiger Transformationsregeln. Die Algorithmenschemata, die wir in Abschnitt 5.5 ausführlich behandeln werden, können als eine sehr ausgereifte Form einer Transformationsregel auf hohem Abstraktionsniveau angesehen werden.

[7] Die Verwendung der Notation $O[x,y]$ soll verdeutlichen, daß es sich bei O um eine komplexere Formel mit freien Variablen `x` und `y` handelt, während `F` tatsächlich für einen Prädikatsnamen steht.

[8] Auf den ersten Blick erscheint dies sehr ziellos. Bei der Suche nach Beweisen einer Aussage ist man jedoch im Prinzip in derselben Lage, denn es steht auch nicht fest, welche Regeln zum Beweis nötig sind. Ausschließlich das Ziel – es muß ein Beweis für die gegebene Aussage sein – steht fest. Ein wichtiger Unterschied ist aber, daß die Anzahl der anwendbaren Regeln bei Beweisverfahren erheblich kleiner und ein zielorientiertes Top-Down Suchverfahren möglich ist – solange man nicht in größerem Maße auf Lemmata angewiesen ist. In diesem Fall steht man vor derselben Problematik wie bei der Synthese durch Transformationen.

[9] Auf diese bedingten Äquivalenzen und Gleichheiten werden wir im Rahmen kontextabhängiger Simplifikationen in Abschnitt 5.6.1 noch einmal zurückkommen. Im Prinzip würde es reichen, anstelle von Äquivalenzen auch umgekehrte Implikationen zu verwenden, wodurch sich dann eine *Verfeinerung* des Programms ergibt. Hierbei verliert man jedoch zwangsläufig mögliche Lösungen, was nicht immer gewünscht sein mag.

Wir wollen an einem Beispiel illustrieren, wie man mit Hilfe von äquivalenerhaltenden Transformationen Programme aus formalen Spezifikationen erzeugen kann. Auf die Strategie, mit deren Hilfe wir die anwendbaren Äquivalenzen finden, wollen wir dabei vorerst nicht eingehen, sondern nur demonstrieren, daß das Paradigma "Synthese durch Transformationen" zumindest aus theoretischer Sicht vollständig ist.

Beispiel 5.4.1 (Maximale Segmentsumme)

In Beispiel 3.4.6 auf Seite 147 hatten wir das Problem der Berechnung der maximalen Summe $\texttt{m} = \Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}$ von Teilsegmenten einer nichtleeren Folge L ganzer Zahlen formalisiert und mit Hilfe des Prinzips "Beweise als Programme" gelöst. Dabei hatten wir uns auf eine ausführliche Analyse des Problems gestützt, die wir in Beispiel 1.1.1 auf Seite 2 aufgestellt hatten. Wir wollen nun zeigen, daß sich dieses Problem durch Verwendung von Äquivalenztransformationen auf ähnliche Weise lösen läßt.

Wir beginnen mit der Definition eines neuen Prädikats maxseg, das wir über eine Äquivalenz einführen.

$$\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{m:}\mathbb{Z}.\ \texttt{L}\neq[]\ \Rightarrow\ \texttt{maxseg(L,m)}\ \Leftrightarrow\ \texttt{m}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})$$

Gemäß unserer Analyse in Beispiel 1.1.1 beginnen wir damit, daß wir das Problem generalisieren und simultan zur Berechnung der maximalen Segmentsumme immer auch die maximale Summe a von Anfangssegmenten berechnen. Wir führen hierzu ein Hilfsprädikat max_aux mit drei Variablen L, m und a ein, und stützen maxseg hierauf ab: m ist die maximale Segmentsumme von L, wenn man ein a angeben kann, so daß max_aux(L,m,a) gilt.

$$\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{m:}\mathbb{Z}.\quad \texttt{L}\neq[]\ \Rightarrow\ \texttt{maxseg(L,m)}\ \Leftrightarrow\ \exists \texttt{a:}\mathbb{Z}.\ \texttt{max\_aux(L,m,a)}$$

$$\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{m,a:}\mathbb{Z}.\ \texttt{L}\neq[]\ \Rightarrow\ \texttt{max\_aux(L,m,a)}\ \Leftrightarrow$$
$$\texttt{m}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\ \wedge\ \texttt{a}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})$$

Unter der Voraussetzung, daß max_aux für jede Wahl von L erfüllbar ist, ist dies tatsächlich eine Äquivalenzumformung. In den folgenden Schritten betrachten wir nur noch das Prädikat max_aux. Wir beginnen mit einer Fallanalyse über die Gestalt von L und verwenden hierzu das Lemma

$$\forall \texttt{L:Seq}(\mathbb{Z}).\ \texttt{L}\neq[]\ \Rightarrow\ \texttt{L=[first(L)]}\ \vee\ \texttt{rest(L)}\neq[]$$

Da die Vorbedingung dieses Lemmas erfüllt ist, können wir auf der rechten Seite der Äquivalenz die Disjunktion L=[first(L)] ∨ rest(L)≠[] ergänzen und die disjunktive Normalform bilden.

$$\begin{aligned}&\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{m,a:}\mathbb{Z}.\ \texttt{L}\neq[]\ \Rightarrow\ \texttt{max\_aux(L,m,a)}\ \Leftrightarrow\\ &\quad\texttt{L=[first(L)]}\ \wedge\ \texttt{m}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\\ &\qquad\qquad\wedge\ \texttt{a}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\\ &\vee\ \texttt{rest(L)}\neq[]\ \wedge\ \texttt{m}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\\ &\qquad\qquad\wedge\ \texttt{a}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\end{aligned}$$

Nun verwenden wir (aufbereitete) Lemmata über die Kombination von maximaler Segment- bzw. Anfangssumme und einelementigen Listen und Listen mit nichtleerem Rest.

- Die maximale Segmentsumme einer Liste [i] ist i.

  $$\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{i,m:}\mathbb{Z}.\ \texttt{L=[i]}\ \Rightarrow\ \texttt{m}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\ \Leftrightarrow\ \texttt{m=i}$$

- Die maximale Anfangssumme einer Liste [i] ist i.

  $$\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{i,a:}\mathbb{Z}.\ \texttt{L=[i]}\ \Rightarrow\ \texttt{a}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\Leftrightarrow\ \texttt{a=i}$$

- Die maximale Segmentsumme einer Liste L=i.L', wobei L' nichtleer ist, ist das Maximum der maximalen Segmentsumme von L' und der maximalen Anfangssumme von L.

  $$\begin{aligned}&\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{i,a:}\mathbb{Z}.\ \texttt{rest(L)}\neq[]\ \Rightarrow\ \texttt{m}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\\ &\quad\Leftrightarrow\ \exists \texttt{a:}\mathbb{Z}.\ \texttt{a}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\\ &\qquad\wedge\ \exists \texttt{m':}\mathbb{Z}.\ \texttt{m'}=\texttt{MAX}(\bigcup\{\,\{\Sigma_{\texttt{i}=\texttt{p}}^{\texttt{q}}\texttt{rest(L)[i]}\mid \texttt{p}\in\{\texttt{1..q}\}\}\mid \texttt{q}\in\{\texttt{1..}|\texttt{rest(L)}|\}\,\})\\ &\qquad\wedge\ \texttt{m=max(m',a)}\end{aligned}$$

- Die maximale Anfangssumme einer Liste L=i.L', wobei L' nichtleer ist, ist das Maximum von i und der Summe von i und der maximalen Anfangssumme von L'

  $$\begin{aligned}&\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{i,a:}\mathbb{Z}.\ \texttt{rest(L)}\neq[]\ \Rightarrow\ \texttt{a}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{L[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{L}|\}\,\})\\ &\quad\Leftrightarrow\ \exists \texttt{a':}\mathbb{Z}.\ \texttt{a'}=\texttt{MAX}(\{\Sigma_{\texttt{i}=\texttt{1}}^{\texttt{q}}\texttt{rest(L)[i]}\mid \texttt{q}\in\{\texttt{1..}|\texttt{rest(L)}|\}\,\})\ \wedge\ \texttt{a=max(first(L), a'+first(L))}\end{aligned}$$

Wenden wir diese Lemmata nacheinander als Transformationen auf die obige Formel an (wobei wir im dritten Schritt kein neues a mehr einführen müssen), so ergibt sich

$$\begin{array}{l}\forall L{:}Seq(\mathbb{Z}).\forall m,a{:}\mathbb{Z}.\ L\neq[]\ \Rightarrow\ max\_aux(L,m,a)\ \Leftrightarrow\\ \quad L=[first(L)]\ \wedge\ m=first(L)\ \wedge\ a=first(L)\\ \vee\ rest(L)\neq[]\ \wedge\ \exists m'{:}\mathbb{Z}.\ m'=MAX(\bigcup\{\{\Sigma_{i=p}^{q}rest(L)[i]\mid p\in\{1..q\}\}\mid q\in\{1..|rest(L)|\}\})\\ \qquad\qquad\wedge\ \exists a'{:}\mathbb{Z}.\ a'=MAX(\{\Sigma_{i=1}^{q}rest(L)[i]\mid q\in\{1..|rest(L)|\}\})\\ \qquad\qquad\wedge\ a=max(first(L),\ a'+first(L))\\ \qquad\qquad\wedge\ m=max(m',a)\end{array}$$

Den ersten Fall können wir aufgrund der Gleichung $m=first(L)$ zu $L=[m]\ \wedge\ m=first(L)\ \wedge\ a=m$ vereinfachen. Aufgrund des Lemmas $L=[m]\ \Rightarrow\ m=first(L)$ kann das mittlere Konjunkt dann entfallen. Im zweiten Fall erlaubt uns der Kontext $rest(L)\neq[]$, die Definition von max_aux für rest(L) wieder zurückzufalten. Wir erhalten insgesamt

$$\forall L{:}Seq(\mathbb{Z}).\forall m{:}\mathbb{Z}.\quad L\neq[]\ \Rightarrow\ maxseg(L,m)\ \Leftrightarrow\ \exists a{:}\mathbb{Z}.\ max\_aux(L,m,a)$$

$$\begin{array}{l}\forall L{:}Seq(\mathbb{Z}).\forall m,a{:}\mathbb{Z}.\ L\neq[]\ \Rightarrow\ max\_aux(L,m,a)\ \Leftrightarrow\\ \quad L=[m]\ \wedge\ a=m\\ \vee\ rest(L)\neq[]\ \wedge\ \exists m',a'{:}\mathbb{Z}.\ max\_aux(rest(L),m',a')\\ \qquad\qquad\wedge\ a=max(first(L),\ a'+first(L))\ \wedge\ m=max(m',a)\end{array}$$

Dieses Formelpaar kann nun ohne großen Aufwand schrittweise in ein logisches Programm überführt werden. Hierzu lassen wir alle Quantoren entfallen, da in logischen Programmen alle Variablen der linken Seite all-quantifiziert und alle zusätzlichen Variablen der rechten Seite existentiell quantifiziert sind. Die Vorbedingungen können im Programmcode ebenfalls entfallen. Dies führt zu der vereinfachten Form

$$maxseg(L,m)\ \Leftrightarrow\ max\_aux(L,m,a)$$

$$\begin{array}{l}max\_aux(L,m,a)\ \Leftrightarrow\\ \quad L=[m]\ \wedge\ a=m\\ \vee\ rest(L)\neq[]\ \wedge\ max\_aux(rest(L),m',a')\ \wedge\ a=max(first(L),\ a'+first(L))\ \wedge\ m=max(m',a)\end{array}$$

Nun zerlegen wir die Disjunktion in zwei separate Bedingungsklauseln, was wiederum der Semantik von logischen Programmen entspricht. In der letzten Klausel kann dann die Bedingung $rest(L)\neq[]$ entfallen, da sie der Nichtanwendbarkeit der zweiten Klauselfolgt. Da Prolog-Programme – ähnlich zum ML-Abstraktionsmechanismus– Listen automatisch strukturell zerlegen können schreiben wir i.L' anstelle von L, um uns die Notationen first und rest zu ersparen. Dies liefert

$$maxseg(L,m)\ \Leftrightarrow\ max\_aux(L,m,a)$$

$$\begin{array}{lcl}max\_aux(L,m,a) & \Leftarrow & L=[m]\ \wedge\ a=m\\ max\_aux(i.L',m,a) & \Leftarrow & max\_aux(L',m',a')\ \wedge\ a=max(i,\ a'+i)\ \wedge\ m=max(m',a)\end{array}$$

Die Semantik logischer Programme erlaubt nun, in der zweiten Klausel die Gleichungen in den Programmkopf zu verlagern. Da Funktionen in logischen Programmen nicht erlaubt sind, ersetzen wir die Funktion max durch das entsprechende prädikative Programm MAX, dessen dritte Komponente die Ausgabe der Berechnung des Maximums zweier Zahlen ist. Nun ersetzen wir noch $\wedge$ durch ein Komma und $\Leftrightarrow$ und $\Leftarrow$ durch :- und wir haben ein Prolog-Programm zur Berechnung der maximalen Segmentsumme einer nichtleeren Folge ganzer Zahlen erzeugt.

```
maxseg(L,m)      :- max_aux(L,l,m).
max_aux([m],m,m).
max_aux(a.L',l,m):- max_aux(L,m',l'),max(a,l'+a,l),max(l,m',m).
```

In diesem Beispiel ist noch sehr vieles von Hand geschehen und auch die Auswahl der Lemmata war sehr speziell. Auch wäre ohne eine vorherige Analyse das Problem nicht zu lösen gewesen, da man sonst sehr wahllos nach anwendbaren Lemmata gesucht hätte. Diese Möglichkeit steht einer maschinell gesteuerten Synthese, in der alle Schritte durch eine Strategie bestimmt werden müssen, ebensowenig zur Verfügung wie die auf das Problem zugeschnittenen Lemmata. Eine Strategie muß daher nach wesentlich allgemeineren Kriterien vorgehen und eine Methodik verfolgen, die in manchen Fällen zwar etwas umständlich sein mag, aber (fast) immer zum Erfolg führt.

Mit einer konkreten Synthesestrategie auf der Basis äquivalenzerhaltender Transformationen werden wir uns im nächsten Abschnitt befassen. Zuvor wollen wir im Rest dieses Abschnitts jedoch noch den abschließenden Schritt einer Synthese, also der Erzeugung konkreter Programme diskutieren. In Beispiel 5.4.1 hatten wir zur Illustration ein logisches Programm generiert. Wir wollen die hierbei eingesetzten *Programmformationsregeln* kurz zusammenstellen.

Strategie 5.4.2 (Formation logischer Programme)
Eine Menge bedingter Äquivalenzen der Gestalt

$$\forall \mathtt{x}{:}D\,.\,\forall \mathtt{y}{:}R\,.\quad I[x]\ \Rightarrow\ \mathtt{F(x,y)}\ \Leftrightarrow\ O^*[\mathtt{F},\mathtt{F}_i,x,y]$$
$$\forall \mathtt{x}_1{:}D_1\,.\,\forall \mathtt{y}_1{:}R_1\,.\ I_1[x_1]\ \Rightarrow\ \mathtt{F}_1\mathtt{(x}_1\mathtt{,y}_1\mathtt{)}\ \Leftrightarrow\ O_1^*[\mathtt{F},\mathtt{F}_i,x_1,y_1]$$
$$\vdots$$
$$\forall \mathtt{x}_n{:}D_n\,.\,\forall \mathtt{y}_n{:}R_n\,.\ I_n[x_n]\ \Rightarrow\ \mathtt{F}_n\mathtt{(x}_n\mathtt{,y}_n\mathtt{)}\ \Leftrightarrow\ \mathtt{O}_n^*[\mathtt{F},\mathtt{F}_i,x_n,y_n]$$

wobei auf der linken Seite nur atomare Formeln stehen, wird durch sukzessive Anwendung der folgenden Regeln in ein logisches Programm umgewandelt.

- *Die äußeren Allquantoren entfallen.*
  (Alle Variablen im Programmkopf sind implizit allquantifiziert.)
- *Existenzquantoren auf der rechten Seite der Äquivalenz entfallen.*
  (Alle neuen Variablen im Programmkörper sind implizit existenzquantifiziert.)
- *Alle Vorbedingungen entfallen*
  (Vorbedingungen werden vom Programm nicht überprüft)
- *Äquivalenzen mit einer unerfüllbaren rechten Seite entfallen*
  (Der Programmaufruf würde keine Lösung finden können)
- *Disjunktionen auf der rechten Seite werden zerlegt in zwei sequentielle Äquivalenzen mit der gleichen*[10] *linken Seite. Hierzu wird die rechte Seite zunächst auf disjunktive Normalform gebracht.*
  (Zwei Klauseln mit demselben Programmkopf gelten als Alternativen zur Lösung des Problems und werden der Reihe nach abgearbeitet.)
- *Funktionesaufrufe der Gestalt* `y=g(x)` *werden durch die entsprechenden (bereits bekannten) prädikativen Programmaufrufe der Gestalt* `G(x,y)` *ersetzt.*
  (Logische Programme kennen nur prädikative Programme)
- *Zerlegungen von Standarddatentypen wie* `first(L)/rest(L)` *werden durch Strukturmuster wie* `x.L'` *anstelle von* `L` *im Programmkopf ersetzt.*
  (Eine strukturelle Zerlegung durch Pattern-Matching geschieht automatisch)
- *Gleichheiten zwischen Variablen der rechten Seite können durch entsprechende Substitutionen der Variablen auf der linken Seite der Äquivalenz ersetzt werden.*
  (Durch Pattern-Matching wird der Ausgabevariablen der Wert der entsprechenden Eingabe zugewiesen.)

Für logische Programme sind die Formationsregeln verhältnismäßig einfach, da im wesentlichen nur noch syntaktische Anpassungen an die Konventionen logischer Programmiersprachen vorgenommen werden müssen. Für die Erzeugung andersartiger Programme sind die Programmformationsregeln ein wenig komplizierter, da nun logische Konnektive in *Programmstrukturen* übersetzt werden können. Zugunsten einer Vergleichbarkeit der Paradigmen geben wir diese Regeln nun auch für die funktionale Programmierung an.

[10] Der Sonderfall, daß die rechte Seite nur aus Disjunktionen einiger F_i besteht, kann optimiert werden. Da nun mehrere Klauseln nur aus dem Aufruf eines F_i bestehen, können all diese F_i umbenannt werden in dasselbe Prädikat. Die dadurch entstehenden redundanten Klauseln können entfernt werden.

Dabei gehen wir wie bei der Formation logischer Programme davon aus, daß alle vorkommenden atomaren Formeln bereits als erfüllbar bekannt oder rekursive Varianten einer der linken Seiten sind. Die Formationsregeln müssen also nur angeben, wie man aus bekannten Teillösungen eine Gesamtlösung zusammensetzt, welche die zusammengesetzte Spezifikation erfüllt. Hierzu geben wir für jedes logische Konnektiv eine eigene Regel an, wobei – ebenfalls wie bei logischen Programmen – Implikationen und Negationen nicht individuell verarbeitet werden können. Wir erwarten also am Ende der Transformationen eine gewisse Normalform.

Die einfachste Formationsregel gibt es für die *Konjunktion*. Wenn O einzeln erfüllbar ist, und die Ausgabebedingung $O[x,y] \wedge P(x,y)$ erfüllt werden soll, so kann man P als eine Art Filter für die Lösungen von O betrachten, die man zuvor berechnet hat. Dies bedeutet, daß für alle Eingaben, deren Lösung auch die Bedingung P erfüllt, einfach diese Lösung übernommen wird. Eine Konjunktion in der Ausgabebedingung entspricht also einer *Restriktion* der Menge der legalen Eingaben.

Lemma 5.4.3 (Formationsregel für die Konjunktion)

Es sei spec$=(D,R,I,O)$ *eine beliebige Spezifikation,* $P\colon D\times R\to\mathbb{B}$ *ein Prädikat. Die Funktion* $body\colon D\not\to R$ *erfülle die Spezifikation* `FUNCTION F(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$. *Dann wird die Spezifikation*

`FUNCTION F(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y] \wedge P(x,y)$

von der Funktion body *erfüllt, falls für alle* $x\in D$ *mit* $I[x]$ *die Eigenschaft* $P(x,body(x))$ *gilt.*

Diese Regel hat natürlich nur einen sehr engen Anwendungsbereich, da sie nicht darauf eingeht, ob für die kombinierte Ausgabebedingung eine andere Lösungsfunktion besser geeignet wäre. Eine angemessenere Regel kann man eigentlich nur für mengenwertige Programme angeben, da nur bei diesen Programmen ausgedrückt werden kann, daß die *Menge* der Lösungen eingeschränkt wird zu $\{\, y \mid y\in body(x) \wedge P(x,y)\}$.

Eine *Disjunktion* hatte bei logischen Programmen zu zwei alternativen Programmdefinitionsteilen geführt, die gemäß der Sematik logischer Programme der Reihe nach verarbeitet werden: die zweite Alternative wird nur dann aufgerufen, wenn die erste nicht zum Erfolg führt. Dies bedeutet, daß die zweite Alternatie genau dann aufgerufen wird, wenn die (nicht explizit vorhandene) Eingabebedingung der ersten nicht mehr erfüllt ist. In einer funktionalen Denkweise entspricht dies einer *Fallunterscheidung*: die Lösungsfunktion des ersten Disjunkts wird aufgerufen, wenn ihre Vorbedingung erfüllt ist und ansonsten die Lösungsfunktion der zweiten.

Lemma 5.4.4 (Formationsregel für die Disjunktion)

Es seien spec$=(D,R,I,O)$ *und* spec'$=(D,R,I',O')$ *beliebige Spezifikationen, die von den Funktionen* $body\colon D\not\to R$ *bzw.* $body'\colon D\not\to R$ *erfüllt werden. Dann wird die Spezifikation*

`FUNCTION F(`x`:`D`):`R `WHERE` $I[x]\vee I'[x]$ `RETURNS` y `SUCH THAT` $O[x,y] \vee O'[x,y]$

erfüllt von der Funktion λ`x. if` I[x] `then` $body$`(x) else` $body'$`(x).`

Der *Existenzquantor* führte bei logischen Programmen dazu, daß für die neue Variable ein Wert bestimmt werden mußte, der dann zur Bestimmung der Lösung weiter verarbeitet wurde. Hierbei gibt es zwei Lesarten.

- Im ersten Fall läßt sich die Ausgabebedingung zerlegen in ein e Teilbedingung O, bei der die neue Variable eine Ausgabe beschreibt, und eine Teilbedingung O', bei der sie als Eingabe verwendet wird. Der Existenzquantor dient dann der Beschreibung einer *kaskadischen Berechnung*.

- Im anderen Fall ist die neue Variable nur eine zusätzliche Ausgabevariable eines anderen Prädikates O, das nicht weiter untergliedert werden kann. Bei diesem Prädikat handelt es sich dann um eine G*eneralisierung* des Problems und der Existenzquantor deutet an, daß nur eine der Ausgabevariablen von Interesse ist.

Lemma 5.4.5 (Formationsregeln für den Existenzquantor)

1. *Es seien* spec=(*D*,*R*,*I*,*O*) *und* spec'=(*D*×*R*,*R'*,*I'*,*O'*) *beliebige Spezifikationen, die von den Funktionen* $body: D \not\to R$ *bzw.* $body': D \times R \not\to R'$ *erfüllt werden. Dann wird die Spezifikation*

   FUNCTION F(x:D):R' WHERE $I[x]$ RETURNS z SUCH THAT $\exists y{:}R.\ O[x,y] \wedge O'(x,y,z)$

   erfüllt λx. *body'*(x,*body*(x)), *falls für alle* $x \in D$ *mit* $I[x]$ *die Eigenschaft* $I'(x, body(x))$ *gilt.*

2. *Es sei* spec=(*D*,*R'*×*R*,*I*,*O*) *eine beliebige Spezifikationen, die von der Funktionen* $body: D \not\to R' \times R$ *erfüllt werde. Dann wird die Spezifikation*

   FUNCTION F(x:D):R WHERE $I[x]$ RETURNS z SUCH THAT $\exists y{:}R'.\ O[x,y,z]$

   von der Funktion λx. *body*(x).1 *erfüllt*

Eine weitere Formationsregel ergibt sich aus der Tatsache, daß Transformationen meist auf eine *rekursive Beschreibung* des Programms abzielen. Während diese bei Logikprogrammen ohne weitere Änderung übernommen werden kann (das Prädikat 'ist' das Programm), muß in einer funktionalen Denkweise ein rekursives Programm erzeugt werden. Hierbei gilt die rekursive Beschreibung der Ausgabebedingung als Voraussetzung für die Formationsregel, taucht aber in der Spezifikation nicht so direkt auf wie in den bisherigen Regeln. Für die totale Korrektheit des generierten Programms muß zudem sichergestellt sein, daß die Rekursion auch *terminiert*, daß also die Eingabe nicht beliebig reduziert werden kann. Die folgende Formationsregel beschreibt die einfachste Version der Erzeugung rekursiver Programme: die Ausgabebedingung läßt sich zerlegen in eine rekursive Variante und eine Reihe von Bedingungen, in denen die rekursiv berechnete Teillösung zu einer Gesamtlösung zusammengesetzt wird.

Satz 5.4.6 (Formationsregel für rekursiv zerlegbare Prädikate)

Es sei spec=(*D*,*R'*×*R*,*I*,*O*) *eine beliebige Spezifikationen,* $f_d: D \not\to D$ *wohlfundierte 'Reduktionsfunktion',*[11] $O_C: D \times D \times R \times R \to \mathbb{B}$ *und* $body: D \times D \times R \not\to R$. *Gilt für alle* $x, x_r \in D$, $y, y_r \in R$ *mit* $I[x]$

1. $I[x] \Rightarrow I[f_d(x)/x]$
2. $I[x] \Rightarrow O[x,y] \Leftrightarrow \exists y_r{:}R.\ O[f_d(x),y_r] \wedge O_C(x,f_d(x),y_r,y)$
3. $I[x] \Rightarrow O_C(x,x_r,y_r,\ body(x,x_r,y_r))$

so wird die Spezifikation

FUNCTION F(x:D):R WHERE $I[x]$ RETURNS z SUCH THAT $O[x,y]$

erfüllt von der Funktion letrec f(x) = *body*(x, f_d(x), f(f_d(x)))

Diese Regel entspricht der Formationsregel für Divide & Conquer Algorithmen, die wir in Abschnitt 5.5.4 ausführlicher besprechen werden. Natürlich ist sie eine drastische Vereinfachung, da im Allgemeinfall das Ergebnis der Transformationen auch mehrere rekursive Varianten der Ausgabebedingung auf der rechten Seite enthalten kann. Die entsprechende allgemeine Regel ist nur wenig anders, bedarf aber einiger zusätzlicher[12] Formalia. Anhand einer Analyse der inneren Struktur rekursiver Funktionen macht man sich relativ schnell klar, daß diese Regel vollständig ist, daß sich also jede berechenbare Funktion schreiben läßt als

letrec f(x) = let $x_1,..,x_n$ = f_d(x) in *body*(x, $x_1,..,x_n$, f(x_1),...,f(x_n))

und somit einer rekursiven Zerlegung der Ausgabebedingung in der obengenannten Art entspricht.

Theorem 5.4.6 beschreibt die wesentliche häufigere $\wedge$-*Reduktion* eines Problems, bei der die Spezifikation in eine Konjunktion von Bedingungen zerlegt wird und zur Berechnung einer einzelnen Lösung *alle* Teillösungen nötig sind. Möglich ist aber auch eine sogenannte $\vee$-*Reduktion*, bei der die verschiedenen Lösungen der Teilprobleme jeweils direkt zu einer Lösung des Gesamtproblems beitragen. In einer rein funktionalen Denkweise

[11] Eine Reduktionsfunktion $f_d: D \not\to D$ heißt wohlfundiert, wenn es keine unendlichen absteigenden Ketten der Form x, f_d(x), $f_d(f_d(x))$, $f_d(f_d(f_d(x)))$, ... gibt.

[12] Die Reduktionsfunktion f_d müsste so beschrieben werden, daß sie die Eingabe x in eine Menge reduzierter Eingaben abbildet. Anstelle von *O* müsste auf der rechten Seite eine Menge von Bedingungen stehen, deren Ein- und Ausgaben entsprechend von O_C weiterverwendet werden. Eine entsprechende Notation zur Kennzeichnung dieser Mengen müßte erst eingeführt werden.

kann dies nur durch die Verwendung mengenwertiger Funktionen (im Sinne von Definition 5.2.2.2) exakt wiedergegeben werden, da andernfalls die entstehende Fallunterscheidung wieder einer $\wedge$-Reduktion entspricht. Überhaupt muß festgestellt werden, daß Synthese durch Transformation mehr auf die Beschreibung *aller Lösungen* einer Ausgabebedingung abzielt und somit logische Programme oder mengenwertige Funktionen als Zielsprache adäquater erscheinen als Funktionen, die nur eine Lösung berechnen. Der hierzu notwendige Formalismus ist bisher allerdings noch nicht aufgestellt worden und bleibt ein Ziel für zukünftige Forschungen.

Wir wollen die allgemeine Diskussion von Synthese durch Transformationen mit dem Beispiel einer Programmformation unter Verwendung der obigen Regeln abschließen.

Beispiel 5.4.7 (Maximale Segmentsumme: Programmformation)

In Beispiel 5.4.1 hatten wir das Problem der maximalen Segmentsumme durch Transformationen in folgende rekursive Beschreibung umgewandelt.

$$\forall L{:}Seq(\mathbb{Z}).\forall m{:}\mathbb{Z}.\quad L\neq[]\ \Rightarrow\ maxseg(L,m)\ \Leftrightarrow\ \exists a{:}\mathbb{Z}.\ max\_aux(L,m,a)$$

$$\begin{aligned}\forall L{:}Seq(\mathbb{Z}).\forall m,a{:}\mathbb{Z}.\ L\neq[]\ &\Rightarrow\ max\_aux(L,m,a)\ \Leftrightarrow\\ L=[first(L)]\ &\wedge\ m=first(L)\ \wedge\ a=first(L)\\ \vee\ rest(L)\neq[]\ &\wedge\ \exists m',a'{:}\mathbb{Z}.\ max\_aux(rest(L),m',a')\\ &\wedge\ a=max(first(L),\ a'+first(L))\ \wedge\ m=max(m',a)\end{aligned}$$

`max_aux` ist eine Generalisierung von `maxseg` im Sinne von Lemma 5.4.5.2 und besitzt selbst eine rekursive Beschreibung. Um Theorem 5.4.6 anzuwenden, müssen wir nur noch die dort vorkommenden Parameter mit den aktuellen Komponenten in Beziehung setzen.

`x` entspricht `L`, `y` entspricht dem Paar `m,a`, f_d`(x)` entspricht L_r`=rest(L)`, y_r entspricht `m',a'` und O_C`(L,`L_r`,m',a',m,a')` ist die Bedingung

$$\begin{aligned}&L=[first(L)]\ \wedge\ m=first(L)\ \wedge\ a=first(L)\\ \vee\ &L_r\neq[]\ \wedge\ a=max(first(L),a'+first(L))\ \wedge\ m=max(m',a)\end{aligned}$$

f_d ist offensichtlich wohlfundiert und eine Lösung für O_C ist die Funktion

$$\begin{aligned}&\lambda L,L_r,m',a'.\ \ \textbf{if}\ L_r=[]\ \textbf{then}\ (first(L),first(L))\\ &\qquad\qquad\qquad\textbf{else let}\ a=max(first(L),\ a'+first(L))\ \textbf{in}\ \ (max(m',a),a)\end{aligned}$$

Insgesamt ergibt sich somit folgende funktionale Lösung des Problems der maximalen Segmentsumme

$$\begin{aligned}&FUNCTION\ Maxseg(L{:}Seq(\mathbb{Z})){:}\mathbb{Z}\ \ WHERE\ L\neq[]\ RETURNS\ m\\ &\quad SUCH\ THAT\ m=MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\mid p\in\{1..q\}\}\mid q\in\{1..|L|\}\})\\ &=\ \ snd(Max\_aux(L))\end{aligned}$$

$$\begin{aligned}&FUNCTION\ Max\_aux(L{:}Seq(\mathbb{Z})){:}\mathbb{Z}\times\mathbb{Z}\ \ WHERE\ L\neq[]\ \ RETURNS\ m,l\\ &\quad SUCH\ THAT\ m=MAX(\bigcup\{\{\Sigma_{i=p}^{q}L[i]\mid p\in\{1..q\}\}\mid q\in\{1..|L|\}\})\ \wedge\ a=MAX(\{\Sigma_{i=1}^{q}L[i]\mid q\in\{1..|L|\}\})\\ &=\ \textbf{if}\ rest(L)=[]\ \textbf{then}\ (first(L),first(L))\\ &\qquad\qquad\textbf{else let}\ (m',l')=Max\_aux(rest(L))\ \textbf{in}\\ &\qquad\qquad\qquad\textbf{let}\ a=max(first(L),\ a'+first(L))\ \textbf{in}\\ &\qquad\qquad\qquad\quad(max(m',a),a)\end{aligned}$$

5.4.2 Die LOPS Strategie

LOPS – eine Abkürzung für logische Programmsynthese – ist ein transformationsbasiertes Syntheseverfahren, welches auf einer Reihe von Strategien basiert, die erstmalig in [Bibel, 1978, Bibel *et.al.*, 1978, Bibel, 1980, Bibel & Hörnig, 1984] beschrieben wurden. Den Mittelpunkt dieses Verfahrens bilden zwei Strategien namens GUESS-DOMAIN und GET-REC. GUESS-DOMAIN versucht einen Teil der Spezifikation zu bestimmen, mit dessen Hilfe man das Problem so in kleinere Teilprobleme zerlegen kann, daß sich die Gesamtlösung aus Lösungen der Teilprobleme ergibt. Auf der Basis dieser Zerlegung versucht GET-REC dann eine rekursive Lösung des Problems zu erzeugen. Diese beiden Strategien werden unterstützt durch eine Reihe weiterer Strategien zur Vor- und Nachverarbeitung sowie zur Vereinfachung von Teilproblemen. Aus dem Resultat der Transformationen wird in einem abschließenden Schritt ein Programm erzeugt, wobei man im wesentlichen der im vorigen Abschnitt beschriebenen Methode folgt.

Im Gegensatz zu den meisten Ansätzen ist die LOPS Strategie ein typischer Vertreter eines Verfahrens aus der *künstlichen Intelligenz*. Es basiert im wesentlichen auf *syntaktischen Suchverfahren*, die – zugunsten einer Begrenzung des Suchraumes – durch semantische Informationen und Heuristiken gesteuert werden. Dies wird vor allem durch die Vorstellung motiviert, daß ein Mensch bei der Programmierung ähnliche Fähigkeiten einsetzt wie bei der Lösung anderer Probleme und daß somit *allgemeine* Techniken, die auf Raten von Lösungen und Reduktionen auf geklöste Probleme basieren, Anwendung finden. Wir wollen im folgenden die LOPS-Strategie anhand eines einfachen Leitbeispiels entwickeln und zum Schluß in geschlossener Weise zusammenstellen.

Beispiel 5.4.8 (Maximum-Problem)

Das Maximum-Problem besteht darin, einen Algorithmus zu generieren, der aus einer gegebenen nichtleeren Menge S das maximale Element m bestimmt. Wie immer beginnt eine Synthese mit der Definition eines neuen Prädikats max, das wir über eine Äquivalenz einführen.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow\ \ max(S,m) \ \Leftrightarrow\ (m\in S \ \wedge\ \forall x\in S.\, x\leq m)$$

S wird als Input-Variable und m als Output-Variable gekennzeichnet. Die Ausgabebedingung ist nicht auswertbar und bedarf daher einer Transformation.

5.4.2.1 GUESS-DOMAIN

Als ein typischer KI-Ansatz beginnt eine LOPS-Synthese mit dem Versuch, Informationen über die Ausgabewerte zu *raten*. Ein solches Raten kann erfolgreich sein, was bedeutet, daß man entweder die gesamte Lösung oder zumindest einen Teil davon geraten hat, oder fehlschlagen, was uns erlaubt, die geratenen Werte aus der Menge der zu betrachtenden Möglichkeiten zu streichen. In jedem Falle bedeutet das Raten aber einen Informationsgewinn für die weiteren Syntheseschritte.

Beispiel 5.4.9

Für die Lösung des Maximum-Problems können wir versuchen, die gesamte Lösung zu raten. Wir führen hierzu eine neue "Guess"-Variable g ein und drücken die Tatsache, daß wir entweder erfolgreich geraten haben oder nicht, durch die Tautologie $g{=}m \vee g{\neq}m$ aus.

Natürlich müssen die Möglichkeiten für die Auswahl von g auf irgendeine Weise limitiert werden, die mit der ursprünglichen Problemstellung zu tun hat, da ansonsten aus algorithmischer Sicht nichts gewonnen wird. Wir begrenzen daher das Raten auf Werte $g\in S$ und nehmen diese "Domain"-Bedingung für g zusätzlich in die Formel mit auf. Insgesamt wird das Maximum-Problem durch den Rateschritt in die folgende Formel transformiert.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow\ \ max(S,m) \ \Leftrightarrow\ \exists g{:}\mathbb{Z}.\, g\in S \ \wedge\ m\in S \wedge \forall x\in S.x\leq m \ \wedge\ (g{=}m \vee g{\neq}m)$$

Nun trennen wir die beiden Fälle auf, indem wir die Disjunktion $g{=}m \vee g{\neq}m$ über die ursprüngliche Ausgabebedingung distributieren.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow$$
$$max(S,m) \ \Leftrightarrow\ \exists g{:}\mathbb{Z}.\, g\in S \wedge\ (m\in S \wedge \forall x\in S.x\leq m \wedge g{=}m) \ \vee\ (m\in S \wedge \forall x\in S.x\leq m \wedge g{\neq}m)$$

Damit ist klar, wie das Problem gelöst werden muß: wir müssen zunächst einen Wert $g\in S$ bestimmen und dann die beiden Fälle einzeln lösen, wobei wir g nun als Eingabewert betrachten dürfen. Um dies deutlicher zu machen, führen wir zwei neue Prädikate max1 und max2 ein, welche den Erfolgs- bzw. Mißerfolgsfall kennzeichnen und zerlegen wir das Problem in drei Teilprobleme.

$$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \ \Rightarrow\ \ max(S,m) \ \Leftrightarrow\ \exists g{:}\mathbb{Z}.\ g\in S \ \wedge\ (max1(S,g,m) \vee max2(S,g,m))$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\neq\emptyset \ \wedge\ g\in S \ \Rightarrow\ \ max1(S,g,m) \ \Leftrightarrow\ m\in S \ \wedge\ \forall x\in S.\, x\leq m \ \wedge\ g{=}m$$
$$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\neq\emptyset \ \wedge\ g\in S \ \Rightarrow\ \ max2(S,g,m) \ \Leftrightarrow\ m\in S \ \wedge\ \forall x\in S.\, x\leq m \ \wedge\ g{\neq}m$$

Die Lösung des ersten Problems steht fest, sobald die beiden anderen gelöst sind, da es leicht ist, einen Wert $g\in S$ zu berechnen. Das zweite Problem ist trivial zu lösen, da hier die Ausgabe m identisch mit der Eingabe g sein soll. Uns bleibt also noch die Lösung des Mißerfolgsfalls, wobei uns nun erheblich mehr Informationen zur Verfügung stehen als zu Beginn der Synthese.

Aus syntaktischer Sicht ist Raten also dasselbe wie die Einführung einer neuen Guess-Variablen g. Dieses Raten muß einerseits mit dem ursprünglichen Problem in Beziehung gesetzt werden, damit man die geratenen Werte auch verwenden kann, und andererseits auf irgendeine Weise limitiert werden, damit es algorithmisch einigermaßen effizient durchgeführt werden kann.

Für das erste muß man eine mögliche Relation[13] t zwischen der Guess-Variablen g und der Ausgabevariablen y fixieren, die nach dem Raten entweder erfüllt ist oder nicht. Im einfachsten Fall ist dies, wie in Beispiel 5.4.9 illustriert, die Gleichheit. Wenn die Ausgaben jedoch strukturierte Objekte wie Listen, Mengen, Bäume etc. sind, kann man kaum davon ausgehen, die gesamte Ausgabe in einem Schritt zu raten. Daher muß es auch möglich sein, nur einen Teil der Ausgabe zu raten, den man z.B. in einem Schritt auf dem Bildschirm darstellen könnte, und die Relation t entsprechend als g∈S, g=first(S), etc. zu wählen. Da hierfür nur wenige sinnvolle Möglichkeiten bestehen, die im wesentlichen nur von der Struktur der Ausgaben abhängt, empfiehlt es sich, die Informationen über zulässige *Guess-Schemata* in einer Wissensbank abzulegen.

Definition 5.4.10 (Guess-Schema)

Ein Guess-Schema ist ein 4-Tupel $(R, G, t, \mathit{displayable})$*, wobei* R *und* G *Datentypen sind,* $t\colon R\times G\to\mathbb{B}$*,* $\mathit{displayable}\colon R\to\mathbb{B}$ *und die Eigenschaft* $\forall \mathtt{y}{:}R.\ \mathit{displayable}(\mathtt{y}) \Leftrightarrow \exists \mathtt{g}{:}G.\, t(\mathtt{g},\mathtt{y})$ *erfüllt ist.*

Ein Guess-Schema enthält also alle wichtigen Standardinformationen über mögliche Arten, Informationen über einen Ausgabewert zu beschreiben. Während einer Synthese wird ein solches Schema passend zu einem vorgegebenen Ausgabetyp R ausgewählt und dann instantiiert. Die Komponente *displayable* ist eine feste Zusatzinformation, die benötigt wird um festzustellen, ob GUESS überhaupt anwendbar ist oder ob zunächst ein Vorverarbeitungsschritt (siehe Abschnitt 5.4.2.3) durchgeführt werden muß, weil manche Ausgabewerte keine darstellbaren Anteile besitzen, die man raten könnte. Sie könnte im Prinzip aus t hergeleitet werden. Es ist aber effizienter, sie im Schema mit abzuspeichern.

Während die Beziehung zwischen Guess-Variable g und Ausgabevariable y also aus der Wissensbank genommen wird, muß die Begrenzung des Ratens durch eine *Domain-Condition* an g heuristisch bestimmt werden. Eine Heuristik *DOMAIN* muß hierzu eine Bedingung DC bestimmen, die es erlaubt, die geratenen Werte effizient zu bestimmen und die Möglichkeit falscher Rateschritte klein zu halten, ohne dabei aber darstellbare Anteile möglicher Lösungen aus der Betrachtung zu eliminieren. Präzisiert man diese Bedingungen, so ergeben sich folgende Anforderungen an die Heuristik DOMAIN.

1. Raten muß berechenbar sein, d.h. es muß gelten

   FUNCTION $\mathtt{F}_D C(x{:}D){:}G$ WHERE $I[x]$ RETURNS y SUCH THAT $DC(x,y)$ ist erfüllbar

   Dies beschränkt die heuristische Suche auf die Auswahl von Prädikaten, die *als erfüllbar bekannt* sind.

2. Alle darstellbaren Anteile möglicher Lösungen müssen durch Raten erreichbar sein, d.h. es muß gelten

   $\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\ I[\mathtt{x}] \Rightarrow (\exists \mathtt{y}{:}R.\ O[\mathtt{x},\mathtt{y}] \wedge t(\mathtt{g},\mathtt{y})) \Rightarrow DC(\mathtt{x},\mathtt{g})$[14]

   Dies bedeutet, daß sich $DC(\mathtt{x},\mathtt{g})$ normalerweise als eine Vereinfachung der Formel $O[\mathtt{x},\mathtt{y}] \wedge t(\mathtt{g},\mathtt{y})$ ergibt, wozu normalerweise (limitierte) Vorwärtsinferenzen eingesetzt werden: man versucht aus der Formel $O[\mathtt{x},\mathtt{y}] \wedge t(\mathtt{g},\mathtt{y})$ alle Eigenschaften von x und g zu bestimmen, die sich durch Anwendung weniger Lemmata beweisen lassen. Im einfachen Fall ($t(\mathtt{g},\mathtt{y})$ ist g=y) wählt man üblicherweise eine *echte* Teilmenge der Konjunkte aus $O(\mathtt{x},\mathtt{g})$.

3. Die Menge der zu ratenden Werte $\{\mathtt{g}{:}G \mid DC(\mathtt{x},\mathtt{g})\}$ sollte klein sein.

   Hierdurch wird insbesondere auch die Menge der Mißerfolge beim Raten, also $\{\mathtt{g}{:}G \mid DC(\mathtt{x},\mathtt{g}) \wedge \neg t(\mathtt{g},\mathtt{y})\}$ für jede Lösung y klein gehalten.

4. Raten sollte effizient sein, d.h. es sollte eine schnelle Lösung existieren für das Problem

   FUNCTION $\mathtt{F}_D C(x{:}D){:}G$ WHERE $I[x]$ RETURNS y SUCH THAT $DC(x,y)$

[13] Der Name t deutet an, daß diese Relation in tautologischer Weise zur Formel hinzugefügt wird.

[14] In diesem Fall gilt $\{\mathtt{g}{:}G \mid \exists \mathtt{y}{:}R.\ O[\mathtt{x},\mathtt{y}] \wedge t(\mathtt{g},\mathtt{y})\} \subseteq \{\mathtt{g}{:}G \mid DC(\mathtt{x},\mathtt{g})\}$

Die letzten beiden Kriterien dienen vor allem der Entscheidungshilfe, wenn mehrere Möglichkeiten für die Auswahl der Domain-Bedingung *DC* existieren. Sie geben Kriterien für einen Vergleich der offenstehenden Möglichkeiten an und verlangen Abschätzungen, die sich auf semantische Information stützen. Da derartige Informationen derzeit nur sehr unvollständig von einem automatischen System bestimmt werden können, ist in dieser Phase von DOMAIN ein Benutzereingriff sinnvoll.

Für die erfolgreiche Durchführung eines Rateschrittes sind insgesamt also die Wahl eines Guess-Schemas der Wissensbank und die heuristische, zum Teil benutzerunterstützte Auswahl der Domain-Bedingung erforderlich. Die eigentliche Guess-Transformation auf der Basis dieser Informationen verläuft dann ganz schematisch ab: es wird die Domain-Bedingung und die Tautologie zu der Ausgabebedingung ergänzt, die Disjunktion über die Ausgabebedingung distributiert und schließlich das ganze in drei Teilprobleme zerlegt. All dies läßt sich wie folgt in einem Schritt zusammenfassen.

Definition 5.4.11 (GUESS-Transformation)

Es sei spec=(D, R, I, O) *eine beliebige Spezifikation, GS* = $(R, G, t,$ *displayable*$)$ *ein Guess-Schema und* $DC\colon D\times G \to \mathbb{B}$. *Dann ist die durch GS und DC bestimmte GUESS-Transformation definiert als Transformation der Spezifikationsformel*

$$\forall \mathtt{x}\colon D.\, \forall \mathtt{y}\colon R.\; I[\mathtt{x}] \quad\Rightarrow\quad \mathtt{F(x,y)} \;\Leftrightarrow\; O[\mathtt{x},\mathtt{y}]$$

in die Formelmenge

$$\forall \mathtt{x}\colon D.\, \forall \mathtt{y}\colon R.\; I[\mathtt{x}] \;\Rightarrow\; \mathtt{F(x,y)} \;\Leftrightarrow\; \exists \mathtt{g}\colon G.\; DC(\mathtt{x},\mathtt{g}) \wedge (\mathtt{F}_1(\mathtt{x},\mathtt{g},\mathtt{y}) \vee \mathtt{F}_2(\mathtt{x},\mathtt{g},\mathtt{y}))$$
$$\forall \mathtt{x}\colon D.\, \forall \mathtt{g}\colon G.\, \forall \mathtt{y}\colon R.\; I[\mathtt{x}] \wedge DC(\mathtt{x},\mathtt{g}) \;\Rightarrow\; \mathtt{F}_1(\mathtt{x},\mathtt{g},\mathtt{y}) \;\Leftrightarrow\; O[\mathtt{x},\mathtt{y}] \wedge t(\mathtt{g},\mathtt{y})$$
$$\forall \mathtt{x}\colon D.\, \forall \mathtt{g}\colon G.\, \forall \mathtt{y}\colon R.\; I[\mathtt{x}] \wedge DC(\mathtt{x},\mathtt{g}) \;\Rightarrow\; \mathtt{F}_2(\mathtt{x},\mathtt{g},\mathtt{y}) \;\Leftrightarrow\; O[\mathtt{x},\mathtt{y}] \wedge \neg \mathtt{t}(\mathtt{g},\mathtt{y})$$

Es läßt sich verhältnismäßig einfach nachweisen, daß diese Transformation tatsächlich äquivalenzerhaltend ist.

5.4.2.2 GET-REC

Nach der Anwendung von GUESS-DOMAIN sind nur noch die beiden neu definierten Teilprobleme zu untersuchen. Von diesen ist normalerweise eines entweder trivial erfüllbar wie imm Falle des Maximum-Problems oder relativ schnell als unlösbar nachzuweisen. Eine Überprüfung dieser beiden Möglichkeiten wird daher unmittelbar im Anschluß an GUESS-DOMAIN durchgeführt. Für die verbleibenden ungelösten Teilprobleme wird nun versucht, Rekursion einzuführen.

Beispiel 5.4.12

Nach der GUESS-Phase ist nur noch das zweite Teilproblem `max2` des Maximum-Problems ungelöst.

$$\forall \mathtt{S}\colon \mathtt{Set}(\mathbb{Z}).\, \forall \mathtt{g},\mathtt{m}\colon \mathbb{Z}.\; \mathtt{S}\neq\emptyset \;\wedge\; \mathtt{g}\in\mathtt{S} \quad\Rightarrow\quad \mathtt{max2(S,g,m)} \;\Leftrightarrow\; \mathtt{m}\in\mathtt{S} \;\wedge\; \forall \mathtt{x}\in\mathtt{S}.\, \mathtt{x}\leq\mathtt{m} \;\wedge\; \mathtt{g}\neq\mathtt{m}$$

Wir versuchen nun, die rechte Seite der Äquivalenz in eine rekursive Variante der Ausgangsformel $\mathtt{m}\in\mathtt{S} \wedge \forall \mathtt{x}\in\mathtt{S}.\mathtt{x}\leq\mathtt{m}$ umzuschreiben. Dazu benötigen wir natürlich Informationen darüber, welche Arten von Rekursion auf einer endlichen Menge `S` im Zusammenhang mit einem Element $\mathtt{g}\in\mathtt{S}$ überhaupt möglich sind, um eine terminierende rekursive Beschreibung des Problems zu erlauben.

Ein einfache sallgemeines Schema zur Reduktion endlicher Mengen mit Hilfe eines Elementes besteht darin, dieses Element aus der Menge hinauszunehmen. Ein solcher Schritt kann nur endlich oft wiederholt werden, bis die Menge leer ist, was bedeutet, daß eine Rekursion wohlfundiert ist.

Wir versuchen daher, Lemmata zu finden, mit denen wir die Formel $\mathtt{m}\in\mathtt{S} \wedge \forall \mathtt{x}\in\mathtt{S}.\, \mathtt{x}\leq\mathtt{m} \wedge \mathtt{g}\neq\mathtt{m}$ im Kontext aller Vorbedingungen in eine Formel der Gestalt $\mathtt{m}\in\mathtt{S\text{-}g} \wedge \forall \mathtt{x}\in\mathtt{S\text{-}g}.\, \mathtt{x}\leq\mathtt{m} \wedge \ldots$

transformieren können. Zuvor müssen wir jedoch auch die Eingabebedingung $\mathtt{S}\neq\emptyset$ mit Hilfe von Lemmata zerlegen in die Form $\mathtt{S\text{-}g}\neq\emptyset \vee \ldots$. Damit können wir die Eingabewerte bestimmen, die für eine rekursive Verarbeitung zulässig sind, während wir die anderen direkt behandeln müssen.

Wir suchen also in der Wissensbank gezielt nach Lemmata über Mengendifferenz und leere Menge (B.1.10.7, negiert), Elementrelation (B.1.9.5) und beschränkten Allquantor (B.1.11.6):

B.1.10.7	$S-g=\emptyset \Leftrightarrow S=\{g\} \vee S=\emptyset$
negiert	$S-g\neq\emptyset \Leftrightarrow S\neq\{g\} \wedge S\neq\emptyset$
Nach Ergänzung der Disjunktion $S=\{g\}$:	$S=\{g\} \vee S-g\neq\emptyset \Leftrightarrow S\neq\emptyset$
B.1.9.5	$m\in S-g \Leftrightarrow g\neq m \wedge m\in S$
B.1.11.6	$\forall x\in S-g.\, x\leq m \wedge g\leq m \Leftrightarrow \forall x\in S.\, x\leq m$

Nach Anwendung dieser Lemmata und Zerlegung der Disjunktion in der Eingabebedingung erhalten wir

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S=\{g\} \wedge g\in S \Rightarrow max2(S,g,m) \Leftrightarrow m\in S \wedge \forall x\in S.\, x\leq m \wedge g\neq m$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S-g\neq\emptyset \wedge g\in S \Rightarrow max2(S,g,m) \Leftrightarrow m\in S-g \wedge \forall x\in S-g.\, x\leq m \wedge g\neq m \wedge g\leq m$

In der ersten Teilformel stellt sich sofort heraus, daß die Bedingungen $m\in S \wedge g\neq m$ unter der Voraussetzung $S=\{g\}$ widersprüchlich sind. In der zweiten Teilformel können wir nun die Definition des Prädikates max mit S-g und m zurückfalten, da Ein- und Ausgabebedingung in rekursiver Form vorliegen. Zusätzlich fassen wir $g\neq m \wedge g\leq m$ zusammen. Damit ergeben sich insgesamt die folgenden vier Teilformeln

$\forall S{:}Set(\mathbb{Z}).\forall m{:}\mathbb{Z}.\ S\neq\emptyset \Rightarrow max(S,m) \Leftrightarrow \exists g{:}\mathbb{Z}.\ g\in S \wedge (max1(S,g,m) \vee max2(S,g,m))$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S\neq\emptyset \wedge g\in S \Rightarrow max1(S,g,m) \Leftrightarrow \forall x\in S.\, x\leq g \wedge g=m$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S=\{g\} \wedge g\in S \Rightarrow max2(S,g,m) \Leftrightarrow false$

$\forall S{:}Set(\mathbb{Z}).\forall g,m{:}\mathbb{Z}.\ S-g\neq\emptyset \wedge g\in S \Rightarrow max2(S,g,m) \Leftrightarrow max(S-g,m) \wedge g<m$

wobei wir die Ausgabebedingung von max1 ebenfalls durch Substitutionen und Entfernung von Redundanz vereinfacht haben. Nun erweisen sich alle Bestandteile als auswertbar und wir können mit der Strategie 5.4.2 das folgende Logikprogramm zur Lösung des Maximumproblems erzeugen.

```
max(S,M)       :- member(G,S), max-aux(S,G,M).
max-aux(S,G,M) :- setminus(S,G,SminusG), max(SminusG,M), less(G,M),!.
max-aux(S,M,M) :- setless(S,M).
```

Im Gegensatz zu GUESS-DOMAIN besteht die Strategie GET-REC zur Einführung von Rekursion nicht aus einer festgelegten Transformation sondern ist durch ihr Ziel – die Einführung von Rekursion[15] – definiert. Dieses Ziel ist, genau besehen, eine *disjunktive Zerlegung der Eingabebedingung* in eine rekursive Variante der ursprünglichen Eingabebedingung und einer Bedingung an Eingaben, für die Rekursion nicht in Frage kommt, sowie eine *konjunktive Zerlegung der Ausgabebedingung* in eine rekursive Variante der ursprünglichen Ausgabebedingung und zusätzlichen Anforderungen. Zusätzlich soll erreicht werden, daß außer den Ausgangsbedingungen nur noch erfüllbare Prädikate vorkommen.

Die Transformation mit dem Ziel der Rekursion wird im wesentlichen durch eine Suche nach geeigneten Lemmata erreicht, in denen die Beziehung zwischen einer reduzierten und der nichtreduzierten Form der vorkommenden Prädikate behandelt wird. Dies verlangt vor allem nach einer guten Strukturierung der Wissensbank und einer entsprechenden Suchmethode, mit der anwendbare Lemmata schnell aufgespürt[16]

Wesentlich ist aber natürlich auch die Frage, welche Arten von Rekursion überhaupt möglich sind, um einen sinnvollenden terminierenden Algorithmus zu generieren. Da Rekursionseinführung nur dann Sinn macht, wenn auch die geratene Information zur Problemreduktion verwendet wird, wird die Art der Rekursion im allgemeinen von der Guessvariablen und der Eingabevariablen abhängen, also aus einer Reduktionsfunktion bestehen, die zu $x\in D$ und $g\in G$ einen Wert $x\searrow g$ bestimmt, der kleiner ist als x. Natürlich muß diese Funktion*wohlfundiert* sein, da sonst keine Terminierungsgarantie gegeben werden könnte. Da letzteres zur Laufzeit einer Synthese eine sehr hohe Belastung des Inferenzsystems bedeuten würde und Rekursionen normalerweise ohnehin nur von den Typen D der Eingabewerte und G der Guessvariablen abhängen, empfiehlt es sich, auch diese Information als Rekursionsschema in einer Wissensbank abzulegen.

Definition 5.4.13 (Rekursionsschema)

Ein Rekursionsschema ist ein Tripel $(D, G, \searrow)$*, wobei* D *und* G *Datentypen sind und* $\searrow: D\times G\to D$ *eine wohlfundierte Reduktionsfunktion.*

[15]Der Name *GET-REC* Goal-oriented Equivalence Transformation) deutet an, daß die Transformation zielorientiert ist.

[16]Eine gute Methode ist die Verwendung von Hashtabellen, die nach dem Kriterium äüserer/innerer Operator erstellt werden. Die Lemmata des Anhang B sind nach einem solchen Kriterium sortiert.

Diese Definition behandelt nur die einfache Form einer Rekursion. Im Allgemeinfall kann eine rekursive Zerlegung einer Eingabe auch mehrere "kleinere" Eingabewerte erzeugen (vergleiche die Diskussion auf Seite 236). Die allgemeinste Form eines Rekursionsschemas besteht daher aus einer wohlfundierten Reduktionsfunktion $\searrow$: $D \times G \rightarrow \mathtt{Set}(D)$, was formal allerdings etwas aufwendiger zu handhaben ist. Unter Verwendung all dieser Informationen geht die Strategie GET-REC insgesamt wir folgt vor.

Strategie 5.4.14 (GET-REC)
Es sei $\forall \mathtt{x}{:}D.\forall \mathtt{y}{:}R.\ I[\mathtt{x}] \Rightarrow \mathtt{F(x,y)} \Leftrightarrow O[\mathtt{x},\mathtt{y}]$ *die Ausgangsformel der Synthese und G der Typ der in der GUESS-Phase bestimmten zusätzlichen Variablen. Die Strategie GET-REC transformiert eine gegebene Spezifikationsformel der Gestalt* $\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \mathtt{I'(x,g)} \Rightarrow \mathtt{F'(x,g,y)} \Leftrightarrow O'[\mathtt{x},\mathtt{g},\mathtt{y}]$ *mit den folgenden Schritten in eine rekursive Form*

1. *Wähle ein Rekursionsschema* $(D, G, \searrow)$ *aus der Wissensbank.*

2. *Durch gezielte Suche nach anwendbaren Lemmata über die Beziehung zwischen einer reduzierten und der nichtreduzierten Form der vorkommenden Prädikate schreibe die Eingabebedingung um in eine Formel der Gestalt* $\mathtt{I}_b\mathtt{(x,g)} \vee I[\mathtt{x}\searrow\mathtt{g}\,/\,\mathtt{x}]$ *und die Ausgabebedingung um in eine Formel der Gestalt* $\exists \mathtt{y}_r\,.\,O[\mathtt{x}\searrow\mathtt{g},\mathtt{y}_r\,/\,\mathtt{x},\mathtt{y}]) \wedge \mathtt{O}_r\mathtt{(x,g,y}_r\mathtt{,y)}$

3. *Spalte die nichtrekursiven Lösungen ab, ersetze die Konjunktionsglieder aus O durch* F *und erzeuge insgesamt die Äquivalenzformeln*
   $\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \mathtt{I}_b\mathtt{(x,g)} \Rightarrow \mathtt{F'(x,g,y)} \Leftrightarrow O'[\mathtt{x},\mathtt{g},\mathtt{y}]$
   $\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \mathtt{I(x}\searrow\mathtt{g)} \Rightarrow \mathtt{F'(x,g,y)} \Leftrightarrow \exists \mathtt{y}_r\,.\,\mathtt{F(x}\searrow\mathtt{g,y}_r\mathtt{)} \wedge \mathtt{O}_r\mathtt{(x,g,y}_r\mathtt{,y)}$

4. *Vereinfache die entstandenen Formeln soweit wie möglich und überprüfe die Auswertbarkeit.*

5.4.2.3 Unterstützende Strategien

Die Strategien GUESS und GET-REC bilden den Kernbestandteil jeder mit LOPS durchgeführten Synthese. In einfachen Fällen reichen sie in der bisher beschriebenen Form aus, um das vorgegebene Problem zu lösen. Normalerweise sind jedoch einige unterstützende Techniken notwendig, um eine LOPS-Synthese erfolgreich zum Ziel zu führen. Neben einigen allgemeinen Techniken wie *Normalisierung*, *Simplifikation* durch gerichtete Lemmata (vergleiche Abschnitt 5.6.1), *Unterproblem-Erzeugung* für komplexere Ausdrücke und Ersetzung nichtauswertbarer Prädikate durch äquivalente auswertbare Prädikate (GET-EP) geht es hierbei vor allem um Strategien zur Unterstützung von GUESS-DOMAIN im zusammenhang mit strukturierten Ausgabewerten.

So kann es zum Beispiel bei induktiven Datenstrukturen Ausgabewerte ohne darstellbare Anteile geben, die durch einen Vorverarbeitungsschritt (*Preprocessing*) abgespalten und separat gelöst werden müssen. Bei mehreren Ausgabewerten (Produkten) ist es sinnvoller, das Raten auf einen Ausgabewert zu fokussieren. Hierzu muß man entweder die Anzahl der Ausgabevariablen durch Bestimmung funktionaler Abhängigkeiten *reduzieren* (GET-RNV), die Ausgabebedingungen separieren, um die Ausgabevariablen separat behandeln zu können (GET-SOC), oder eine hierarchische Lösung versuchen, indem zunächst die bestgeeigneteste Ausgabevariable für das Raten ausgewählt wird (CHVAR) und dann in einer inneren Schleife die anderen Ausgabevariabln behandelt werden, wobei die Abhängigkeiten zwischen den Ausgabevariablen ebenfalls berücksichtigt werden (DEPEND). Wir wollen diese Unterstützungsstrategien im folgenden kurz besprechen.

Preprocessing für GUESS

Bei der bisherigen Beschreibung der Strategie GUESS sind wir davon ausgegangen, daß das Raten einer Lösung bzw. eines Teils davon, im Prinzip immer möglich ist. Für induktive Ausgabedatentypen ist dies jedoch nicht immer der Fall, da zu diesen meist auch ein "leeres" Basisobjekt gehört, welches keinerlei darstellbare Anteile besitzt. Derartige Sonderfälle müssen daher durch einen Vorverarbeitungsschritt abgespalten werden, bevor das eigentliche Raten beginnen kann.

Technisch bedeutet dies, daß man zunächst diejenigen Eingabewerte bestimmen muß, die zu einem Ausgabewert ohne darstellbare Anteile gehören. Während für alle anderen Eingaben die übliche GUESS-DOMAIN Strategie anwendbar ist, muß für diese "primitiven" Werte eine direkte eine Lösung bestimmt werden. Im Allgemeinfall ist dies jedoch nicht sehr schwer, da die Menge $\{\mathtt{y}{:}R \mid \neg\mathtt{displayable(y)}\}$ der Ausgabewerte ohne darstellbare Anteile meist einelementig ist.

Die zentrale Aufgabe des Vorverarbeitungsschrittes ist somit, die primitiven Eingaben zu identifizieren, also ein Prädikat `prim` zu bestimmen mit den Eigenschaften

$\forall\mathtt{x}{:}D.\ (\exists\mathtt{y}{:}R.\ I[\mathtt{x}] \wedge O[\mathtt{x},\mathtt{y}] \wedge \neg\mathtt{displayable(y)}) \ \Rightarrow\ \mathtt{prim(x)}$

$\forall\mathtt{x}{:}D.\ I[\mathtt{x}] \wedge \mathtt{prim(x)} \ \Rightarrow\ \exists\mathtt{y}{:}R.\ O[\mathtt{x},\mathtt{y}]$

Im allgemeinen bestimmt man `prim` durch Vereinfachung der Formel $I[\mathtt{x}] \wedge O[\mathtt{x},\mathtt{y}] \wedge \neg\mathtt{displayable(y)}$, wozu normalerweise (limitierte) Vorwärtsinferenzen[17] eingesetzt werden. Mit diesem Prädikat transformiert man nun die Spezifikationsformel

$\forall\mathtt{x}{:}D.\forall\mathtt{y}{:}R.\ I[\mathtt{x}] \ \Rightarrow\ \mathtt{F(x,y)} \ \Leftrightarrow\ O[\mathtt{x},\mathtt{y}]$

in die Formelmenge

$\forall\mathtt{x}{:}D.\forall\mathtt{y}{:}R.\ I[\mathtt{x}] \wedge \mathtt{prim(x)} \ \Rightarrow\ \mathtt{F(x,y)} \ \Leftrightarrow\ O[\mathtt{x},\mathtt{y}]$

$\forall\mathtt{x}{:}D.\forall\mathtt{y}{:}R.\ I[\mathtt{x}] \wedge \neg\mathtt{prim(x)} \ \Rightarrow\ \mathtt{F(x,y)} \ \Leftrightarrow\ O[\mathtt{x},\mathtt{y}]$

Das erste dieser beiden Teilprobleme besitzt üblicherweise eine sehr einfache Lösung – nämlich genau das eine `y`, für das ¬displayable(y) gilt. Das zweite Teilproblem kann nun in der bisher bekannten Weise weiterverarbeitet werden. Wir wollen diese Technik an dem Vorverarbeitungsschritt bei der Synthese von Sortieralgorithmen illustrieren.

Beispiel 5.4.15 (Synthese von Sortieralgorithmen – Vorverarbeitung)

Das Problem der Sortierung von Listen ganzer Zahlen besteht darin, zu einer gegebenen Liste eine Umordnung der Elemente zu finden, die geordnet ist. Dabei ist eine Liste L *geordnet*, wenn ihre Elemente in aufsteigender Reihenfolge angeordnet sind, was sich wie folgt formal definieren läßt

$\mathtt{ordered}(L) \ \equiv\ \forall\mathtt{i} \in \{\mathtt{1..}|L|\mathtt{-1}\}.L[\mathtt{i}] \leq L[\mathtt{i+1}]$

Da die Umordnung von Listen durch das in Anhang B.2 definierte boolesche Prädikat `rearranges` repräsentiert wird, können wir das Sortierproblem nun durch die folgende Äquivalenzformel pezifizieren.

$\forall\mathtt{L,S:Seq}(\mathbb{Z}).\ \mathtt{true} \ \Rightarrow\ \underline{\mathtt{SORT(L,S)}} \ \Leftrightarrow\ \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)}$

Jedes GUESS-Schema der Wissensbank muß die Ausgabewerte mit darstellbaren Anteilen charakterisieren. Unabhängig von der konkreten Tautologierelation t sind dies genau die nichtleeren Listen ganzer Zahlen, d.h. es gilt $\neg\mathit{displayable}(\mathtt{S}) \equiv \mathtt{S=[]}$. Bevor wir also die Strategie GUESS-DOMAIN auf das Sortierproblem anwenden können, müssen wir den Bereich der "primitiven" Eingaben bestimmen, also ein Prädikat $\mathtt{prim:Seq}(\mathbb{Z})\rightarrow\mathbb{B}$ herleiten mit der Eigenschaft

$\exists\mathtt{S:Seq}(\mathbb{Z}).\ \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)} \wedge \mathtt{S=[]} \ \Rightarrow\ \mathtt{prim(L)}$

Die Substitution von `S` durch `[]` und die Anwendung des Lemmas über die Umordnung leerer Listen (Lemma B.2.26.1) liefert $\mathtt{prim(L)} \equiv \underline{\mathtt{L=[]}}$. Die zweite Bedingung

$\mathtt{L=[]} \ \Rightarrow\ \exists\mathtt{S:Seq}(\mathbb{Z}).\ \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)}$

hat trivialerweise die Lösung $\mathtt{S}\equiv\mathtt{[]}$, da es nur eine Art von Listen ohne darstellbare Anteile gibt. Nach der Vorverarbeitung muß nun das folgende Teilproblem des Sortierproblems mit der üblichen LOPS-Strategie gelöst werden.

$\forall\mathtt{L,S:Seq}(\mathbb{Z}).\ \mathtt{L}\neq\mathtt{[]} \ \Rightarrow\ \underline{\mathtt{SORT(L,S)}} \ \Leftrightarrow\ \mathtt{rearranges(L,S)} \wedge \mathtt{ordered(S)}$

[17] Diese Strategie könnte zum Beispiel mit Hilfe einer allgemeinen Technik zur Bestimmung der *Vorbedingungen* für die Gültigkeit einer logischen Formel, die in [Smith, 1985b, Section 3.2] beschrieben ist, automatisiert werden.

GET-RNV – Reduce Number of Output Variables

Bei Syntheseproblemen mit mehreren Ausgabewerten würde das raten verhältnismäßig kompliziert, wenn gleichzeitig alle Ausgabewerte geraten werden müssten. Die Strategie GUESS ist daher darauf ausgelegt, jeweils nur einen einzelnen Ausgabewert zu verarbeiten. Um dies zu unterstützen, versucht die Strategie GET-RNV, die Anzahl der Ausgabevariablen des Problems dadurch zu verringern, daß *unmittelbar erkennbare Abhängigkeiten* zwischen den Variablen aufgespürt werden und eine *Ausgabevariable als Funktion der anderen Variablen* beschrieben wird. Dies bedeutet, daß eine Spezifikationsformel der Form

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y},\mathtt{y'}){:}R{\times}R'.\ I[x] \quad\Rightarrow\quad \mathtt{F(x,y,y')} \;\Leftrightarrow\; O[\mathtt{x,y,y'}]$$

transformiert wird in eine Formelmenge der Art

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y},\mathtt{y'}){:}R{\times}R'.\ I[x] \;\Rightarrow\; \mathtt{F(x,y,y')} \;\Leftrightarrow\; \mathtt{F'(x,y)} \;\wedge\; \mathtt{y'{=}g[x,y]}$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{y}{:}R.\ I[x] \;\Rightarrow\; \mathtt{F'(x,y)} \;\Leftrightarrow\; O[\mathtt{x,y,g[x,y]\,/\,x,y,y'}]$$

Die Bestimmung der funktionalen Abhängigkeit `y'=g[x,y]` beschränkt sich hierbei auf Zusammenhänge, die unmittelbar mit Hilfe weniger Lemmata nachzuweisen sind. Ansonsten ist diese Strategie nicht anwendbar.

Beispiel 5.4.16 (Reduktion des FIND-Problems)

Das Problem, zu einer gegebenen Menge `S` von ganzen Zahlen das `i`-t-größte Element `a` zu bestimmen, ist nicht ganz leicht zu formalisieren, wenn man nur die elementaren Mengenoperationen (ohne den allgemeinen Set-Former) zur Verfügung stehen hat. In diesem Falle lautet die naheliegenste Formalisierung, daß `S` in zwei Mengen `S1` und `S2` sowie das Element `a` zerlegbar sein muß, wobei `S1` nur aus Elementen besteht, die kleiner sind als `a`, und `S2` nur aus größeren Elementen, wobei `S2` genau `i-1` Elemente haben muß. Diese Charakterisierung führt zu folgender Spezifikation des FIND-Problems

$$\forall \mathtt{S,i}.\forall \mathtt{S1,a,S2}.\ \mathtt{i}\in\{\mathtt{1..|S|}\} \;\Rightarrow\; \mathtt{FIND((S,i),(S1,a,S2))}$$
$$\Leftrightarrow\; \mathtt{S{=}S1}\cup\{\mathtt{a}\}\cup\mathtt{S2} \;\wedge\; \forall \mathtt{x}\in\mathtt{S1.x<a} \;\wedge\; \forall \mathtt{x}\in\mathtt{S2.a<x} \;\wedge\; \mathtt{|S2|{=}i{-}1}$$

Dieses Problem enthält drei Ausgabevariablen, von denen zumindest eine redundant ist, da sie vollständig durch `S, a` und die andere Ausgabemenge beschrieben werden kann. Genau dies festzustellen ist die Aufgabe von GET-RNV: da `S1, S2` und `{a}` disjunkt sind, läßt sich `S1` sich darstellen als `S1 = S\(S2+a)`, was zu folgender reduzierten Form des Problems führt.

$$\forall \mathtt{S,i}.\forall \mathtt{S1,a,S2}.\ \mathtt{i}\in\{\mathtt{1..|S|}\} \;\Rightarrow\; \mathtt{FIND((S,i),(S1,a,S2))}$$
$$\Leftrightarrow\; \mathtt{FIND'((S,i),(S1,a))} \;\wedge\; \mathtt{S1{=}}\boxed{\mathtt{S\backslash(S2{+}a)}}$$

$$\forall \mathtt{S,i}.\forall \mathtt{S2,a}.\ \mathtt{i}\in\{\mathtt{1..|S|}\} \;\Rightarrow\; \mathtt{FIND'((S,i),(S2,a))}$$
$$\Leftrightarrow\; \mathtt{a}\in\mathtt{S} \;\wedge\; \mathtt{S2}\subseteq\mathtt{S{-}a} \;\wedge\; \forall \mathtt{x}\in\boxed{\mathtt{S\backslash(S2{+}a)}}\mathtt{.a<x} \;\wedge\; \forall \mathtt{x}\in\mathtt{S2.x>a} \;\wedge\; \mathtt{|S2|{=}i{-}1}$$

GET-SOC – Separate Output Conditions

Da die Strategie GET-RNV schnell an die Grenzen ihrer Anwendungsmöglichkeiten stößt, versucht die Strategie GET-SOC als nächstes, die Ausgabebedingung O[x,y,y'] in unabhängige zu zerlegen, in denen jeweils nur eine der beiden Ausgabevariablen vorkommt. Dies bedeutet, daß eine Spezifikationsformel der Form

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y},\mathtt{y'}){:}R{\times}R'.\ I[x] \quad\Rightarrow\quad \mathtt{F(x,y,y')} \;\Leftrightarrow\; O[\mathtt{x,y,y'}]$$

transformiert wird in eine Formelmenge der Art

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y},\mathtt{y'}){:}R{\times}R'.\ I[x] \;\Rightarrow\; \mathtt{F(x,y,y')} \;\Leftrightarrow\; \mathtt{F1(x,y)} \;\wedge\; \mathtt{F2(x,y')}$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{y}{:}R.\ I[x] \;\Rightarrow\; \mathtt{F1(x,y)} \;\Leftrightarrow\; O_1[\mathtt{x,y}]$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{y'}{:}R'.\ I[x] \;\Rightarrow\; \mathtt{F2(x,y')} \;\Leftrightarrow\; O_2[\mathtt{x,y'}]$$

Diese Zerlegung von O[x,y,y'] in O_1[x,y] und O_2[x,y'] muß auf rein syntaktischer Basis durchführbar sein. Die einfachste Methode hierfür ist, jeweils diejenigen Konjunktionsglieder von O[x,y,y'] aufzusammeln, in denen ausschließlich y bzw. y' als Ausgabevariablen vorkommen. Wenn diese Technik versagt, ist die Strategie GET-SOC nicht anwendbar.

CHVAR – Choose Output-Variable

Bei Problemspezifikationen mit mehreren Ausgabewerten, bei denen weder GET-RNV noch GET-SOC zum Ziel führen, muß versucht werden, eine hierarchische Lösung des Problems zu generieren. Dies bedeutet, daß die Berechnung des zweiten Ausgabewertes ein Teilproblem während der Berechnung des ersten werden wird, wobei dieses Teilproblem erst nach der Anwendung von GUESS-DOMAIN auf die "äußere" Variable angegangen wird, also nachdem für diese Variable ein geratener Wert als zusätzliche Eingabe vorliegt. Die Hauptfrage, die hierfür zu klären ist, welche der Ausgabevariablen als erstes zu behandeln ist.

Das Verfahren CHVAR, mit dem diese Variable bestimmt wird, ist verhältnismäßig aufwendig und benötigt eine Reihe semantischer Zusatzinformationen, die eigentlich nur von einem Benutzer des Synthesesystems gegeben werden können. Bei einer Spezifikationsformel der Form

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y,y'}){:}R{\times}R'.\ \ I[x] \quad\Rightarrow\quad \mathtt{F(x,y,y')} \ \Leftrightarrow\ O[\mathtt{x,y,y'}]$$

Wählt man für jede der beiden Ausgabevariablen `y` und `y'` ein Guess-Schema aus der Wissensbank und führt die Heuristik DOMAIN aus. Anschließend vergleicht man die Anzahl der Werte der Ausgabebereiche R und R', die durch Raten erreicht werden können, also

$$|\,\{\mathtt{y}{:}R \mid \exists \mathtt{g}{:}G.\ DC(\mathtt{x,g}) \wedge t(\mathtt{g,y})\}\,| \quad\text{und}\quad |\,\{\mathtt{y'}{:}R' \mid \exists \mathtt{g'}{:}G'.\ DC'(\mathtt{x,g'}) \wedge t'(\mathtt{g',y'})\}\,|$$

und wählt diejenige Ausgabevariable, bei der diese Anzahl geringer ist. Dies hat zur Folge, daß bei der Bestimmung aller Lösungen des Problems die Anzahl der äußeren Rekursionen des generierten Algorithmus kleiner ist, was – hoffentlich – zu einem effizienteren Algorithmus führt.

DEPEND – Bestimme Abhängigkeiten der Ausgaben

Nachdem CHVAR die äußere Variable für GUESS-DOMAIN festgelegt hat und der äußere Rateschritt ausgeführt worden ist, versucht die Strategie DEPEND die Abhängigkeiten zwischen der äußeren und der inneren Ausgabevariablen zu bestimmen. Dieses Ziel ähnelt dem der Strategie GET-RNV, verlangt aber ein aufwendigeres Verfahren.

Es sei o.B.d.A. `y` die von CHVAR ausgewählte Ausgabevariable für den äußeren Rateschritt. Dann ist nach der Anwendung von GUESS-DOMAIN die folgende Formelmenge generiert worden.

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y,y'}){:}R{\times}R'.\ \ I[x] \quad\Rightarrow\quad \mathtt{F(x,y,y')} \ \Leftrightarrow\ \exists \mathtt{g}{:}G.\ DC(\mathtt{x,g}) \wedge (\mathtt{F}_1(\mathtt{x,g,y,y'}) \vee \mathtt{F}_2(\mathtt{x,g,y,y'}))$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall(\mathtt{y,y'}){:}R{\times}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F}_1(\mathtt{x,g,y,y'}) \ \Leftrightarrow\ O[\mathtt{x,y,y'}] \wedge t(\mathtt{g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall(\mathtt{y,y'}){:}R{\times}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F}_2(\mathtt{x,g,y,y'}) \ \Leftrightarrow\ O[\mathtt{x,y,y'}] \wedge \neg\mathtt{t(g,y)}$$

Die Strategie DEPEND versucht nun die maximale Teilbedingung von O[x,y,y'] zu bestimmen, die *beide* Teilprobleme lösbar macht, wenn `x,g` und `y` als Eingabevariablen vorausgesetzt werden.

Die Heuristik hierfür geht relativ grob vor: es wird die größte Menge O'[x,y,y'] von Konjunktionsgliedern aus O[x,y,y'] gewählt, in denen `y'` vorkommt. Anschließend wird versucht, die beiden reduzierten Probleme

$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \forall \mathtt{y'}{:}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F'}_1(\mathtt{x,g,y,y'}) \ \Leftrightarrow\ O'[\mathtt{x,y,y'}] \wedge t(\mathtt{g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ \forall \mathtt{y'}{:}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F'}_2(\mathtt{x,g,y,y'}) \ \Leftrightarrow\ O'[\mathtt{x,y,y'}] \wedge \neg\mathtt{t(g,y)}$$

mit Eingabevariablen `x,g` und `y` zu synthetisieren, wobei man sich wider auf "leicht zu findende" Lösungen konzentriert. Sind $\mathtt{f}_1$ und $\mathtt{f}_2$ die zugehörigen generierten Lösungsfunktionen, so wird – wie bei GET-RNV – im ursprünglichen Problem `y'` durch $\mathtt{f}_i(\mathtt{x,g,y})$ ersetzt, was zu folgender Formelmenge führt.

$$\forall \mathtt{x}{:}D.\forall(\mathtt{y,y'}){:}R{\times}R'.\ \ I[x] \quad\Rightarrow\quad \mathtt{F(x,y,y')} \ \Leftrightarrow\ \exists \mathtt{g}{:}G.\ DC(\mathtt{x,g}) \wedge (\mathtt{F}_1(\mathtt{x,g,y,y'}) \vee \mathtt{F}_2(\mathtt{x,g,y,y'}))$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall(\mathtt{y,y'}){:}R{\times}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F}_1(\mathtt{x,g,y,y'}) \ \Leftrightarrow\ \mathtt{F}_1^*(\mathtt{x,g,y}) \ \wedge\ \mathtt{y'{=}f}_1(\mathtt{x,g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F}_1^*(\mathtt{x,g,y}) \ \Leftrightarrow\ O[\mathtt{x,y,f}_1(\mathtt{x,g,y})\,/\,\mathtt{x,y,y'}] \wedge t(\mathtt{g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall(\mathtt{y,y'}){:}R{\times}R'.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F}_2(\mathtt{x,g,y,y'}) \ \Leftrightarrow\ \mathtt{F}_2^*(\mathtt{x,g,y}) \ \wedge\ \mathtt{y'{=}f}_2(\mathtt{x,g,y})$$
$$\forall \mathtt{x}{:}D.\forall \mathtt{g}{:}G.\forall \mathtt{y}{:}R.\ I[\mathtt{x}] \wedge DC(\mathtt{x,g}) \ \Rightarrow\ \mathtt{F}_2^*(\mathtt{x,g,y}) \ \Leftrightarrow\ O[\mathtt{x,y,f}_2(\mathtt{x,g,y})\,/\,\mathtt{x,y,y'}] \wedge \neg t(\mathtt{g,y})$$

5.4.2.4 Das Verfahren als Gesamtstrategie

Mit den soeben vorgestellten Zusatzstrategien lassen sich die Hauptstrategien GUESS-DOMAIN und GET-REC auf viele verschiedene Problemstellungen anpassen, wodurch das LOPS-Verfahren – zumindest aus theoretischer Sicht – zu einem recht mächtigen Syntheseverfahren wird. Wir fassen die bisher einzeln vorgestellten Strategien nun zu einer Gesamtstrategie zusammen.

Strategie 5.4.17 (LOPS–Verfahren)
Gegeben sei eine Problemspezifikation spec=(D, R, I, O).

1. *Definiere das Prädikat* `F` *durch* $\forall$`x`$:D.\forall$`y`$:R.\ I[x] \Rightarrow$ `F(x,y)` $\Leftrightarrow O[x,y]$ *und markiere* `x` *als Eingabe.*
2. *GUESS-Phase:*
   - *Besteht* `y` *aus mehreren Ausgabevariablen, so versuche eine Problemreduktion mit GET-RNV, GET-SOC oder CHVAR/DEPEND*

   (a) *Wähle ein GUESS-Schema* $(R, G, t,$ *displayable*$)$ *mit Ausgabetyp* R *aus der Wissensbank*

   (b) *Enthält* `R` *nichtdarstellbare Ausgaben, so spalte diese durch Bestimmung eines Prädikats* `prim` *ab.*

   (c) *Bestimme die Domain-Condition* `DC` *mit der Heuristik DOMAIN*

   (d) *Transformiere die Ausgangsformel durch Hinzufügen der geratenen Information und spalte das Problem in Teilprobleme gemäß Definition 5.4.11.*

   Markiere `g` *als Eingabevariable der neuen Hilfsprädikate und behandle alle Teilprobleme einzeln.*
3. *Teste die Auswertbarkeit und Widersprüchlichkeit einzelner Teilprobleme heuristisch. Der Nachweis muß in wenigen Schritten durchführbar sein.*

   Dabei führt einfaches Guessing immer zu trivial lösbaren Teilproblemen im Erfolgsfall und induktive Datenstrukturen oft zu widersprüchlichen Teilprobleme im Mißerfolgsfall.
4. *GET-REC-Phase (siehe Strategie 5.4.14 auf Seite 242)*

   (a) *Wähle ein Rekursionsschema* $(D, G, \searrow)$ *aus der Wissensbank.*

   (b) *Durch gezielte Suche nach Lemmata über die Beziehung zwischen einer reduzierten und der nichtreduzierten Form der vorkommenden Prädikate schreibe die Formel um in eine rekursive Variante der Ausgangsformel. Dabei kann eine Abspaltung nichtrekursiver Lösungen erforderlich sein.*
5. *Vereinfache die entstandenen Formeln soweit wie möglich und überprüfe die Auswertbarkeit. Starte LOPS erneut, wenn eine Formel nicht auswertbar ist.*
6. *Konstruiere aus der rekursiven Formel ein Programm*

Aus Sicht der Künstlichen Intelligenz ist das LOPS-Verfahren ein sehr vielseitiges und mächtiges Konzept, da sehr verschiedenartige Problemstellungen mit nur wenigen Grundstrategien auf recht elegante Weise gelöst werden können. Diese Eleganz gilt bisher jedoch nur auf dem Papier, da es in der Tat sehr viel Intelligenz benötigt, um eine LOPS-Synthese erfolgreich zum Ziel zu führen, und noch nicht geklärt ist, wie diese Schritte schematisiert und in eine automatisch auszuführende Methode umgewandelt werden können. Auf einen ungeübten Anwender wirkt das LOPS-Verfahren daher oft recht ziellos, da ihm keine festen Regeln zur Verfügung stehen, nach denen er die nötigen Umformungen auswählen kann. Die Angabe solcher zuverlässiger Regeln und ihre Einbettung in ein formales Konzept ist daher noch ein offenes Forschungsproblem.

5.4.3 Integration in das allgemeine Fundament

Wir wollen das nur andeuten

Satz 5.4.18 (Integration von LOPS – simple Variante)

Es sei ...

Ist (D,R,$\searrow$) *ein wohlfundiertes Rekursionsschema und gilt*

$\forall$x:D.$\forall$g:R. I(x) $\Leftrightarrow$ I_{base}(x,g) $\vee$ I(x$\searrow$g)
$\forall$x:D.$\forall$g,y,y_r:R. DC(x,g) $\wedge\neg$O(x,g) $\wedge$ g$\neq$y $\Rightarrow$
O(x,y) $\Leftrightarrow$ O(x$\searrow$g,y_r) $\wedge$ O_{rec}(x,g,y_r,y)

und sind die folgenden Programme korrekt

FUNCTION f_{DC}(x:D):R WHERE I(x) RETURNS g SUCH THAT DC(x,g) = f_{dc}(x)

FUNCTION f_{BASE}(x,g:D$\times$R):R WHERE I_{base}(x)$\wedge$DC(x,g)$\wedge\neg$O(x,g)
RETURNS y SUCH THAT O(x,y) $\wedge$ g$\neq$y = f_{base}(x,g)

FUNCTION f_{REC}(x,g,y_r:D$\times$R$\times$R):R WHERE I(x$\searrow$g)$\wedge$DC(x,g)$\wedge\neg$O(x,g)$\wedge$O(x$\searrow$g,y_r)
RETURNS y SUCH THAT g$\neq$y $\wedge$ O_{rec}(x,g,y_r,y) = f_{rec}(x,g,y_r)

dann ist das folgende rekursive Programm korrekt

FUNCTION f(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)
= let g=f_{dc}(x) in if O(x,g) then g
else if I_{base}(x,g) then f_{base}(x,g)
else f_{rec}(x,g,f(x$\searrow$g))

Erweiterung: allgemeines GUESS, komplexere Rekursion, mengenwertige Funktionen
$\mapsto$ Einbettung von LOPS durch Theoreme
$\mapsto$ Strategien = Bestimmung der notwendigen Parameter

Versuch der Umsetzung in ein formales System

1. vie EQ trafo des Theorems – vorstellen

Ebenenwechsel und algo-theorie – am simplen LOPS vorführen (hierzu mehr in 5.5)

bergang: das bestreben um die Verbesserung von LOPS führt in die Gleiche Richtung wie Algo-Theorien. Nur so terminiert es.

5.4.4 Synthese durch Tranformationen im Vergleich

Auch wenn die Verwendung von Transformationsregeln und Rewrite-Techniken anstelle von starren Kalkül-Regeln vom Prinzip her wesentlich flexiblere und effizientere Systeme verspricht, wurde in praktischer Hinsicht mit Verfahren zur Synthese durch Transformationen bisher nicht mehr erreicht als mit dem Prinzip "Beweise als Programme". Dies liegt im wesentlichen daran, daß sich viele Transformationen sich auch als Beweistaktiken (sogenannte *Konversionen*, siehe [Jackson, 1993c, Kapitel 8.8]) in beweisbasierte Systeme integrieren lassen. Zum anderen

5.5 Programmentwicklung durch Algorithmenschemata

Praktische Erfahrungen haben gezeigt, daß der Umgang mit Synthesesystemen, die auf der Basis des Prinzips "Beweise als Programme" operieren oder Programme durch Anwendung von äquivalenzerhaltenden Transformationen erzeugen, sehr mühsam ist und daß man über die Synthese einfachster Beispiele[18] nicht hinauskommt, ohne von einem Benutzer massive Eingriffe in den Syntheseprozeß zu verlangen. Der Grund hierfür liegt im wesentlichen darin, daß in beiden Ansätzen *allgemeine Verfahren* die Anwendung elementarer logischer Inferenzen steuern und zur Unterstützung bestenfalls Lemmata über verschiedene Anwendungsbereiche heranziehen. Eine Verarbeitung von wirklichem Programmierwissen, also von Erkenntnissen über Programmstrukturen und ihre Eigenschaften, findet jedoch nicht statt. Aus diesem Grunde ist eine Steuerung derartiger Systeme durch einen "normalen" Benutzer" kaum möglich, da dies tiefe Einsichten in die Strategie und Vertrautheit mit der Denkweise der zugrundeliegenden Kalküle verlangt. Da Programmierer im allgemeinen aber keine Logiker sind, kann von einer echten Unterstützung bei der Entwicklung korrekter Programme noch nicht gesprochen werden.

Ein Konzept, das sich deutlich stärker nach den Bedürfnissen der Praxis richtet, ist der Versuch, Programme durch Verwendung von *Algorithmenschemata* zu synthetisieren. Es basiert auf der Erkenntnis, daß in Lehrbüchern über die Methodik des Programmierens [Knuth, 1968, Knuth, 1972, Knuth, 1975, Gries, 1981] sehr stark auf die Struktur von Algorithmen eingegangen wird, die für eine Lösung bestimmter Probleme besonders geeignet ist, und ebenso auf Methoden, derartige Algorithmen zu entwickeln. Programmiertechniken wie *Generate-and-Test, Divide-and-Conquer, Problemreduktion, Binärsuche, Backtracking, Dynamische Programmierung, Siebe, etc.* sind den meisten Programmierern wohl vertraut und es gibt nur sehr wenige algorithmische Probleme, die nicht mit einer dieser Techniken elegant gelöst werden können. Syntheseverfahren, die für gewöhnliche Programmierer verständlich und kontrollierbar sein sollen, sollten sich daher an diesen Erkenntnissen der Programmiermethodik orientieren und Wissen über die Struktur von Algorithmen verarbeiten. Ein Benutzer sollte die Algorithmenstruktur seiner Wahl festlegen können und das Synthesesystem sollte die Strategien bereitstellen, die Komponenten eines derartigen Algorithmus zu bestimmen und die Bedingungen für seine Korrektheit zu überpüfen. Durch die engeren Vorgaben ist dieses Vorgehen erheblich zielgerichteter als Verfahren, die auf allgemeinen Techniken (wie Induktionsbeweiser oder die LOPS Strategie) aufgebaut sind, und führt daher auch schneller zum Ziel.

Obwohl dieser Gedanke an sich schon relativ alt ist, hat es lange gedauert, ihn in einem erfolgreichen und leistungsfähigen System umzusetzen. Das Hauptproblem hierbei war vor allem, ein tragfähiges theoretisches Fundament zu schaffen, welches die Korrektheit schematisch erzeugter Algorithmen sicherstellt und es ermöglichte, einen Großteil der Beweislast ein für alle Mal im Voraus abzuhandeln und somit aus dem eigentlichen Syntheseprozeß herauszunehmen. Zudem war es nötig, die Struktur verschiedener Klassen von Algorithmen zu analysieren, die Komponenten dieser Algorithmen innerhalb einer einheitliche Beschreibung zu identifizieren, Axiome für die Korrektheit der Algorithmen aufzustellen und die Korrektheit des schematischen Algorithmus nachzuweisen. Schließlich mußten Strategien zur systematischen Erzeugung der fehlenden Komponenten entwickelt werden, die sich – der praktischen Durchführbarkeit wegen – meistens auf vorgefertigte Teilinformationen stützen, ansonsten aber in einem formalen logischen Kalkül eingebettet sind und somit als Beweis- oder Transformationsregel eines sehr hohen Abstraktionsniveaus angesehen werden können.

In den letzten Jahren sind die wichtigsten algorithmischen Strukturen wie *Divide-and-Conquer* [Smith, 1983a, Smith, 1985a, Smith, 1987a], *Lokalsuche* [Lowry, 1988, Lowry, 1987, Lowry, 1989, Lowry, 1991], *Globalsuchalgorithmen* [Smith, 1987b, Smith, 1991a], *Dynamische Programmierung* [Smith, 1991b] und *Problemreduktionsge-*

[18] Nicht selten hört man das Argument, daß Synthesesysteme aus diesem Grunde keinerlei praktische Relevanz besitzen. Man könne ebensogut die wenigen Beispiele direkt in einer Wissensbak abspeichern und dann nach bedarf herausholen.

Im Prinzip ist die Programmentwicklung durch Algorithmenschemata eine konsequente Fortsetzung dieses Gedankens: man speichert tatsächlich vorgefertigte Lösungen in einer Wissensbank ab. Allerdings läßt die Vielzahl der möglichen Algorithmen es nicht zu, diese komplett abzuspeichern. Statt dessen muß man von den Besonderheiten konkreter Algorithmen abstrahieren und Schemata abspeichern, bei denen einige Parameter noch einzusetzen sind. Die Strategie läuft dann darauf hinaus, diese Parameter im konkreten Fall zu bestimmen.

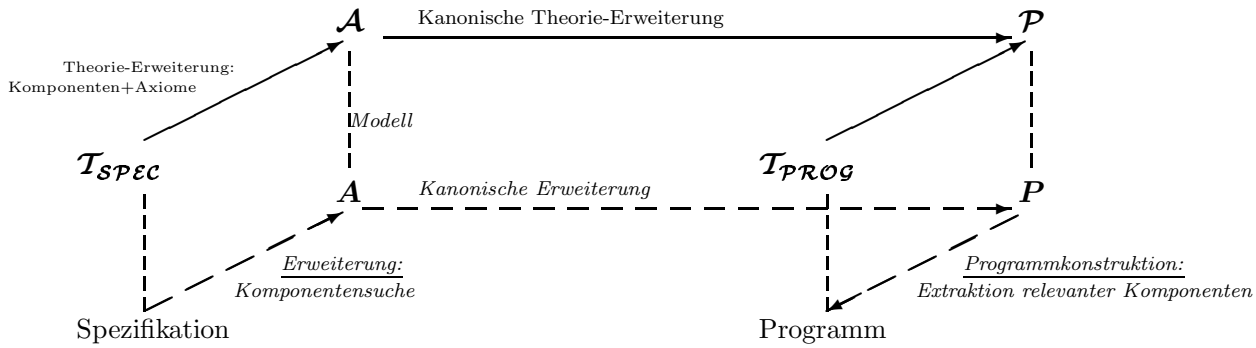


Abbildung 5.2: Synthese durch Erweiterung algebraischer Theorien

neratoren [Smith, 1991c] unabhängig voneinander untersucht worden. Sie haben zu einer Reihe erfolgreicher Algorithmenentwurfsstrategien geführt, die alle in das KIDS System [Smith, 1990, Smith, 1991a] integriert wurden. Die praktischen Erfahrungen mit diesem System zeigen, daß der Einsatz von Algorithmenschemata keineswegs eine Einengung der Möglichkeiten eines Synthesesystems bedeutet sondern vielmehr zu flexiblen und praktisch erfolgreichen Verfahren führen kann.

Wir wollen im folgenden zunächst die theoretischen Grundlagen erklären, die eine formal korrekte Synthese mit Algorithmenschemata ermöglichen. Anschließend werden wir dieses Paradigma am Beispiel von drei Synthesestrategien erläutern und dabei zeigen, wie sich diese Vorgehensweise in ein allgemeines Inferenzsystem wie NuPRL integrieren läßt.

5.5.1 Algorithmenschemata als Algebraische Theorien

Eine formal präzise Grundlage für die Synthese von Algorithmen mit Hilfe von Algorithmenschemata läßt sich am leichtesten im Kontext algebraischer Theorien beschreiben, die für die Beschreibung formaler Spezifikationen [Guttag & Horning, 1978, Sannella & Tarlecki, 1987, Sannella & Tarlecki, 1988] ein gebräuchliches Hilfsmittel sind. Die prinzipielle Vorgehensweise ist schematisch in Abbildung 5.2 dargestellt.

- In der algebraischen Sichtweise werden Spezifikationen als Modelle einer allgemeinen *Problemtheorie* $\mathcal{T}_{\mathcal{SPEC}}$ betrachtet und das Ziel einer Synthese ist, dieses Objekt durch Hinzunahme weiterer Datentypen und Operationen zu einem Modell A einer abstrakten Algorithmentheorie $\mathcal{A}$ zu erweitern.

- Die abstrakte Algorithmentheorie $\mathcal{A}$ wird unabhängig von dem eigentlichen Syntheseprozeß entworfen. Sie beschreibt Namen von Datentypen und Operationen sowie Axiome, die es ermöglichen, innerhalb dieser Theorie Theoreme über die Korrektheit abstrakter Algorithmen zu beweisen, die sich vollständig mit den Bestandteilen der Theorie $\mathcal{A}$ beschreiben lassen.

  Formal bedeutet dies, daß sich die Theorie $\mathcal{A}$ auf kanonische Weise zu einer *Programmtheorie* $\mathcal{P}$ fortsetzen läßt, die neben den Komponenten der Problemtheorie $\mathcal{T}_{\mathcal{SPEC}}$ auch noch einen Programmkörper, seine Axiome und eine Beschreibung durch Komponenten der Theorie $\mathcal{A}$ beinhaltet und eine Erweiterung der Standardprogrammtheorie $\mathcal{T}_{\mathcal{PROG}}$ (mit den Mindestanforderungen an Programme) ist.

- Die Erzeugung eines konkreten Algorithmus benötigt somit nur noch die Instantiierung der abstrakten Namen der Theorie $\mathcal{A}$ durch die konkreten Objekte des Modells A, was gleichbedeutend ist mit der kanonischen Erzeugung eines Modells P von $\mathcal{P}$, aus dem dann die relevante Komponente – nämlich das Programm – extrahiert wird.

Beispiel 5.5.1 (Divide-and-Conquer Algorithmen)

Divide-and-Conquer Algorithmen, die wir in Abschnitt 5.5.4 ausführlicher betrachten werden, lösen ein Problem durch Reduktionen mit Hilfe von Dekompositionsoperationen. Formal lassen sie sich als Programme mit der folgenden Grundstruktur beschreiben:

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

hierbei sind `Decompose, G, Compose, Directly-solve` und `primitive` Parameter, die im Verlaufe einer Synthese noch zu bestimmen sind.

Die Theorie $\mathcal{A}_{D\&C}$ der Divide-and-Conquer Algorithmen enthält neben den Namen `D, R` für Ein- und Ausgabebereich und den Namen `I, O` für Ein- und Ausgabebedingungen also noch die Namen `Decompose, G, Compose, Directly-solve` und `primitive` als zusätzliche Operationen. Hinzu kommen zwei zusätzliche Namen für Datentypen, die für eine korrekte Typisierung erforderlich sind und natürlich die Axiome, welche die Korrektheit des obigen Algorithmus garantieren.

Synthese bedeutet somit, aufgrund der konkreten Werte für `D, R, I` und `O` konkrete Operationen zu finden, welche anstelle von `Decompose, G, Compose, Directly-solve` und `primitive` eingesetzt werden können und dann das obige Schema entsprechend zu belegen.

Das allgemeine Konzept operiert also auf zwei Ebenen. Auf der Ebene der *algorithmischen Theorien* werden die grundsätzlichen Zusammenhänge formal beschrieben[19] und überprüft. Auf der Ebene der konkreten Probleme wird dann die eigentliche Synthese ausgeführt, wobei der schwierigste Schritt – die Erzeugung des Algorithmus – nun ganz schematisch und *ohne weitere Inferenzen* geschehen kann. Spezialisierte Synthesestrategien können sich daher auf die Erzeugung von Modellen der entsprechenden algorithmischen Theorie konzentrieren. Um dies zu präzisieren, wollen wir einige Definitionen aus der Theorie der algebraischen Spezifikationen aufgreifen.

Definition 5.5.2 (Formale Theorien und Modelle)

1. *Eine Signatur Σ ist ein Paar (S, Ω), wobei S eine Familie von Namen für Sorten (Datentypen) und Ω eine Familie von typisierten Operationsnamen ist.*

2. *Eine formale Theorie $\mathcal{T}$ besteht aus einer Signatur Σ, welche das Vokabular der Theorie beschreibt und einer Menge Ax von Axiomen für die genannten Sorten und Operationen.*

3. *Eine Theorie $\mathcal{T}_1$ erweitert eine Theorie $\mathcal{T}_2$, wenn alle Sortennamen, Operationsnamen und Axiome von $\mathcal{T}_2$ – bis auf konsistente Umbenennungen – auch in $\mathcal{T}_1$ existieren.*

4. *Eine Struktur T für $\mathcal{T}$ besteht aus einer Familie nichtleerer Datentypen und einer Familie von Operationen, welche die in $\mathcal{T}$ genannten Typisierungen respektieren.*

5. *Eine Struktur T_1 für $\mathcal{T}_1$ erweitert eine Struktur T_2 für $\mathcal{T}_1$, wenn alle Datentypen und Operationen von T_2 auch in T_1 existieren und dort die gleichen (in $\mathcal{T}_2$ genannten) Typbedingungen erfüllen.*

6. *Ein Modell T $\mathcal{T}$ ist eine Struktur für $\mathcal{T}$, deren Datentypen und Operationen alle Axiome aus $\mathcal{T}$ erfüllen.*

Formale (algebraische) Theorien sind somit nichts anderes als formale Theorien über Objekte, von denen wir nicht mehr kennen als ihren Namen, ihre Typisierung und ihre Axiome. Strukturen sind Belegungen dieser Namen durch konkrete typkorrekte Objekte und Modelle sind Belegungen der Namen durch konkrete

[19] Im Prinzip ist die abstrakte algorithmischen Theorie nichts anderes als eine geringfügig formalere Beschreibung der zentralen Komponenten eines Algorithmus einer bestimmten Klasse. Wir haben immer dann, wenn wir allgemein über Spezifikationen, Programme und Strategien gesprochen haben, abstrakte Namen `D, R` für Ein- und Ausgabebereich und abstrakte Namen `I, O` für Ein- und Ausgabebedingungen verwendet und implizit vorausgesetzt, daß es sich hierbei um *Platzhalter* für die tatsächlichen Bereiche und Bedingungen handelt. Agorithmischen Theorien sind somit nichts anderes als formale Theorien über derartige Platzhalter, von denen wir nicht mehr kennen als ihren Namen und ihre Axiome.

Objekte, die neben der Typkorrektheit auch noch die Axiome erfüllen. Wir wollen diese Begriffe anhand zweier formaler Theorien illustrieren, welche das Konzept der Spezifikationen und das der Programme im Kontext algebraischer Theorien repräsentieren.

Definition 5.5.3 (Problem- und Programmtheorien)

1. $\underline{\mathcal{T}_{\mathcal{SPEC}}}$ = ({D,R}, {I:D→𝔹, O:D×R→𝔹}, ∅) *ist die algebraische Theorie der Spezifikationen.*
2. $\underline{\mathcal{T}_{\mathcal{PROG}}}$ = ({D,R}, {I: D→𝔹, O: D×R→𝔹, body: D⇀R}, { ∀x:D. I(x) ⇒ O(x,body(x)) }) *ist die algebraische Theorie der Programme*
3. *Eine Theorie $\mathcal{P}$ heißt Problemtheorie, wenn sie $\mathcal{T}_{\mathcal{SPEC}}$ erweitert.*
4. *Eine Theorie $\mathcal{P}$ heißt Programmtheorie, wenn sie $\mathcal{T}_{\mathcal{PROG}}$ erweitert.*

Eine Problemtheorie[20] ist also eine beliebige algebraische Theorie, deren Modelle unter anderem eine Programmspezifikation enthalten, und eine Programmtheorie ist eine algebraische Theorie, deren Modelle unter anderem ein korrektes Programm enthalten. Ein Modell einer Programmtheorie, welches ein Modell P einer Problemtheorie erweitert, enthält also eine Lösung der in P genannten Spezifikation.

Beispiel 5.5.4

Strukturen und Modelle für Problem- und Programmtheorien sind nichts anderes als eine etwas umsortierte Schreibweise für Spezifikationen und Programme im Sinne von Definition 5.2.1. Die Repräsentation einer Spezifikation des Sortierproblems

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true
  RETURNS S  SUCH THAT rearranges(L,S) ∧ ordered(S)
```

wird somit dargestellt[21] als folgendes Modell von $\mathcal{T}_{\mathcal{SPEC}}$

{Seq(ℤ), Seq(ℤ)}, {λL. true, λL,S. rearranges(L,S) ∧ ordered(S)}

Die Repräsentation eines Selectionsort-Programms stellt eine Erweiterung dieses Modells durch Hinzunahme einer weiteren Komponente dar:

```
{Seq(ℤ), Seq(ℤ)}, {λL. true, λL,S. rearranges(L,S) ∧ ordered(S),
                   letrec sort(L) = if L=[] then [] else let g=min(L) in g.sort(L-g)}
```

Eine Erweiterung wäre auch

{Seq(ℤ), Seq(ℤ)}, {λL. true, λL,S. rearranges(L,S) ∧ ordered(S), letrec sort(L) = []}

Diese Erweiterung wäre zwar immer noch eine Struktur für $\mathcal{T}_{\mathcal{PROG}}$, da Typkorrektheit nach wie vor erfüllt ist. Ein Modell aber ist sie nicht, da das Axiom von $\mathcal{T}_{\mathcal{PROG}}$ verletzt ist.

Die in Beispiel 5.5.4 charakterisierte Umwandlung von Spezifikationen und Programmen in Modelle algebraischer Theorien wird durch das folgende Lemma gerechtfertigt.

Lemma 5.5.5

Es sei spec=(D, R, I, O) *eine beliebige Spezifikation und* $body: D \rightharpoonup R$. *Dann gilt*

1. *Das Programm* `FUNCTION F(`x`:`D`):`R `WHERE` $I(x)$ `RETURNS` y `SUCH THAT` $O(x,y) = body(x)$ *ist genau dann korrekt, wenn* $\underline{T_{spec,\ body}} = (\{D, R\}, \{I, O, body\})$ *ein Modell von* $\mathcal{T}_{PROG}$ *ist.*
2. *Die Spezifikation* `FUNCTION F(`x`:`D`):`R `WHERE` $I(x)$ `RETURNS` y `SUCH THAT` $O(x,y)$ *ist genau dann erfüllbar, wenn es ein Modell P von $\mathcal{T}_{PROG}$ gibt, welches* $\underline{T_{spec}} = (\{D, R\}, I, O\})$ *erweitert.*

[20] Man beachte, daß die Formulierung der Theorie $\mathcal{T}_{\mathcal{SPEC}}$ eine große Verwandtschaft zu der Formalisierung des Konzeptes "Spezifikation" (siehe Übung 7) durch Konstrukte der Typentheorie besitzt. Diese Verwandtschaft zwischen algebraischen Theorien und den noch formaleren typentheoretischen Ausdrücken (die keine informalen Beschreibungen zulassen) weist einen Weg für eine Integration algorithmischer Theorien und der zugehörigen Synthesestrategien in einen allgemeines formales Inferenzsystem.

[21] Wir verwenden die Mengenschreibweise, um anzudeuten, daß es auf die Reihenfolge der Objekte einer Familie eigentlich nicht ankommt. Allerdings muß der Bezug zwischen einer Komponente und seinem Namen in der abstrakten Theorie erkennbar bleiben, so daß man bei einer Implementierung besser auf Listen zurückgreift.

Aufgrund von Lemma 5.5.5 können wir Programmsynthese tatsächlich als Erweiterung algebraischer Strukturen beschreiben. Das Konzept der Algorithmentheorien als Hilfsmittel für eine systematische Erweiterung von Spezifikationen läßt sich mit den soeben eingeführten Begriffen nun relativ leicht beschreiben: eine Algorithmentheorie ist eine Problemtheorie, die eine kanonische Erweiterung zu einer Programmtheorie besitzt.

Definition 5.5.6 (Algorithmentheorien)

Eine Problemtheorie $\mathcal{A}$ heißt Algorithmentheorie, wenn es ein Programmschema `BODY` *gibt mit der Eigenschaft*

- `BODY` *weist jeder Struktur A für $\mathcal{A}$ eine Funktion aus* $\mathtt{D}_A \not\to \mathtt{R}_A$ *zu,*
- *Für jedes Modell A von $\mathcal{A}$ ist das folgende Programm korrekt.*[22]
  `FUNCTION F(x:`$\mathtt{D}_A$`):`$\mathtt{R}_A$ `WHERE` $\mathtt{I}_A$`(x) RETURNS y SUCH THAT` $\mathtt{O}_A$`(x,y) = BODY(`A`)(x)`

  Dabei sei $\underline{\mathtt{D}_A}$ *(bzw.* $\underline{\mathtt{R}_A}$, $\underline{\mathtt{I}_A}$, $\underline{\mathtt{O}_A}$*) diejenige Komponente der Struktur A, welche in der Theorie $\mathcal{A}$ dem Namen* `D` *(bzw.* `R, I, O`*) entspricht.*

Algorithmenschemata sind also nichts anderes als Funktionen, die in einem schematischen Algorithmus jeden Namen durch ein konkretes Objekt eines Modells A ersetzen und so eine Standardlösung der Spezifikation $\mathtt{spec}_A$ `=` $(\mathtt{D}_A, \mathtt{R}_A, \mathtt{I}_A, \mathtt{O}_A)$ generieren. Eine Algorithmentheorie ist nichts anderes als eine Beschreibung von Komponenten und Axiomen, welche die Existenz eines solchen Algorithmenschemas garantieren.

Beispiel 5.5.7 (Formale Theorie der Divide-and-Conquer Algorithmen)

Die formale Theorie der Divide-and-Conquer Algorithmen (vergleiche Beispiel 5.5.1) enthält neben den Bestandteilen des eigentlichen Algorithmus noch ene Reihe zusätzlicher Komponenten, die für eine Spezifikation der Bestandteile des Algorithmus und für einen Nachweis der Terminierung benötigt werden. Alle Komponenten und Axiome ergeben zusammengefaßt die formale Algorithmentheorie $\mathcal{T}_{D\&C} = (S_{D\&C}, \Omega_{D\&C}, Ax_{D\&C})$ mit den folgenden Bestandteilen:

$\mathtt{S}_{D\&C} \equiv \{\mathtt{D}, \mathtt{R}, \mathtt{D'}, \mathtt{R'}\}$

$\Omega_{D\&C} \equiv \{\mathtt{I}: \mathtt{D} \to \mathbb{B},\ \mathtt{O}: \mathtt{D}\times\mathtt{R} \to \mathbb{B},\ \mathtt{O}_D: \mathtt{D}\times\mathtt{D'}\times\mathtt{D} \to \mathbb{B},\ \mathtt{I'}: \mathtt{D'} \to \mathbb{B},\ \mathtt{O'}: \mathtt{D'}\times\mathtt{R'} \to \mathbb{B},\ \mathtt{O}_C: \mathtt{R'}\times\mathtt{R}\times\mathtt{R} \to \mathbb{B},$
$\succ: \mathtt{D}\times\mathtt{D} \to \mathbb{B},\ \mathtt{Decompose}: \mathtt{D} \to \mathtt{D'}\times\mathtt{D},\ \mathtt{G}: \mathtt{D'} \to \mathtt{R'},\ \mathtt{Compose}: \mathtt{R'}\times\mathtt{R} \to \mathtt{R},$
$\mathtt{Directly\text{-}solve}: \mathtt{D} \to \mathtt{R},\ \mathtt{primitive}: \mathtt{D} \to \mathbb{B}\}$

$Ax_{D\&C} \equiv \{$
`FUNCTION` $\mathtt{F}_p$`(x:D):R WHERE I(x)` $\wedge$ `primitive(x) RETURNS y SUCH THAT O(x,y)`
`= Directly-solve(x)` korrekt,
$\mathtt{O}_D$`(x,y',y)` $\wedge$ `O(y,z)` $\wedge$ `O'(y',z')` $\wedge$ $\mathtt{O}_C$`(z,z',t)` $\Rightarrow$ `O(x,t)`,
`FUNCTION` $\mathtt{F}_d$`(x:D):D'×D WHERE I(x)` $\wedge$ $\neg$`primitive(x)`
`RETURNS y',y SUCH THAT I'(y')` $\wedge$ `I(y)` $\wedge$ `x` $\succ$ `y` $\wedge$ $\mathtt{O}_D$`(x,y',y) = Decompose(x)` korrekt,
`FUNCTION` $\mathtt{F}_c$`(z,z':R×R'):R RETURNS t SUCH THAT` $\mathtt{O}_C$`(z,z',t) = Compose(z,z')` korrekt,
`FUNCTION` $\mathtt{F}_G$`(y':D'):R' WHERE I'(y') RETURNS z' SUCH THAT O'(y',z') = G(y')` korrekt,
$\succ$ `ist wohlfundierte Ordnung auf D`
$\}$

Das abstrakte Programmschema für diese Algorithmentheorie ist eine Abbildung von einer Struktur $A =$ `({D, R, D', R'}, {I, O,` $\mathtt{O}_D$`, I', O',` $\mathtt{O}_C$`,` $\succ$`, Decompose, G, Compose, Directly-solve, primitive})` in den Algorithmus

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

`BODY(`A`)` ist korrekt, wenn A alle obengenannten Axiome erfüllt, also ein Modell von $\mathcal{T}_{D\&C}$ ist.

[22] Gemäß Lemma 5.5.5 könnten wir als algebraisch formulierte Bedingung auch schreiben: "`({`$\mathtt{D}_A, \mathtt{R}_A$`}, {`$\mathtt{I}_A, \mathtt{O}_A$`, BODY(`$A$`)})` ist ein Modell für $\mathcal{T}_{PROG}$". Eine schematische Beschreibung der Funktion `BODY` durch Komponenten von $\mathcal{A}$ muß sich daher ausschließlich mit den Axiomen aus $\mathcal{A}$ als korrekt im Sinne des Axiomes für $\mathcal{T}_{PROG}$ nachweisen lassen: die Korrektheit ist also ein Theorem der Theorie $\mathcal{A}$.

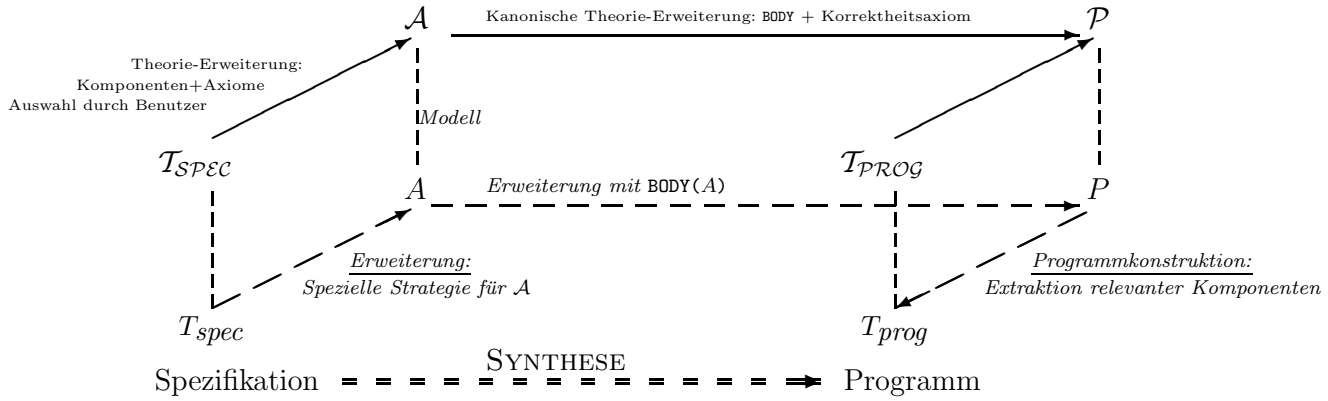


Abbildung 5.3: Synthese mit Algorithmenschemata: Generelle Methode

Mit den bisher gegebenen Definitionen ist es nicht sehr schwierig eine allgemeine Strategie zur Synthese von Programmen mit Hilfe von Algorithmenschemata zu berschreiben und formal zu rechtfertigen. Ihre Grundlage ist der folgende Satz.

Satz 5.5.8

Eine Spezifikation spec=(D, R, I, O) ist genau dann erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell A für $\mathcal{A}$ gibt, welches die Struktur T_{spec} erweitert.

Beweis: Es sei $\mathcal{A}$ eine Algorithmentheorie und A ein Modell für $\mathcal{A}$, welches die Struktur $T_{spec} = (\{D, R\}, \{I, O\})$ erweitert. Gemäß Definition 5.5.2 ist dann *spec*=$\mathtt{spec}_A$, also $\mathtt{D}_A$=D, $\mathtt{R}_A$=R, $\mathtt{I}_A$=I und $\mathtt{O}_A$=O und für das zu $\mathcal{A}$ gehörige abstrakte Programmschema BODY ist dann das Programm

FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y) = BODY(A)(x)

korrekt und somit ist die Spezifikation *spec* erfüllbar. □

Satz 5.5.8 rechtfertigt die folgende generelle Methode zur Synthese von Programmen, die in Abbildung 5.3 (eine Präzisierung von Abbildung 5.2) schematisch dargestellt ist.

Strategie 5.5.9 (Synthese mit Algorithmenschemata: Generelle Methode)

Gegeben sei eine Problemspezifikation FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)

1. *Wähle eine Algorithmentheorie $\mathcal{A}$ aus der Wissensbank*
2. *Erweitere $T_{spec} = (\{D, R\}, \{I, O\})$ durch Hinzufügen von Komponenten und Überprüfung von Axiomen zu einem Modell A von $\mathcal{A}$*
3. *Bestimme* BODY(A) *und liefere als Antwort das Programm*

   FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y) = BODY(A)(x)

   Dieses Programm ist garantiert korrekt.

Der erste Schritt dieser Strategie ist im wesentlichen eine Entwurfsentscheidung, die einem Benutzer des Systems überlassen werden sollte. Der zweite Schritt dagegen sollte dagegen durch eine Strategie weitestgehend automatisch durchgeführt werden.

Natürlich ist die soeben beschriebene Methode noch sehr allgemein, da sie ja noch nicht sagt, wie denn nun die zusätzlichen Komponenten des Modells A von $\mathcal{A}$ bestimmt werden sollen. Grundsätzlich sollten die entsprechenden Teilstrategien auch sehr stark auf die gewählten Algorithmentheorien zugeschnitten werden, um effizient genug zu sein. Dennoch hat es sich als sinnvoll herausgestellt, nicht nur bei der Auswahl der Algorithmentheorien als solche auf vorgefertigtes Wissen zurückzugreifen sondern dies auch bei der Bestimmung eines konkreten Modells zu tun, also zu jeder Algorithmentheorie eine Reihe von möglichst allgemeinen Modellen in einer Wissensbank abzulegen und diese dann auf ein gegebenes Problem zu *spezialisieren*. Auf

diese Art kann man sich eventuell die Überprüfung komplizierter Axiome – wie etwa der Wohlfundiertheit von $\succ$ bei Divide-and-Conquer Algorithmen – ersparen.

Es gibt eine Reihe von allgemeinen Techniken, bereits bekannte Modelle auf ein neues Problem anzuwenden, durch die Strategie 5.5.9 ein wenig verfeinert werden kann.

- Ein allgemeines Modell kann durch Verwendung von *Typvariablen* in ein generisches Modell für eine Klasse von Algorithmen verwandelt werden.
- Durch *Reduktion* kann ein Problem auf ein spezielleres abgebildet werden, für das bereits eine Lösung vorliegt ("Operator Match").
- Bei mengenwertigen Problemstellungen (Suche nach *allen* Lösungen einer Ausgabebedingung) kann ein allgemeines Modell auf eine neue Problemstellung *spezialisiert* werden.

Da die Vorgehensweise im ersten Fall naheliegend ist, beschränken wir uns im folgenden auf Reduktion und Spezialisierung.

Definition 5.5.10 (Reduktion und Spezialisierung)

Es seien spec=(D,R,I,O) und spec'=(D',R',I',O') zwei beliebige Spezifikationen

1. *spec ist reduzierbar auf spec' (im Zeichen spec$\trianglelefteq$spec'), wenn gilt*

$$\forall x{:}D.\ I(x) \ \Rightarrow\ \exists x'{:}D'.\ I'(x') \wedge \forall y'{:}R'.\ O'(x',y') \Rightarrow \exists y{:}R.O(x,y)$$

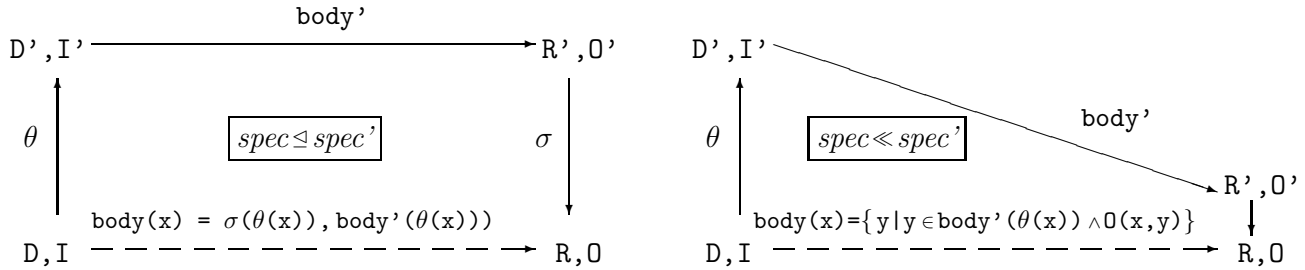
2. *spec spezialisiert spec' (im Zeichen spec$\ll$spec'), wenn R' eine Obermenge von R ist und gilt*

$$\forall x{:}D.\ I(x) \ \Rightarrow\ \exists x'{:}D'.\ I'(x') \wedge \forall y{:}R.\ O(x,y) \Rightarrow O'(x',y)$$

Man sagt auch, daß spec' die Spezifikation spec generalisiert

Es ist also *spec* reduzierbar auf *spec'*, wenn die Eingabebedingung von *spec* (nach Transformation) stärker ist als die von *spec'* und die Ausgabebedingung schwächer ist. *spec* spezialisiert *spec'*, wenn sowohl die Ein- als auch die Ausgabebedingung stärker ist, wobei keine Transformation der Ausgaben stattfindet.

Beide Definitionen enthalten eine typische Anwendung des Prinzip "Beweise als Programme" in seiner elementarsten Form. Ein formaler Nachweis der Reduzierbarkeit induziert zwei Transformationsfunktionen $\theta{:}D \to D'$ und $\sigma{:}D' \times R' \to R$, die zur Transformation eines bekannten Algorithmus gemäß dem untenstehen Diagramm verwendet werden können. Ein Nachweis der Spezialisierung induziert eine Transformation $\theta{:}D \to D'$, die zusammen mit der Ausgabebedingung O einen bekannten mengenwertigen Algorithmus in einen Lösungsalgorithmus für die Ausgangsspezifikation verwandeln kann.



Die Kombination dieser Erkenntnisse mit dem in Satz 5.5.8 allgemeinen Verfahren führt zu zwei Verfeinerungen des allgemeinen Syntheseverfahrens, deren Basis die folgenden zwei Sätze sind.

Satz 5.5.11 (Operator Match)

Eine Spezifikation spec=(D,R,I,O) ist genau dann erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell A für $\mathcal{A}$ gibt, mit der Eigenschaft spec$\trianglelefteq \mathtt{spec}_A$.

Satz 5.5.11 erweitert Satz 5.5.8 insofern, daß die Bedingung *spec*=$\texttt{spec}_A$ zu *spec*$\trianglelefteq\texttt{spec}_A$ abgeschwächt wurde, daß also anstelle von Gleichheit zwischen Spezifikation und dem Spezifikationsanteil des Modells nur noch Reduzierbarkeit gefordert wird. Entsprechend kann auch die Synthesestrategie verfeinert werden.

Strategie 5.5.12 (Synthese mit Algorithmenschemata und Reduktion)

Gegeben sei eine Problemspezifikation FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)

1. *Wähle eine Algorithmentheorie $\mathcal{A}$ aus der Wissensbank*
2. *Wähle ein allgemeines Modell A von $\mathcal{A}$ aus der Wissensbank*
3. *Beweise* $(D,R,I,O)\trianglelefteq\texttt{spec}_A$ *und extrahiere aus dem Beweis zwei Transformationsfunktionen σ und θ*
4. *Spezialisiere* BODY(A) *und liefere als Antwort das Programm*
   FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y) = σ(θ(x),BODY(A)(θ(x)))
   Dieses Programm ist garantiert korrekt.

Dieses Verfahren läßt sich immer dann anwenden, wenn man eine Vorstellung davon hat, auf welche Art man das zu lösende Problem verallgemeinern möchte. Für die Auswahl von A ist daher eine gewisse Benutzersteuerung notwendig. So könnte man zum Beispiel Berechnungen über natürlichen Zahlen auf Berechnungen über Listen reduzieren, da die Zahl n isomorph zur Liste [1..n] ist, und auf diese Art einige Reduktionsstrategien für Listen auf Zahlen übertragen. Weitere Strategien zur Auswahl von A hängen im allgemeinen jedoch sehr von der konkreten Algorithmentheorie ab.

Für mengenwertige Problemstellungen läßt sich die Technik der Generalisierung einsetzen. Der folgende Satz sagt im wesentlichen, wie dies geschehen kann.

Satz 5.5.13 (Synthese durch Generalisierung)

Eine Spezifikation FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS $\{$z | O(x,z)$\}$ *ist genau dann erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell A für $\mathcal{A}$ gibt, das (D,R,I,O) generalisiert.*

Man beachte, daß die Generalisierung sich auf die Ausgabebedingung O bezieht und nicht etwa auf $\{$z | O(x,z)$\}$. Da bei der Generalisierung die der Ausgabebereich des zu wählenden Modells bis auf Obermengenbildung bereits festliegt, läßt sich die allgemeine Strategie 5.5.9 wie folgt verfeinern.

Strategie 5.5.14 (Synthese mit Algorithmenschemata und Spezialisierung)

Gegeben sei eine Problemspezifikation FUNCTION F(x:D):R WHERE I(x) RETURNS $\{$z | O(x,z)$\}$

1. *Wähle eine Algorithmentheorie $\mathcal{A}$ aus der Wissensbank.*
2. *Wähle ein allgemeines Modell A von $\mathcal{A}$ aus der Wissensbank mit der Eigenschaft $R\subseteq R_A$.*
3. *Beweise* $(D,R,I,O)\trianglelefteq\texttt{spec}_A$ *und extrahiere aus dem Beweis eine Transformationsfunktion θ*
4. *Spezialisiere* BODY(A) *und liefere als Antwort das Programm*
   FUNCTION F(x:D):R WHERE I(x) RETURNS $\{$z | O(x,z)$\}$ = $\{$y | y$\in$BODY(A)(θ(x)) $\wedge$ O(x,y)$\}$
   Dieses Programm ist garantiert korrekt.

Der Anwendungsbereich dieser Grundstrategie, die sich je nach Algorithmentheorie noch ein wenig verfeinern läßt, ist erstaunlich groß. Sie hat besonders im Zusammenhang mit Globalsuch-Algorithmen (siehe Abschnitt 5.5.2) zu einer sehr effizienten Synthesestrategie geführt, in der außer dem Nachweis der Generalisierung und einer ähnlich leichten Überprüfung von sogenannten Filtern praktisch keine logischen Schlußfolgerungen – insbesondere also keine Induktionsbeweise – anfallen.

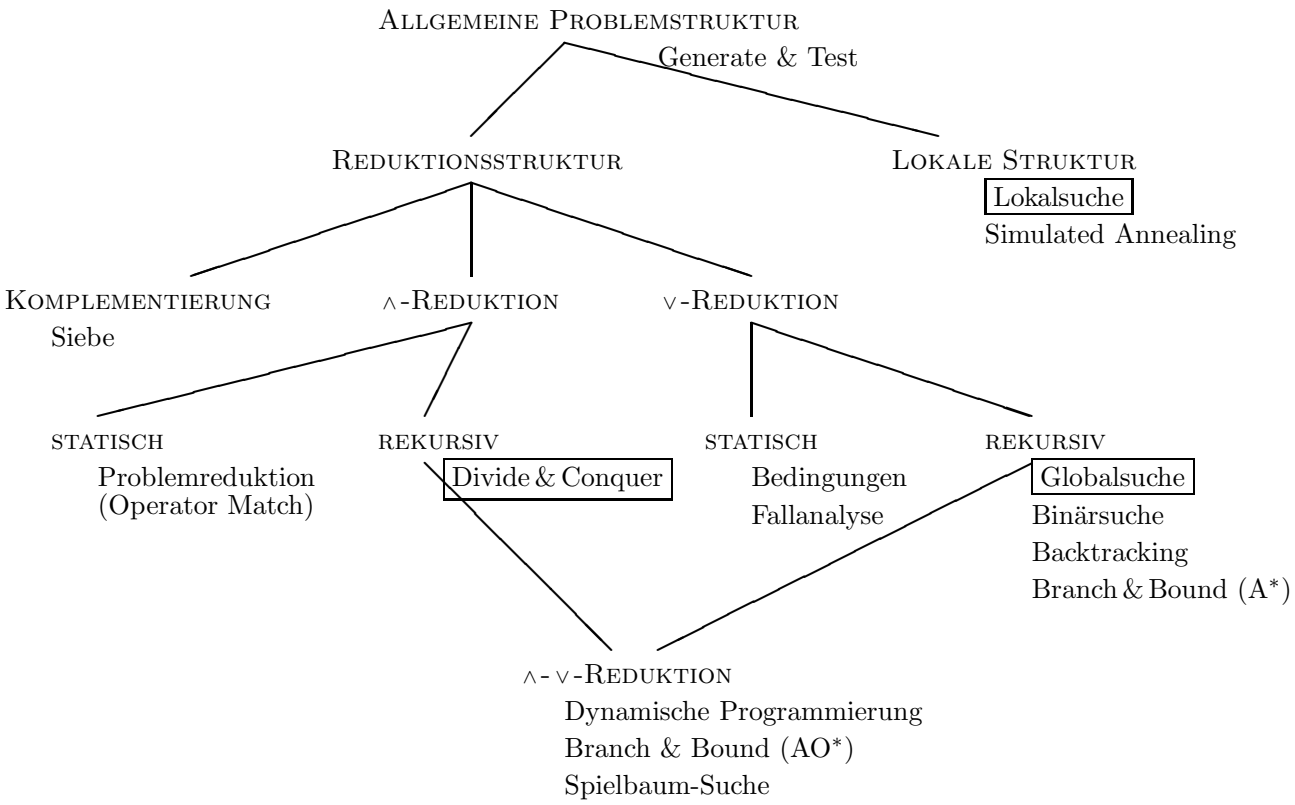


Abbildung 5.4: Hierarchie algorithmischer Theorien

Im Laufe der letzten Jahre ist eine Reihe algorithmischer Strukturen ausführlich untersucht und in der Form von Algorithmentheorien formalisiert worden. Dabei wurden auch Zusammenhänge zwischen den einzelnen Formalisierungen entdeckt, die sich mehr oder weniger direkt aus der hierachischen Anordnung der einzelnen Algorithmenstrukturen (siehe Abbildung 5.4) ergeben. Es erscheint daher sinnvoll, nicht nur einzelne Algorithmentheorien zu betrachten, sondern auch Mechanismen zur Verfeinerung von Theorien und zur Generierung von Entwurfsstrategien zu entwickeln. Derartige Mechanismen werden zur Zeit erforscht [Smith, 1992, Smith, 1993] und sollen in der Zukunft zu einer erheblichen Flexibilitätssteigerung von Synthesesystemen führen.

Wir wollen im folgenden nun einige konkrete Algorithmentheorien – Globalsuchalgorithmen, Divide-and-Conquer Algorithmen und Lokalsuchalgorithemn – im Detail betrachten und hieran die Leistungsfähigkeit dieses Paradigmas demonstrieren.

5.5.2 Globalsuch-Algorithmen

Problemlösung durch Aufzählen einer Menge von Lösungskandidaten ist eine weitverbreitete Programmiertechnik. *Globalsuche* ist eine Aufzählungsmethode, die als Verallgemeinerung von Binärsuche, Backtracking, Branch-and-Bound, Constraint Satisfaction, heuristischer Suche und ähnlichen Suchverfahren angesehen werden kann, und – zumindest aus theoretischer Sicht – zur Lösung jeder beliebigen Problemstellung eingesetzt werden kann.

Abstrahiert man von den Besonderheiten individueller Präsentationen so stellt sich heraus, daß die Grundidee, die all diese Algorithmen gemeinsam haben, eine Manipulation gewisser Mengen von Lösungskandidaten ist: man beginnt mit einer *Initialmenge*, die alle möglichen Lösungen enthält und *zerteilt* diese in kleinere Kandidatenmengen, aus denen man schließlich konkrete Lösungen *extrahieren* und überprüfen kann. Überflüssige Kandidatenmengen können durch *Filter* eliminiert werden. Der gesamte Prozeß wird wiederholt, bis keine weitere Aufspaltung der Kandidatenmengen mehr möglich ist.

Nur in den seltensten Fällen werden die Kandidatenmengen explizit dargestellt, da zu Beginn üblicherweise unendliche Mengen von Lösungskandidaten betrachtet werden müssen. Aus diesem Grunde werden Kandidatenmengen in Globalsuchalgorithmen im allgemeinen durch sogenannte *Deskriptoren* beschrieben – also durch endliche Objekte, die ein charakterisierendes Merkmal dieser Mengen repräsentieren – und die Zugehörigkeitsrelation zwischen Ausgabewerten und Kandidatenmengen durch ein allgemeineres *Erfüllbarkeitsprädikat* zwischen Ausgabewerten und Deskriptoren ersetzt.

Formalisiert man diese Vorgehensweise in einem einheitlichen Algorithmenschema, so erweist es sich als sinnvoll, Globalsuchalgorithmen durch mengenwertige Programme zu beschreiben, die versuchen *alle* möglichen Lösungen eines Problems zu berechnen. Auf diese Art kann tatsächlich jede der obengenannten Programmiertechniken als Spezialfall von Globalsuchalgorithmen dargestellt werden. Die allgemeine Grundstruktur von Globalsuchalgorithmen ist daher die folgende.

$$\begin{array}{l}\texttt{FUNCTION F(x:}\underline{\texttt{D}}\texttt{):Set(}\underline{\texttt{R}}\texttt{) WHERE }\underline{\texttt{I}}\texttt{(x) RETURNS }\{\,\texttt{z}\mid\underline{\texttt{O}}\texttt{(x,z)}\,\}\\ =\ \texttt{if }\underline{\Phi}\texttt{(x,}\underline{\texttt{s}_0}\texttt{(x)) then F}_{gs}\texttt{(x,}\underline{\texttt{s}_0}\texttt{(x)) else }\emptyset\\ \\ \texttt{FUNCTION F}_{gs}\texttt{(x,s:}\underline{\texttt{D}}\times\underline{\texttt{S}}\texttt{):Set(}\underline{\texttt{R}}\texttt{) WHERE }\underline{\texttt{I}}\texttt{(x)}\wedge\underline{\texttt{J}}\texttt{(x,s)}\wedge\underline{\Phi}\texttt{(x,s)}\\ \quad\texttt{RETURNS }\{\,\texttt{z}\mid\underline{\texttt{O}}\texttt{(x,z)}\wedge\underline{\texttt{sat}}\texttt{(z,s)}\,\}\\ =\ \{\,\texttt{z}\mid\texttt{z}\in\underline{\texttt{ext}}\texttt{(s)}\wedge\underline{\texttt{O}}\texttt{(x,z)}\,\}\ \cup\ \bigcup\{\,\texttt{F}_{gs}\texttt{(x,t)}\mid\texttt{t}\in\underline{\texttt{split}}\texttt{(x,s)}\wedge\underline{\Phi}\texttt{(x,t)}\,\}\end{array}$$

Hierbei treten neben den Bestandteilen der eigentlichen Spezifikation die folgenden Komponenten auf.

- Ein Raum $\boxed{\texttt{S}}$ von *Deskriptoren* für Mengen von Lösungskandidaten.
- Ein *Zugehörigkeitsprädikat* $\boxed{\texttt{sat}}$ auf $\texttt{R}\times\texttt{S}$, welches angibt, ob ein gegebener Wert $\texttt{z}\in\texttt{R}$ zu der durch den Deskriptor $\texttt{s}\in\texttt{S}$ beschriebenen Kandidatenmenge gehört.
- Ein Prädikat $\boxed{\texttt{J}}$ auf $\texttt{D}\times\texttt{S}$, welches angibt, ob ein gegebener Deskriptor $\texttt{s}\in\texttt{S}$ für eine Eingabe $\texttt{x}\in\texttt{D}$ überhaupt *sinnvoll*[23] ist.
- Ein *Initialdeskriptor* $\boxed{\texttt{s}_0\texttt{(x)}}\in\texttt{S}$, der den Ausgangspunkt des Suchverfahrens beschreibt und zwangsläufig alle alle korrekten Lösungen "enthalten" muß.
- Eine mengenwertige Funktion $\boxed{\texttt{ext}}\texttt{:S}\to\texttt{Set(R)}$, welche die *direkte Extraktion* konkreter Lösungskandidaten aus einem Deskriptor durchführt.
- Eine mengenwertige Funktion $\boxed{\texttt{split}}\texttt{: D}\times\texttt{S}\to\texttt{Set(S)}$, welche die (rekursive) Aufspaltung von Deskriptoren in Mengen von Deskriptoren für "kleinere" Kandidatenmengen beschreibt.
- Ein Filter $\boxed{\Phi}\texttt{: D}\times\texttt{S}\to\mathbb{B}$, der dazu genutzt werden kann, unnötige Deskriptoren von der weiteren Betrachtung zu eliminieren. Diese dienen im wesentlichen der Effizienzsteigerung und können auch in die Funktion `split` integriert sein.

Ein Globalsuchalgorithmus `F` ruft also bei Eingabe eines Wertes `x` eine Hilfsfunktion $\texttt{F}_{gs}$ mit `x` und dem Initialdeskriptor $\texttt{s}_0\texttt{(x)}$ auf, sofern diese Werte den Filter Φ passieren. $\texttt{F}_{gs}$ wiederum berechnet bei Eingabe von `x` und einem Deskriptor die Menge aller zulässigen Lösungen, die direkt aus `s` extrahiert werden können, und vereinigt diese mit der Menge aller Lösungen, die rekursiv durch Analyse aller Deskriptoren `t` bestimmt werden können, welche sich durch Aufspalten mit `split` und Filterung mit Φ ergeben.

Wir haben hier zur Beschreibung des Algorithmus eine funktionale Darstellung in der von uns angegebenen mathematischen Programmiersprache gewählt, da diese die kürzeste und eleganteste Darstellung des Verfahrens ermöglicht. Wir hätten ebenso aber auch ein Programmschema in einer beliebigen anderen Programmiersprache (sogar in einer parallelverarbeitenden Sprache) angeben können, denn hiervon hängt die Synthese nicht im geringsten ab. Wichtig ist nur, daß sich die obengenannten 7 zusätzlichen Komponenten in dem Schema wiederfinden lassen.

[23] Will man zum Beispiel Folgen über Elementen einer Eingabemenge $\texttt{S}\subseteq\texttt{Set}(\mathbb{Z})$ aufzählen und Kandidatenmengen durch ihr größtes gemeinsames Präfix darstellen (siehe Beispiel 5.5.18), so muß dieser Deskriptor zwangsläufig eine Folge über `S` sein. Ansonsten würde keine sinnvolle Kandidatenmenge mehr dargestellt.

Nur 5 Axiome müssen erfüllt werden, um die Korrektheit des obengenannten Algorithmus sicherzustellen. Die ersten beiden davon sind nahezu trivial,denn sie verlangen nur, daß die Hilfsfunktion $\mathtt{F}_{gs}$ ausschließlich mit sinnvollen Deskriptoren umgeht.

1. Für alle legalen Eingaben ist der Initialdeskriptor sinnvoll.
2. Splitting sinnvoller Deskriptoren liefert ausschließlich sinnvolle Deskriptoren

Das dritte und vierte Axiom garantiert die *Vollständigkeit* der Suche

3. Alle korrekten Lösungen sind in dem Initialdeskriptor enthalten
4. Alle korrekten Lösungen, die in einem Deskriptor `s` enthalten sind, lassen sich aus diesem auch nach endlich vielen `split`-Operationen extrahieren.

Das letzte Axiom garantiert schließlich Terminierung der Suche durch *Wohlfundiertheit* der `split`-Operation.

5. Jeder Deskriptor kann nur endlich oft in Deskriptorenmengen aufgespalten werden, bis keine aufspaltbaren Deskriptoren mehr vorhanden sind.

Um die letzten beiden Axiome formal präzise darstellen zu können, ist es nötig, die Menge aller Deskriptoren, die durch k-fache Interation der `split`-Operation entstehen, induktiv zu definieren.

Definition 5.5.15

Die k-fache Iteration $\underline{split^k}$ einer Operation split: $\mathtt{D}\times\mathtt{S}\to\mathtt{Set(S)}$ *ist induktiv definiert durch*

$$split^{k}(\mathtt{x},\mathtt{s}) \equiv \mathtt{if}\ k{=}\mathtt{0}\ \mathtt{then}\ \{\mathtt{s}\}\ \mathtt{else}\ \bigcup\{\, split^{\mathtt{k-1}}(\mathtt{x},\mathtt{t}) \mid \mathtt{t}\in split(\mathtt{x},\mathtt{s})\,\}$$

Mit dieser Definition können wir die obengenannten Axiome formalisieren und die Korrektheit des schematischen Globalsuchalgorithmus beweisen. Wir lassen dabei zunächst die Filter außer Betracht, da diese später gesondert betrachtet und bis dahin als Bestandteil der `split`-Operation aufgefaßt werden können.

Satz 5.5.16 (Korrektheit von Globalsuch-Algorithmen)

Es sei spec=$(\mathtt{D},\mathtt{R},\mathtt{I},\mathtt{O})$ *eine beliebige Spezifikation,* $\mathtt{S}$ *ein Datentyp,* $\mathtt{J}:\mathtt{D}\times\mathtt{S}\to\mathbb{B}$, $\mathtt{s}_0:\mathtt{D}\to\mathtt{S}$, $\mathtt{sat}:\mathtt{R}\times\mathtt{S}\to\mathbb{B}$, $\mathtt{split}:\mathtt{D}\times\mathtt{S}\to\mathtt{Set(S)}$ *und* $\mathtt{ext}:\mathtt{S}\to\mathtt{Set(R)}$*. Das Programmpaar*

$\mathtt{FUNCTION\ F(x:D):Set(R)\ \ WHERE\ I(x)\ \ RETURNS}\ \{\mathtt{z}\mid\mathtt{O(x,z)}\}\ =\ \mathtt{F}_{gs}(\mathtt{x},\mathtt{s}_0(\mathtt{x}))$

$\mathtt{FUNCTION}\ \mathtt{F}_{gs}\mathtt{(x,s:D\times S):Set(R)\ WHERE\ I(x)}\wedge\mathtt{J(x,s)\ RETURNS}\ \{\mathtt{z}\mid\mathtt{O(x,z)}\wedge\mathtt{sat(z,s)}\}$
$=\{\mathtt{z}\mid\mathtt{z}\in\mathtt{ext(s)}\wedge\mathtt{O(x,z)}\}\cup\bigcup\{\mathtt{F}_{gs}(\mathtt{x},\mathtt{t})\mid\mathtt{t}\in\mathtt{split(x,s)}\}$

ist korrekt, wenn für alle $\mathtt{x}\in\mathtt{D},\ \mathtt{s}\in\mathtt{S}$ *und* $\mathtt{z}\in\mathtt{R}$ *die folgenden fünf Axiome erfüllt sind*

1. $\mathtt{I(x)}\ \Rightarrow\ \mathtt{J(x,s}_0\mathtt{(x))}$
2. $\mathtt{I(x)}\wedge\mathtt{J(x,s)}\ \Rightarrow\ \forall\mathtt{t}\in\mathtt{split(x,s).J(x,t)}$
3. $\mathtt{I(x)}\wedge\mathtt{O(x,z)}\ \Rightarrow\ \mathtt{sat(z,s}_0\mathtt{(x))}$
4. $\mathtt{I(x)}\wedge\mathtt{J(x,s)}\wedge\mathtt{O(x,z)}\ \Rightarrow\ \mathtt{sat(z,s)}\ \Leftrightarrow\ \exists\mathtt{k}{:}\mathbb{N}.\exists\mathtt{t}\in\mathtt{split}^{\mathtt{k}}\mathtt{(x,s).\ z}\in\mathtt{ext(t)}$
5. $\mathtt{I(x)}\wedge\mathtt{J(x,s)}\ \Rightarrow\ \exists\mathtt{k}{:}\mathbb{N}.\ \mathtt{split}^{\mathtt{k}}\mathtt{(x,s)}=\emptyset$

Beweis: Es sei $\mathtt{D},\mathtt{R},\mathtt{I},\mathtt{O},\mathtt{S},\mathtt{J},\mathtt{s}_0,\mathtt{sat},\mathtt{split},\mathtt{ext}$ wie oben angegeben und die Axiome 1–5 seien erfüllt. Dann gilt

1. *Wenn für ein beliebiges* $\mathtt{x}\in\mathtt{D}$ *und* $\mathtt{s}\in\mathtt{S}$ *die Berechnung von* $\mathtt{F}_{gs}\mathtt{(x,s)}$ *nach* `i` *Schritten terminiert (d.h. wenn* $\mathtt{split}^{\mathtt{i}}\mathtt{(x,s)}=\emptyset$ *ist), so ist das Ergebnis der Berechnung die Mennge aller Lösungen, die aus Deskriptoren extrahierbar sind, die zu einem* $\mathtt{split}^{\mathtt{j}}\mathtt{(x,s)}$ *mit* $\mathtt{j<i}$ *gehören:*

$$\mathtt{F}_{gs}\mathtt{(x,s)}\ =\ \bigcup\{\ \{\mathtt{z}\mid\mathtt{z}\in\mathtt{ext(t)}\wedge\mathtt{O(x,z)}\}\ \mid\ \mathtt{t}\in\bigcup\{\mathtt{split}^{\mathtt{j}}\mathtt{(x,s)}\mid\mathtt{j}\in\{\mathtt{0..i\text{-}1}\}\}\ \}$$

Dies läßt sich durch Induktion über `i` und viele Detailberechnungen wie folgt beweisen.

- i=0 ist nach Definition $\texttt{split}^{\texttt{i}}\texttt{(x,s)} = \{\texttt{s}\} \neq \emptyset$, da die Berechnung von $\texttt{F}_{gs}\texttt{(x,s)}$ nicht anhalten kann, ohne mindestens einen Schritt auszuführen. Damit ist die Behauptung bewiesen, da die Voraussetzungen nicht erfüllt sind, und somit ist die Induktion verankert.
- Da die obige Beweisführung zwar logisch korrekt ist, den meisten Menschen jedoch suspekt erscheint, hier noch eine zweite Verankerung bei `i=1`.
  Es ist $\texttt{split}^{1}\texttt{(x,s)} = \bigcup\{\, \texttt{split}^{0}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)} \,\} = \bigcup\{\{\texttt{t}\} \mid \texttt{t} \in \texttt{split(x,s)}\} = \texttt{split(x,s)}$
  Ist also $\texttt{split}^{1}\texttt{(x,s)} = \emptyset$, so folgt
  $\texttt{F}_{gs}\texttt{(x,s)}$
  $= \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\} \cup \bigcup\{\, \texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)} \,\}$
  $= \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\} \cup \bigcup\{\, \texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \emptyset \,\}$
  $= \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\}$
  $= \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \{\texttt{s}\} \,\}$
  $= \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \texttt{split}^{0}\texttt{(x,s)} \,\}$
  $= \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..1-1}\}\} \,\}$
  Damit ist die Behauptung für `i=1` ebenfalls verankert.
- Es sei für ein `i>0` und alle $\texttt{x} \in \texttt{D}$ und $\texttt{s} \in \texttt{S}$ bewiesen
  $$\texttt{F}_{gs}\texttt{(x,s)} = \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\} \,\},$$
  falls $\texttt{split}^{\texttt{i}}\texttt{(x,s)} = \emptyset$ ist.
- Es gelte $\texttt{split}^{\texttt{i+1}}\texttt{(x,s)} = \emptyset$. Nach Definition 5.5.15 ist $\texttt{split}^{\texttt{i+1}}\texttt{(x,s)} = \bigcup\{\texttt{split}^{\texttt{i}}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)}\}$ und somit folgt nach den üblichen Gesetzen über die Mengenvereinigung, daß $\texttt{split}^{\texttt{i}}\texttt{(x,t)} = \emptyset$ für alle $\texttt{t} \in \texttt{split(x,s)}$ gilt. Damit ist die Induktionsannahme für diese `t` anwendbar und es gilt
  $$\texttt{F}_{gs}\texttt{(x,t)} = \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,t)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\} \,\}$$
  für alle $\texttt{t} \in \texttt{split(x,s)}$. Somit folgt
  $\texttt{F}_{gs}\texttt{(x,s)}$
  $= \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\} \cup \bigcup\{\, \texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)} \,\}$
  $= \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\}$
  $\quad \cup \bigcup\{\, \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,t)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\} \,\} \mid \texttt{t} \in \texttt{split(x,s)} \,\}$
  $= \bigcup\{\, \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)} \,\} \mid \texttt{u} \in \{\texttt{s}\} \,\}$
  $\quad \cup \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{j+1}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..i-1}\}\} \,\}$
  $= \bigcup\{\, \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)} \,\} \mid \texttt{u} \in \texttt{split}^{0}\texttt{(x,s)} \,\}$
  $\quad \cup \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{k}}\texttt{(x,s)} \mid \texttt{k} \in \{\texttt{1..i}\}\} \,\}$
  $= \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(u)} \wedge \texttt{O(x,z)}\} \mid \texttt{u} \in \bigcup\{\texttt{split}^{\texttt{k}}\texttt{(x,s)} \mid \texttt{k} \in \{\texttt{0..i}\}\} \,\}$
  Damit ist die Behauptung auch für `i+1` beweisen.

Es sei angemerkt, daß durch das zweite Axiom sichergestellt ist, daß die Funktion $\texttt{F}_{gs}$ auf Eingaben `(x,t)` mit $\texttt{t} \in \texttt{split(x,s)}$ überhaupt anwendbar ist.

2. *Das Hilfsprogramm* $\texttt{F}_{gs}$ *ist für alle zulässigen Eingaben korrekt.*

Es sei $\texttt{x} \in \texttt{D}$ und $\texttt{s} \in \texttt{S}$ mit $\texttt{I(x)} \wedge \texttt{J(x,s)}$. Aufgrund von Axiom 5 (Wohlfundiertheit) gibt es dann eine Zahl $\texttt{k} \in \mathbb{N}$ mit der Eigenschaft $\texttt{split}^{\texttt{k}}\texttt{(x,s)} = \emptyset$. Für dieses `k` gilt nach (1.):

$$\texttt{F}_{gs}\texttt{(x,s)} = \bigcup\{\, \{\texttt{z} \mid \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}\} \mid \texttt{t} \in \bigcup\{\texttt{split}^{\texttt{j}}\texttt{(x,s)} \mid \texttt{j} \in \{\texttt{0..k-1}\}\} \,\}$$

Für einen beliebigen Ausgabewert $\texttt{z} \in \texttt{R}$ gilt somit

$$\texttt{z} \in \texttt{F}_{gs}\texttt{(x,s)} \;\Leftrightarrow\; \exists \texttt{j} \in \{\texttt{0..k-1}\}.\, \exists \texttt{t} \in \texttt{split}^{\texttt{j}}\texttt{(x,s)}.\, \texttt{z} \in \texttt{ext(t)} \wedge \texttt{O(x,z)}$$

Da $\texttt{split}^{\texttt{k}}\texttt{(x,s)} = \emptyset$ ist, läßt sich dies verallgemeinern und umschreiben zu

$$\texttt{z} \in \texttt{F}_{gs}\texttt{(x,s)} \;\Leftrightarrow\; \texttt{O(x,z)} \;\wedge\; \exists \texttt{j}{:}\mathbb{N}.\, \exists \texttt{t} \in \texttt{split}^{\texttt{j}}\texttt{(x,s)}.\, \texttt{z} \in \texttt{ext(t)}$$

Mit Axiom 4 folgt nun $\texttt{z} \in \texttt{F}_{gs}\texttt{(x,s)} \;\Leftrightarrow\; \texttt{O(x,z)} \;\wedge\; \texttt{sat(z,s)}$, also $\texttt{F}_{gs}\texttt{(x,s)} = \{\, \texttt{z} \mid \texttt{O(x,z)} \wedge \texttt{sat(z,s)} \}$. Damit ist die Hilfsfunktion korrekt

$\texttt{FUNCTION F}_{gs}\texttt{(x,s:D×S):Set(R) WHERE I(x)} \wedge \texttt{J(x,s) RETURNS } \{\, \texttt{z} \mid \texttt{O(x,z)} \wedge \texttt{sat(z,s)} \}$
$= \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\} \cup \bigcup\{\, \texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)} \,\}$

3. *Das Hauptprogramm* `F` *ist für alle zulässigen Eingaben korrekt.*

Es sei $x \in D$ mit `I(x)`. Nach Axiom 1 folgt dann `J(x,`s_0`(x))` und damit ist `(x,`s_0`(x))` eine zulässige Eingabe für die Hilfsfunktion F_{gs}. Da F_{gs} auf solchen Eingaben korrekt ist, folgt

$$F(x) = F_{gs}(x, s_0(x)) = \{ z \mid O(x,z) \wedge sat(z, s_0(x)) \}$$

Aufgrund von Axiom 3 gilt somit für einen beliebigen Ausgabewert $z \in R$

$$z \in F(x) \Leftrightarrow O(x,z) \wedge sat(z, s_0(x)) \Leftrightarrow O(x,z)$$

Damit ist also $F(x) = \{ z \mid O(x,z) \}$ und somit ist das Hauptprogramm `F` ebenfalls korrekt.

$$\texttt{FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS } \{ z \mid O(x,z) \} = F_{gs}(x, s_0(x))$$ □

Satz 5.5.16 gibt uns also eine präzise Beschreibung der Komponenten, die für die Erzeugung von Globalsuchalgorithmen nötig sind, und der Axiome, die erfüllt sein müssen, um die Korrektheit des schematisch erzeugten Algorithms zu garantieren. Darüber hinaus beschreibt er implizit eine Strategie zur Erzeugung von korrekten Globalsuchalgorithmen: es reicht, die zusätzlichen Komponenten `S,J,`s_0`,sat,split` und `ext` zu bestimmen, die fünf Axiome zu überprüfen und dann den schematischen Algorithmus zu instantiieren. Offen bleibt jedoch, *wie* diese 6 Komponenten effizient gefunden werden können, und wie man sich die Überprüfung der schwierigeren Axiome 4 und 5 zur Laufzeit des Syntheseprozesses erleichtern kann.

Die Antwort hierfür ist verblüffend einfach: *wir holen die wesentlichen Informationen direkt aus einer Wissensbank*. Dies heißt natürlich nicht, daß nun für jeden möglichen Algorithmus die 6 zusätzlichen Komponenten direkt in der Wissensbank zu finden sind,aber es bedeutet in der Tat, daß vorgefertigte Suchstrukturen – sogenannte *Globalsuchtheorien* – in der Wissensbank abgelegt werden. Diese beschreiben die Komponenten von allgemeinen Globalsuchalgorithmen, welche die grundsätzlichen Techniken zur Aufzählung der Elemente dieser Bereiche widerspiegeln und mit Hilfe des Spezialisierungsmechanismus aus dem vorigen Abschnitt (siehe Definition 5.5.10 und Strategie 5.5.14 auf Seite 255) auf eine konkrete Problemstellung angepaßt werden können. Die Erfahrung hat gezeigt, daß nur 6 dieser vorgefertigten Globalsuchtheorien ausreichen, um alle in der Programmsynthese jemals erforschten Probleme zu synthetisieren.

Es hat sich als sinnvoll herausgestellt, Globalsuchtheorien als ein Objekte zu definieren, die aus 10 Komponenten `(D,R,I,O,S,J,`s_0`,sat,split,ext)` bestehen, welche die ersten 4 Axiome erfüllen. Dies liegt darin begründet, daß nicht alle allgemeinen Aufzählungsverfahren von sich aus wohlfundiert sind, sondern meist erst im Rahmen der Spezialisierung durch Hinzunahme sogenannter *notwendiger Filter* zu wohlfundierten Algorithmen verfeinert werden. Diese Filter müssen sich aus den Ausgabebedingungen des Problems (und dem Erfüllbarkeitsprädikat) ableiten lassen und modifizieren die `split`-Operation durch Elimination von Deskriptoren, die nicht mehr zu einer Lösung beitragen können, zu einer effizienteren und wohlfundierten Operation. Da Wohlfundiertheit zur Laufzeit eines Syntheseprozesses im allgemeinen nur sehr schwer zu beweisen ist – immerhin müssten hierzu Induktionsbeweise automatisch (!) geführt werden – lohnt es sich ebenfalls, in der Wissensbank zu jeder Globalsuchtheorie, die nicht bereits wohlfundiert ist, eine kleine Anzahl vorgefertigter Filter bereitzustellen, welche dafür sorgen, daß auch das fünfte Axiom erfüllt wird. Diese *Wohlfundiertheits-Filter* müssen zur Laufzeit dann nur noch daraufhin geprüft werden, ob sie zu der Ausgabebedingung des konkreten Problems passen.

Mit diesen vorgefertigten Informationen kann das Problem, passend zu einer gegebenen Spezifikation 6 neue Komponenten zu bestimmen und 5 Axiome zu überprüfen, darauf reduziert werden, eine geeignete Globalsuchtheorie und einen passenden Wohlfundiertheits-Filter aus der Wissensbank auszuwählen, die Generalisierungseigenschaft nachzuweisen, die Globalsuchtheorie samt Filter zu spezialisieren und – wenn dies *leicht* möglich ist – durch einen zusätzlichen notwendigen Filter weiter zu verfeinern. Anschließend kann dann mit Hilfe von Satz 5.5.16 ein korrekter Globalsuchalgorithmus für die Spezifikation erzeugt werden.

Um diese Strategie zur Konstruktion von Globalsuchalgorithmen genauer beschreiben zu können, müssen wir die Konzepte *Globalsuchtheorie*, *Filter* und *Spezialisierung* präze definieren und zeigen, wie sie dazu beitragen können, zu einer gegebenen Spezifikation die Voraussetzungen für die Anwendbarkeit von Satz 5.5.16 zu schaffen.

Definition 5.5.17 (Globalsuchtheorie)

Eine Globalsuchtheorie ist ein 10-Tupel $G=((D,R,I,O),S,J,s_0,sat,split,ext)$, *wobei* (D,R,I,O) *eine Spezifikation ist,* S *ein Datentyp,* $J: D\times S \to \mathbb{B}$, $s_0: D\to S$, $sat: R\times S \to \mathbb{B}$, $split: D\times S \to Set(S)$, $ext: S\to Set(R)$ *und für alle* $x\in D$, $s\in S$ *und* $z\in R$ *die folgenden vier Bedingungen gelten.*

1. $I(x) \Rightarrow J(x,s_0(x))$
2. $I(x)\wedge J(x,s) \Rightarrow \forall t\in split(x,s).J(x,t)$
3. $I(x)\wedge O(x,z) \Rightarrow sat(z,s_0(x))$
4. $I(x)\wedge J(x,s)\wedge O(x,z) \Rightarrow sat(z,s) \Leftrightarrow \exists k:\mathbb{N}.\exists t\in split^k(x,s).\, z\in ext(t)$

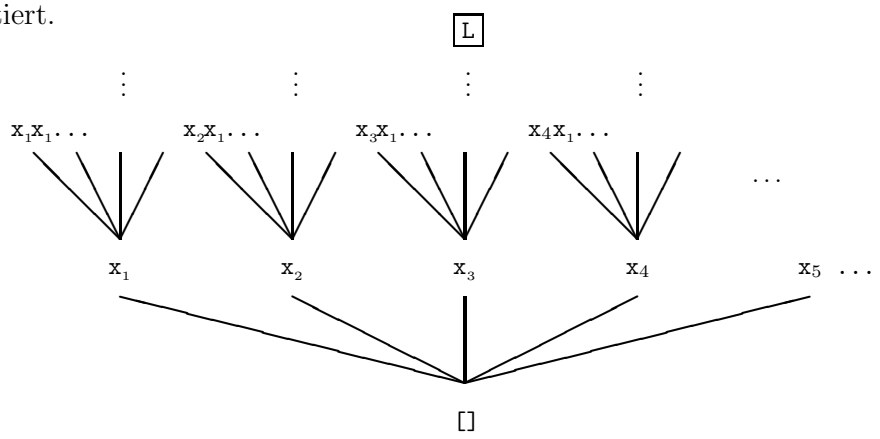
Eine Globalsuchtheorie G heißt wohlfundiert, wenn darüberhinaus für alle $x\in D$ *und* $s\in S$ *gilt:*

$$I(x)\wedge J(x,s) \Rightarrow \exists k:\mathbb{N}.\ split^k(x,s)=\emptyset$$

Wohlfundierte Globalsuchtheorien erfüllen also genau die Voraussetzungen von Satz 5.5.16 und das Ziel einer Synthese ist also, derartige wohlfundierte Globalsuchtheorien mit Hilfe vorgefertigter Informationen zu generieren. In der Wissensbank müssen daher verschiedene allgemeine Suchstrukturen für die wichtigsten *Ausgabebereiche* als Globalsuchtheorien abgelegt werden. Wir wollen hierfür ein Beispiel betrachten.

Beispiel 5.5.18 (Globalsuchtheorie für Folgen über einer endlichen Menge)

Die vorgefertigte Globalsuchtheorie $gs\_seq\_set(\alpha)$ beschreibt eine allgemeine Suchstruktur für Folgen $L\in Seq(\alpha)$, deren Elemente aus einer gegebenen endlichen Menge[24] $S\subseteq\alpha$ stammen müssen. Die Suche geschieht durch Aufzählung der Elemente von L, was bedeutet, daß schrittweise alle Präfixfolgen von L aufgebaut werden, bis die gesuchte Folge gefunden ist. Dieses Suchverfahren wird durch folgenden Suchbaum repräsentiert.



Dieser Baum zeigt, daß die jeweiligen Kandidatenmengen des Suchverfahrens Mengen von Folgen sind, welche die gleiche Präfixfolge V besitzen, also die Gestalt $\{L \mid VpreL\}$ haben, Es bietet sich daher an die Präfixfolge V als Deskriptor und $\lambda L,V.\ VpreL$ als Erfüllbarkeitsprädikat zu verwenden. Sinnvoll sind natürlich nur solche Deskriptoren, die ausschließlich aus Elementen von S bestehen. Die Suche beginnt üblicherweise mit der Menge aller Folgen über S, also mit $[]$ als Initialdeskriptor.

Da bei der Suche schrittweise die Elemente von L aufgezählt werden, ist das Aufspalten von Kandidatenmengen nichts anderes als eine Verlängerung der Deskriptorfolge V um ein weiteres Element der Menge S, was einer Auffächerung der Kandidatenmengen im Baum entspricht. Ist die gesamte Folge L aufgezählt, so ist die Suche beendet und L kann direkt extrahiert werden: Extraktion bedeutet somit, die gesamte Präfixfolge V auszuwählen.

Faßt man all dies in einer Globalsuchtheorie zusammen, so ergibt sich das folgende Objekt, das wir der Lesbarkeit halber komponentenweise beschreiben.

[24] Der Name S steht an dieser Stelle als Platzhalter für eine typische Variable vom Typ Set und darf nicht verwechselt werden mit dem Platzhalter S bei der Beschreibung der Komponenten von Globalsuchtheorien, der das Wort "Suchraum" abkürzt. Unglücklicherweise treten hier Konflikte bei der Vergabe von Standardnamen auf, die nur aus dem Kontext her auflösbar sind.

$$
\begin{array}{lllll}
\underline{\texttt{seq\_set}(\alpha)} \equiv & \texttt{D} & \mapsto & & \texttt{Set}(\alpha) \\
& \texttt{R} & \mapsto & & \texttt{Seq}(\alpha) \\
& \texttt{I} & \mapsto & \lambda\texttt{S}. & \texttt{true} \\
& \texttt{O} & \mapsto & \lambda\texttt{S},\texttt{L}. & \texttt{range(L)} \subseteq \texttt{S} \\
& \texttt{S} & \mapsto & & \texttt{Seq}(\alpha) \\
& \texttt{J} & \mapsto & \lambda\texttt{S},\texttt{V}. & \texttt{range(V)} \subseteq \texttt{S} \\
& \texttt{s}_0 & \mapsto & \lambda\texttt{S}. & [] \\
& \texttt{sat} & \mapsto & \lambda\texttt{L},\texttt{V}. & \texttt{V} \sqsubseteq \texttt{L} \\
& \texttt{split} & \mapsto & \lambda\texttt{S},\texttt{V}. & \{\, \texttt{V}\cdot\texttt{i} \mid \texttt{i} \in \texttt{S} \,\} \\
& \texttt{ext} & \mapsto & \lambda\texttt{V}. & \{\texttt{V}\}
\end{array}
$$

Man beachte, daß es sich hierbei um eine generische Globalsuchtheorie handelt, die auf beliebigen Datentypen α definiert ist und sehr vielen Suchalgorithmen, die auf endlichen Folgen operieren, zugrundeliegt.

Der Beweis, daß `gs_seq_set(`α`)` tatsächlich eine Globalsuchtheorie im Sinne von Definition 5.5.17 ist, ist eine einfache Übungsaufgabe. Man beachte jedoch, daß `gs_seq_set(`α`)` nicht wohlfundiert ist, da mit dieser Suchstruktur Folgen beliebiger Länge aufgezählt werden können.

Mit der Globalsuchtheorie `gs_seq_set(`α`)` haben wir eine allgemeine Suchstruktur für endliche Listen (bzw. Folgen) formalisiert. Wie können wir nun derartige Informationen dazu nutzen, um ein Suchverfahren für eine *konkrete* Problemstellung zu generieren?

Wie bereits angedeutet, wollen wir uns hierzu den Spezialisierungsmechanismus zunutze machen, den wir erstmals in Strategie 5.5.14 auf Seite 255 verwendet haben. Hierzu brauchen wir nur nachzuweisen, daß der Spezifikationsanteil einer Globalsuchtheorie wie `gs_seq_set(`α`)` eine gegebene Spzifikation im Sinne von Definition 5.5.10 generalisiert, aus dem Nachweis eine Transformation θ zu extrahieren, und die Globalsuchtheorie mit θ auf das Ausgangsproblem zu spezialisieren. Wir wollen dies zunächst an einem Beispiel erklären.

Beispiel 5.5.19 (Spezialisierung von `gs_seq_set(`$\mathbb{Z}$`)`)

In Beispiel 5.2.4 auf Seite 228 hatten wir eine Formalisierung des Costas-Arrays Problems vorgestellt, bei dem es um die Bestimmung aller Permutationen der Zahlen `{1..n}` ging, die in keiner Zeile ihrer Differenzentafel doppelt vorkommende Einträge besitzen. Wir hatten hierfür die folgende formale Spezifikation entwickelt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
   RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
```

Die unterstrichenen Teile markieren dabei die Komponenten `D, R, I` und `O` der Problemstellung.

Da der Ausgabebereich der Funktion `Costas` die Menge der endlichen Folgen über den ganzen Zahlen ist, liegt der Versuch nahe, als Lösungsstruktur eine Spezialisierung der Globalsuchtheorie `gs_seq_set(`$\mathbb{Z}$`)` (d.h. der Typparameter α wurde mit $\mathbb{Z}$ instantiiert) zu verwenden. Wir müssen also zeigen

R' ist eine Obermenge von R und es gilt

$$\forall \texttt{x:D}.\ \texttt{I(x)} \ \Rightarrow\ \exists \texttt{x':D'}.\ \texttt{I'(x')} \wedge \forall \texttt{y:R}.\ \texttt{O(x,y)} \Rightarrow \texttt{O'(x',y)}$$

wobei die gestrichenen Komponenten sich auf `gs_seq_set(`$\mathbb{Z}$`)` und die anderen auf das Costas-Arrays Problem beziehen.

1. Die Globalsuchtheorie `gs_seq_set(`$\mathbb{Z}$`)` ist genau so gewählt worden, daß die Ausgabebereiche `R=Seq(`$\mathbb{Z}$`)` und `R'` übereinstimmen. Damit ist die prinzipielle Generalisierbarkeit gewährleistet.
2. Die Eingabebereiche `D=`$\mathbb{Z}$ und `D'=Set(`$\mathbb{Z}$`)` stimmen nicht überein und müssen aneinander angepaßt werden. Auch dies ist prinzipiell möglich, da Mengen mehr Informationen enthalten als natürliche Zahlen, muß aber im Detail noch nachgewiesen werden.
3. Die Eingabebedingung `I'` von `gs_seq_set(`$\mathbb{Z}$`)` ist immer `true`, braucht also nicht weiter berücksichtigt zu werden.

4. Zu zeigen bleibt also

   $\forall \mathtt{n}{:}\mathbb{Z}.\ \mathtt{n}{\geq}1 \Rightarrow\ \exists \mathtt{S}{:}\mathtt{Set}(\mathbb{Z}).$
   $\forall \mathtt{p}{:}\mathtt{Seq}(\mathbb{Z}).\ \mathtt{perm}(\mathtt{p},\{\mathtt{1..n}\}) \wedge \forall \mathtt{j} \in \mathtt{domain}(\mathtt{p}).\mathtt{nodups}(\mathtt{dtrow}(\mathtt{p},\mathtt{j}))\ \Rightarrow\ \mathtt{range}(\mathtt{p}) \subseteq \mathtt{S}$

   Für diesen Nachweis gibt es eine relativ einfache Heuristik, die fast immer in wenigen Schritten zum Ziel führt und nicht einmal einen vollständigen Theorembeweiser benötigt. Das Ziel der Inferenzen muß sein, durch Auffalten von Definitionen und Anwendung von Lemmata Folgerungen von

   $$\mathtt{perm}(\mathtt{p},\{\mathtt{1..n}\}) \wedge \forall \mathtt{j} \in \mathtt{domain}(\mathtt{p}).\mathtt{nodups}(\mathtt{dtrow}(\mathtt{p},\mathtt{j}))$$

   zu bestimmen, in denen $\mathtt{range}(\mathtt{p})$ auf der linken Seite eines Teilmengenoperators vorkommt und auf der rechten Seite ein Ausdruck steht, der für $\mathtt{S}$ eingesetzt werden kann.

   Auffalten von $\mathtt{perm}(\mathtt{p},\{\mathtt{1..n}\})$ liefert $\mathtt{perm}(\mathtt{p},\{\mathtt{1..n}\}) \Leftrightarrow \mathtt{range}(\mathtt{p}) = \{\mathtt{1..n}\} \wedge \mathtt{nodups}(\mathtt{p})$, woraus wiederum durch Anwendung des Gesetzes über Mengengleichheit und Teilmengenbeziehungen (Lemma B.1.13.13, die Wahl ist eindeutig!) folgt $\mathtt{perm}(\mathtt{p},\{\mathtt{1..n}\}) \Rightarrow \mathtt{range}(\mathtt{p}) \subseteq \{\mathtt{1..n}\}$.

   Damit ist die obige Behauptung gültig, wenn wir für $\mathtt{S}$ die Menge $\{\mathtt{1..n}\}$ wählen.

Aus dem Beweis ergibt sich eine Transformation $\theta = \lambda \mathtt{n}.\{\mathtt{1..n}\}$, die gemäß Theorem 5.5.13 zu einer Spezialisierung des generellen Suchalgorithmus für $\mathtt{gs\_seq\_set}(\mathbb{Z})$ zu einem Lösungsalgorithmus für das Costas-Arrays Problem[25] führt. Diese Spezialisierung führt, wie in Strategie 5.5.14 bereits ausgeführt, zu einer Umwandlung eines Eingabewertes $\mathtt{n}$ von $\mathtt{Costas}$ in einen Eingabewert $\theta(\mathtt{n})$ für den durch $\mathtt{gs\_seq\_set}(\mathbb{Z})$ charakterisierten Lösungsalgorithmus und der Elimination von Lösungen, die nicht die Ausgabebedingung $\mathtt{O}$ des Costas-Arrays Problems erfüllen.

Anstelle den Globalsuchalgorithmus zu einem Lösungsverfahren für das konkrete Syntheseproblem zu modifizieren, kann man auch die Globalsuchtheorie $\mathtt{gs\_seq\_set}(\mathbb{Z})$ selbst spezialisieren und in einer Globalsuchtheorie für das Costas-Arrays Problem umwandeln. Hierzu muß man nur den Spezifikationsanteil durch die speziellere konkrete Spezifikation austauscht und in den anderen Komponenten jedes Vorkommen der Menge $\mathtt{S}$ durch $\theta(\mathtt{n})$ ersetzt. Dies führt dann zu folgender spezialisierter Globalsuchtheorie, die in der Tat eine Globalsuchtheorie für das Costas-Arrays Problem ist.

$\mathtt{D}$	$\mapsto$	$\mathtt{Set}(\mathbb{Z})$
$\mathtt{R}$	$\mapsto$	$\mathtt{Seq}(\mathbb{Z})$
$\mathtt{I}$	$\mapsto$	$\lambda \mathtt{n}.\ \mathtt{n}{\geq}1$
$\mathtt{O}$	$\mapsto$	$\lambda \mathtt{n},\mathtt{p}.\ \mathtt{perm}(\mathtt{p},\{\mathtt{1..n}\}) \wedge \forall \mathtt{j} \in \mathtt{domain}(\mathtt{p}).\mathtt{nodups}(\mathtt{dtrow}(\mathtt{p},\mathtt{j}))$
$\mathtt{S}$	$\mapsto$	$\mathtt{Seq}(\mathbb{Z})$
$\mathtt{J}$	$\mapsto$	$\lambda \mathtt{n},\mathtt{V}.\ \mathtt{range}(\mathtt{V}) \subseteq \{\mathtt{1..n}\}$
$\mathtt{s}_0$	$\mapsto$	$\lambda \mathtt{n}.\ []$
$\mathtt{sat}$	$\mapsto$	$\lambda \mathtt{p},\mathtt{V}.\ \mathtt{V} \sqsubseteq \mathtt{p}$
$\mathtt{split}$	$\mapsto$	$\lambda \mathtt{n},\mathtt{V}.\ \{\mathtt{V}{\cdot}\mathtt{i} \mid \mathtt{i} \in \{\mathtt{1..n}\}\}$
$\mathtt{ext}$	$\mapsto$	$\lambda \mathtt{V}.\ \{\mathtt{V}\}$

Die in Beispiel 5.5.19 vorgenommene Spezialisierung von Globalsuchtheorien ist nur wegen des direkten Zusammenhangs zwischen Globalsuchtheorien und Globalsuchalgorithmen und einem expliziten Vorkommen der Ausgabebedingung innerhalb des schematischen Algorithmus möglich. Sie hat gegenüber der in Strategie 5.5.14 vorgestellten nachträglichen Modifikation des Algorithmus den Vorteil, daß die Spezialisierung gezielter in die Struktur des Algorithmus eingreift und somit zu effizienteren Programmen führt. Sie wird durch das folgende Theorem gerechtfertigt.

Satz 5.5.20 (Spezialisierung von Globalsuchtheorien)

Es sei spec $= (D,R,I,O)$ *eine formale Spezifikation und* $G = (\mathtt{D'},\mathtt{R'},\mathtt{I'},\mathtt{O'},\mathtt{S},\mathtt{J},\mathtt{s}_0,\mathtt{sat},\mathtt{split},\mathtt{ext})$ *eine Globalsuchtheorie. Es gelte* $spec \ll \mathtt{spec}_G = (\mathtt{D'},\mathtt{R'},\mathtt{I'},\mathtt{O'})$. *$\theta$ sei die aus dem Spezialisierungsbeweis extrahierte Transformation.*

Dann ist $G_\theta(spec) = (D,R,I,O,\mathtt{S},\lambda \mathtt{x},\mathtt{s}.\mathtt{J}(\theta(\mathtt{x}),\mathtt{s}),\lambda \mathtt{x}.\mathtt{s}_0(\theta(\mathtt{x})),\mathtt{sat},\lambda \mathtt{x},\mathtt{s}.\mathtt{split}(\theta(\mathtt{x}),\mathtt{s}),\mathtt{ext})$ *eine Globalsuchtheorie. $G_\theta(spec)$ ist wohlfundiert, wenn dies auch für G gilt.*

[25] Man beachte, daß der spezialisierte Lösungsalgorithmus zwar partiell korrekt ist, aber – mangels Wohlfundiertheit der Globalsuchtheorie $\mathtt{gs\_seq\_set}(\mathbb{Z})$ – nicht terminiert, solange keine Filter zur Beschneidung des Suchraums verwendet werden.

Beweis: Siehe Übung 10.3.-a/b □

In den Arbeiten, in denen der systematische Entwurf von Globalsuchalgorithmen erstmals untersucht wurden [Smith, 1987b, Smith, 1991a], ist zugunsten iner einfachere Beschreibung von Satz 5.5.20 der Begriff der Generalisierung aus Definition 5.5.10 auf Globalsuchtheorien wie folgt fortgesetzt worden.

G generalisiert spec, falls spec $\ll \mathtt{spec}_G$ *gilt.*

G generalisiert spec mit θ*, falls gilt* $\mathtt{R} \subseteq \mathtt{R'}$ *und* $\forall \mathtt{x{:}D.\ I(x)} \Rightarrow \mathtt{I'}(\theta(\mathtt{x})) \wedge \forall \mathtt{z{:}R.\ O(x,z)} \Rightarrow \mathtt{O'}(\theta(\mathtt{x}),\mathtt{z})$

Mit diesen Begriffen läßt sich der Satz 5.5.20 dann wie folgt verkürzt ausdrücken

Generalisiert eine Globalsuchtheorie G eine Spezifikation spec mit θ*, so ist* G_θ*(spec) eine Globalsuchtheorie.*

Zusammen mit Satz 5.5.16 beschreibt dieser Satz die Grundlage einer mächtigen Strategie zur Synthese von Globalsuchalgorithmen aus formalen Spezifikationen. Man muß nur eine vorgefertigte Globalsuchtheorie der Wissensbank finden, welche die gegebene Spezifikation generalisiert, diese mit der aus dem Generalisierungsbeweis extrahierten Transformation spezialisieren und dann den schematischen Algorithmus instantiieren.

In manchen Fällen reicht diese Strategie jedoch nicht aus, da die gewählte Globalsuchtheorie – wie zum Beispiel `gs_seq_set(`α`)` – nicht notwendigerweise wohlfundiert sein muß und somit nicht zu terminierenden Algorithmen führt. In diesen Fällen kann die Terminierung durch Hinzunahme eines *Filters* Φ garantiert werden, welcher die Menge der zu betrachtenden Deskriptoren nach jeder `split`-Operation so einschränkt, daß jeder Deskriptor im Endeffekt nur endlich oft aufgespalten werden kann. Natürlich darf dieser Filter keine Deskriptoren der auf das Problem spezialisierten Globalsuchtheorie eliminieren, die noch mögliche Lösungen enthalten. Er muß also eine *notwendige* Bedingung für das Vorhandensein von Lösungen in einem Deskriptor beschreiben.

Für Globalsuchtheorien wie `gs_seq_set(`α`)` werden wir eine Reihe von Filtern angeben, welche die `split`-Operation in eine wohlfundierte Operation $\mathtt{split}_\Phi$ verwandeln. Diese im allgemeinen jedoch zu stark, um notwendig für die allgemeine Globalsuchtheorie zu sein. Erst nach der Spezialisierung der Theorie auf eine gegebene Spezifikation, die üblicherweise mit einer deutlichen Verstärkung der Bedingung an die Lösungen verbunden ist, können manche Wohlfundiertheitsfilter auch zu notwendigen Filtern geworden sein.

Definition 5.5.21 (Filter für Globalsuchtheorien)

Es sei G`=(D, R, I, O, S, J,`s_0`,sat,split,ext)` *eine Globalsuchtheorie und* Φ`: D`$\times$`S`$\rightarrow\mathbb{B}$

1. Φ *ist notwendig für G, falls für alle* $\mathtt{x} \in \mathtt{D}$ *und* $\mathtt{s} \in \mathtt{S}$ *gilt* $\Phi(\mathtt{x,s}) \Leftarrow \exists \mathtt{z{:}R.\ sat(z,s)} \wedge \mathtt{O(x,z)}$
2. Φ *ist ein Wohlfundiertheitsfilter für G, falls für alle* $\mathtt{x} \in \mathtt{D}$ *und* $\mathtt{s} \in \mathtt{S}$ *mit* `I(x)` *und* `J(x,s)` *eine Zahl* $\mathtt{k} \in \mathbb{N}$ *existiert, so daß* $\mathtt{split}_\Phi{}^{\mathtt{k}}\mathtt{(x,s)} = \emptyset$ *ist.*

   Dabei ist $\mathtt{split}_\Phi$ *definiert durch* $\mathtt{split}_\Phi\mathtt{(x,s)} \equiv \{\, \mathtt{t} \mid \mathtt{t} \in \mathtt{split(x,s)} \wedge \Phi\mathtt{(x,t)} \,\}$

Offensichtlich ist ein notwendiger Filter Φ genau dann ein Wohlfundiertheitsfilter für die Globalsuchtheorie G, wenn G_Φ`=(D, R, I, O, S, J,`s_0`,sat,`$\mathtt{split}_\Phi$`,ext)` eine wohlfundierte Globalsuchtheorie ist. Die Eigenschaft "notwendig" stellt dabei die Gultigkeit des vierten Axioms sicher (alle Lösungen sind im Endeffekt extrahierbar) während die Wohlfundiertheit genau dem fünften Axiom entspricht. Wohlfundiertheitsfilter müssen im allgemeinen separat von der Globalsuchtheorie betrachtet werden, da eine allgemeine Suchstruktur durch verschiedene Einschränkungen wohlfundiert gemacht werden kann, ohne daß sich hierdurch an dem eigentlichen Suchverfahren etwas ändert. Diese Einschränkungen sind jedoch nur selten notwendig für die allgemeine Globalsuchtheorie.

Da es für jede allgemeine Suchstruktur normalerweise nur sehr wenige Arten von Wohlfundiertheitsfiltern gibt und der Nachweis der Wohlfundiertheit meist einen komplexen Induktionsbeweis verlangt, der von einem Inferenzsystem kaum automatisch geführt werden kann, empfiehlt es sich, für jede nicht wohlfundierte Globalsuchtheorie eine Reihe vorgefertigter Wohlfundiertheitsfilter in der Wissensbank abzulegen. Wir wollen hierfür einige Beispiele geben.

Beispiel 5.5.22 (Wohlfundiertheitsfilter für `gs_seq_set(`α`)`)

Die folgenden drei Filter sind Wohlfundiertheitsfilter für die Globalsuchtheorie `gs_seq_set(`α`)`.

1. Φ_1`(S,V) = |V|≤k`: Begrenzung der Suche auf Folgen einer festen maximalen Länge `k`.

   Dieser Filter ist relativ leicht zu überprüfen, da nur die Länge eines Deskriptors überprüft werden muß, wird aber nur in den seltensten Fällen notwendig sein. Algorithmen mit diesem Filter terminieren (spätestens) nach `k` Schritten und erzeugen einen Suchbaum der Größenordnung $|S|^k$.

2. Φ_2`(S,V) = |V|≤k*|S|`: Begrenzung der Suche auf Folgen einer maximalen Länge, die von der Größe der Menge `S` abhängt.

   Auch dieser Filter, der für eine relativ große Anzahl von Problemen als notwendig nachgewiesen werden kann, ist leicht zu überprüfen. Algorithmen mit diesem Filter terminieren (spätestens) nach `k*|S|` Schritten und erzeugen einen Suchbaum der Größenordnung $|S|^{k*|S|}$.

3. Φ_3`(S,V) = nodups(V)`: Begrenzung der Suche auf Folgen ohne doppelt vorkommende Elemente.

   Der Test ist etwas aufwendiger, wenn er nicht inkrementell durchgeführt wird. Algorithmen mit diesem Filter terminieren (spätestens) nach `k` Schritten und erzeugen aufgrund der immer stärker werdenen Einschränkungen an die noch möglichen Deskriptoren einen Suchbaum der Größenordnung `|S| !`.

Wohlfundiertheitsfilter werden im Rahmen eines Syntheseprozesses passend zu der Globalsuchtheorie ausgewählt, welche die Problemspezifikation generalisiert. Dementsprechend müssen sie zusammen mit dieser auch spezialisiert werden. Das folgende Lemma zeigt, daß die Wohlfundiertheit bei einer Spezialisierung erhalten bleibt und somit zur Laufzeit der Synthese nicht mehr überprüft werden muß.

Lemma 5.5.23 (Spezialisierung von Filtern)

Es sei spec eine Spezifikation, G eine Globalsuchtheorie und Φ *ein Wohlfundiertheitsfilter für G. G generalisiere spec mit* θ*.*

Dann ist Φ_θ`=λx,s. Φ(θ(x),s)` *ein Wohlfundiertheitsfilter für* G_θ*(*spec*)*

Während die Wohlfundiertheit eines Filters bei dem Aufbau der Wissensbank eines Programmsynthesesystems ein für alle Mal überprüft werden ann, muß die Notwendigkeit des spezialisierten Filters zur Laufzeit einer Synthese überprüft werden. Dies ist jedoch wesentlich einfacher, da hierzu nur ein zielgerichtetes Auffalten von Definitionen und Anwenden von Lemmata erforderlich ist.

Beispiel 5.5.24 (Wohlfundiertheitsfilter für das Costas-Arrays Problem)

In Beispiel 5.5.19 hatten wir die Globalsuchtheorie `gs_seq_set(ℤ)` mit Hilfe von `θ = λn. {1..}` zu einer Globalsuchtheorie G' für das Costas-Arrays Problem spezialisiert.

Wir wollen nun prüfen, welcher der Filter aus Beispiel 5.5.22 zu einem notwendigen Wohlfundiertheitsfilter für G' spezialisiert werden kann. Wir suchen also einen Filter Φ, für den gezeigt werden kann

`perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j)), ∧ V⊑p ⇒ Φ({1..n},V)`

wobei `n∈ℤ` und `V∈Seq(ℤ)` beliebig sind. Hierzu könnte man alle vorgegebenen Filter einzeln durchtesten. Es lohnt sich jedoch auch, durch Auffalten von Definitionen und Anwenden von Lemmata nach *leicht abzuleitenden* Folgerungen von

`perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j)), ∧ V⊑p`

zu suchen, in denen nur `n` und `V` vorkommen, und dann einen Filter zu wählen, der in den Folgerungen enthalten ist. Wir falten daher die Definition von `perm(p,{1..n})` auf und erhalten wie zuvor

`perm(p,{1..n}) ⇔ range(p)={1..n} ∧ nodups(p)`

woraus wiederum durch Anwendung des Gesetzes über Präfixfolgen und `nodups` (Lemma B.2.24.3, die Wahl ist eindeutig!) folgt

$\texttt{perm(p,\{1..n\})} \wedge \texttt{V} \sqsubseteq \texttt{p} \Rightarrow \texttt{nodups(V)}.$

Damit ist $\Phi_{3,\theta} = \lambda\texttt{n,V.\ nodups(V)}$ offensichtlich ein notwendiger Wohlfundiertheitsfilter für G'.

Aus der Ausgabebedingung für das Costas-Arrays Problem lassen sich zusammen mit dem Erfüllbarkeitsprädikat auch noch weitere notwenduge Filter ableiten. Diese können dazu benutzt werden, den soeben gefundenen notwendigen Wohlfundiertheitsfilter weiter zu verfeinern. Die Lemma über Präfixfolgen, begrenzte All-Quantoren, `domain`, `dtrow` und `nodups` führen schließlich zu der Folgerung

$$\forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))}, \wedge\ \texttt{V} \sqsubseteq \texttt{p} \Rightarrow\ \forall \texttt{j} \in \texttt{domain(V).nodups(dtrow(V,j))}$$

so daß wir $\Psi = \lambda\texttt{n.V.}\ \forall \texttt{j} \in \texttt{domain(V).nodups(dtrow(V,j))}$ als zusätzlichen notwendigen Filter verwenden können.

Die Verfeinerung eines einmal gefundenen notwendigen Wohlfundiertheitsfilter Φ_θ für $G_\theta(\mathit{spec})$ durch einen zusätzlichen notwendigen Filter Ψ für $G_\theta(\mathit{spec})$ ist für die Erzeugung eines korrekten Globalsuchalgorithmus nicht unbedingt erforderlich, führt im allgemeinen jedoch zu einer nicht unerheblichen Effizienzsteigerung. Aus diesem Grunde beschränkt man sich auf zusätzliche Filter, die *leicht* aus der Bedingung $\texttt{sat(z,s)} \wedge \texttt{O(x,z)}$ abgeleitet werden können, wobei als Heuristik eine Beschränkung auf eine kleine Anzahl anzuwendender Lemmata sinnvoll erscheint. Es ist nicht schwer zu zeigen, die Kombination der beiden Filter zu einem neuen Filter $\underline{\Xi \equiv \lambda\texttt{x,s.}\Phi_\theta\texttt{(x,s)} \wedge \Psi\texttt{(x,s)}}$ einen notwendigen Wohlfundiertheitsfilter für $G_\theta(\mathit{spec})$ liefert.

Faßt man nun die bisherigen Erkenntnisse (Satz 5.5.16, Satz 5.5.20, Lemma 5.5.23 und obige Anmerkung) zusammen, so ergibt sich die folgende effiziente Strategie, die in der Lage ist, eine Synthese von Globalsuchalgorithmen aus vorgegebenen formalen Spezifikationen mit Hilfe von einigen wenigen Zugriffen auf vorgefertigte Objekte der Wissensbank und verhältnismäßig simplen logischen Schlußfolgerungen erfolgreich durchzuführen.

Strategie 5.5.25 (Wissensbasierte Synthese von Globalsuchalgorithmen)

Gegeben sei eine Problemspezifikation $\texttt{FUNCTION F(x:}D\texttt{):}R\ \texttt{WHERE}\ I\texttt{(x) RETURNS}\ \{\texttt{z} \mid O\texttt{(x,z)}\}$

1. *Wähle eine Globalsuchtheorie G mit Ausgabetyp R (oder einer Obermenge) aus der Wissensbank.*
2. *Beweise, daß G die Spezifikation* $\mathit{spec} = (D, R, I, O)$ *generalisiert.*

   Extrahiere aus dem Beweis eine Transformation θ und bestimme die Globalsuchtheorie $G_\theta(\mathit{spec})$.
3. *Wenn G nicht bereits wohlfundiert war, wähle einen Wohlfundiertheitsfilter Φ für G aus der Wissensbank, für den gezeigt werden kann, daß Φ_θ notwendig für $G_\theta(\mathit{spec})$ ist.*
4. *Bestimme heuristisch einen zusätzlichen notwendigen Filter Ψ für $G_\theta(\mathit{spec})$.*
5. *Instantiiere den folgenden schematischen Globalsuchalgorithmus*

   $\texttt{FUNCTION F(x:}D\texttt{):Set(}R\texttt{)}\ \ \texttt{WHERE}\ I\texttt{(x)}\ \ \texttt{RETURNS}\ \{\texttt{z} \mid O\texttt{(x,z)}\}$
   $= \texttt{if}\ \Phi(\theta\texttt{(x),s}_0(\theta\texttt{(x)})) \wedge \Psi\texttt{(x,s}_0(\theta\texttt{(x)}))\ \ \texttt{then F}_{gs}\texttt{(x,s}_0(\theta\texttt{(x)}))\ \ \texttt{else}\ \emptyset$

   $\texttt{FUNCTION F}_{gs}\texttt{(x,s:}D\times\texttt{S):Set(}R\texttt{)}\ \ \texttt{WHERE}\ I\texttt{(x)} \wedge \texttt{J(x,s)} \wedge \Phi(\theta\texttt{(x),s}) \wedge \Psi\texttt{(x,s)}$
   $\texttt{RETURNS}\ \{\texttt{z} \mid O\texttt{(x,z)} \wedge \texttt{sat(z,s)}\}$
   $= \{\texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)}\} \cup \bigcup\{\texttt{F}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split}(\theta\texttt{(x),s}) \wedge \Phi(\theta\texttt{(x),t}) \wedge \Psi\texttt{(x,t)}\}$

   Das resultierende Programmpaar ist garantiert korrekt.

Wir wollen diese Strategie am Beispiel des Costas-Arrays Problems illustrieren

Beispiel 5.5.26 (Costas Arrays Synthese)

Das Costas Arrays Problem ist die Aufgabe, bei Eingabe einer natürlichen Zahl `n` alle sogenannten Costas-Arrays der Ordnung `n` zu berechnen. Dabei ist ein Costas Array der Größe `n` eine Permutation der Zahlen zwischen 1 und `n`, die in keiner Zeile ihrer Differenzentafel doppelt vorkommende Elemente besitzt. In Beispiel 5.2.4 auf Seite 228 hatten wir folgende formale Spezifikation des Problems aufgestellt.

$\texttt{FUNCTION Costas (n:}\underline{\mathbb{Z}}\texttt{):Set(}\underline{\texttt{Seq(}\mathbb{Z}\texttt{)}}\texttt{)}\ \ \texttt{WHERE}\ \underline{\texttt{n} \geq 1}$
$\texttt{RETURNS}\ \{\texttt{p} \mid \underline{\texttt{perm(p,\{1..n\})}\ \wedge\ \forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))}}\}$

1. Der Ausgabetyp des Problems ist $\texttt{Seq}(\mathbb{Z})$. Wir wählen daher eine Globalsuchtheorie für Folgen ganzer Zahlen, wobei im wesentlichen nur $G = \underline{\texttt{gs\_seq\_set}(\mathbb{Z})}$ aus Beispiel 5.5.18 (Seite 261) in Frage kommt.

2. Der Beweis dafür, daß G die Spezifikation *spec* des Costas-Arrays Problems generalisiert, ist in Beispiel 5.5.19 auf Seite 262 bereits ausführlich besprochen worden. Er liefert die Transformation $\underline{\theta = \lambda \texttt{n}.\{\texttt{1..n}\}}$ und führt zu der folgenden spezialisierten Globalsuchtheorie

$$
\begin{array}{lllll}
G_\theta(spec) \equiv & \texttt{D} & \mapsto & & \texttt{Set}(\mathbb{Z}) \\
& \texttt{R} & \mapsto & & \texttt{Seq}(\mathbb{Z}) \\
& \texttt{I} & \mapsto & \lambda\texttt{n}. & \texttt{n} \geq 1 \\
& \texttt{O} & \mapsto & \lambda\texttt{n},\texttt{p}. & \texttt{perm(p,\{1..n\})} \wedge \forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))} \\
& \texttt{S} & \mapsto & & \texttt{Seq}(\mathbb{Z}) \\
& \texttt{J} & \mapsto & \lambda\texttt{n},\texttt{V}. & \texttt{range(V)} \subseteq \texttt{\{1..n\}} \\
& \texttt{s}_0 & \mapsto & \lambda\texttt{n}. & \texttt{[]} \\
& \texttt{sat} & \mapsto & \lambda\texttt{p},\texttt{V}. & \texttt{V} \sqsubseteq \texttt{p} \\
& \texttt{split} & \mapsto & \lambda\texttt{n},\texttt{V}. & \texttt{\{V·i | i} \in \texttt{\{1..n\}\}} \\
& \texttt{ext} & \mapsto & \lambda\texttt{V}. & \texttt{\{V\}}
\end{array}
$$

3. Da G nicht wohlfundiert ist, wählen wir nun einen Wohlfundiertheitsfilter Φ für G aus der Wissensbank und versuchen zu zeigen, daß Φ_θ notwendig für $G_\theta(spec)$ ist.

   In Beispiel 5.5.22 haben wir drei mögliche Wohlfundiertheitsfilter für G angegeben. Von diesen läßt sich – wie in Beispiel 5.5.24 auf Seite 265 gezeigt – der Filter $\underline{\Phi_3 = \lambda \texttt{S},\texttt{V}.\ \texttt{nodups(V)}}$ am schnellsten[26] als geeignet für die Theorie $G_\theta(spec)$ nachweisen.

4. Ebenso haben wir in Beispiel 5.5.22 den Filter $\underline{\Psi = \lambda \texttt{n}.\texttt{V}.\ \forall \texttt{j} \in \texttt{domain(V).nodups(dtrow(V,j))}}$ als zusätzlichen notwendigen Filter für $G_\theta(spec)$ abgeleitet.

5. Die Instantiierung des Standard-Globalsuchalgorithmus mit den Parametern $G_\theta(spec)$, Φ_3 und Ψ liefert das folgende korrekte Programm zur Bestimmung aller Costas-Arrays der Ordnung $\texttt{n}$

$$
\begin{array}{l}
\texttt{FUNCTION Costas (n:}\mathbb{Z}\texttt{):Set(Seq(}\mathbb{Z}\texttt{))} \quad \texttt{WHERE n} \geq 1 \\
\quad \texttt{RETURNS \{p | perm(p,\{1..n\})} \wedge \forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))\}} \\
= \ \texttt{if nodups([])} \wedge \forall \texttt{j} \in \texttt{domain([]).nodups(dtrow([],j))} \ \texttt{then Costas}_{gs}\texttt{(n,[])} \ \texttt{else} \ \emptyset \\
\\
\texttt{FUNCTION Costas}_{gs} \ \texttt{(n:}\mathbb{Z}\texttt{,V:Seq(}\mathbb{Z}\texttt{)):Set(Seq(}\mathbb{Z}\texttt{))} \\
\quad \texttt{WHERE n} \geq 1 \wedge \texttt{range(V)} \subseteq \texttt{\{1..n\}} \wedge \texttt{nodups(V)} \wedge \forall \texttt{j} \in \texttt{domain(V).nodups(dtrow(V,j))} \\
\quad \texttt{RETURNS \{p | perm(p,\{1..n\})} \wedge \texttt{V} \sqsubseteq \texttt{p} \wedge \forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))\}} \\
= \quad \texttt{\{p | p} \in \texttt{\{V\}} \wedge \texttt{perm(p,\{1..n\})} \wedge \forall \texttt{j} \in \texttt{domain(p).nodups(dtrow(p,j))\}} \\
\quad \cup \bigcup \texttt{\{ Costas}_{gs}\texttt{(n,W) | W} \in \texttt{\{V·i | i} \in \texttt{\{1..n\}\}} \wedge \texttt{nodups(W)} \wedge \forall \texttt{j} \in \texttt{domain(W).nodups(dtrow(W,j))\}}
\end{array}
$$

Natürlich kann man den so entstandenen Algorithmus noch in vielerlei Hinsicht vereinfachen und optimieren. So ist z.B. in der Hauptfunktion der Filter immer gültig und in der Hilfsfunktion wird manches zu kompliziert berechnet. Diese Optimierungen kann man jedoch besser im Nachhinein ausführen, da es bei der schematischen Synthese zunächst einmal darum geht, einen Algorithmus mit einer effizienten Berechnungsstruktur zu zu entwerfen, ohne auf alle optimierbaren Details einzugehen. Die Möglichkeiten einer nachträglichen Optimierung des in diesem Beispiel erzeugten Algorithmus werden wir in Abschnitt 5.6 ausführlicher betrachten.

5.5.3 Einbettung von Entwurfsstrategien in ein formales Fundament

Im vorghergehenden Abschnitt haben wir die theoretischen Grundlagen einer Strategie zur wissensbasierten Synthese von Globalsuchalgorithmen aus formalen Spezifikationen ausführlich untersucht und auf dieser eine sehr effiziente Entwurfsstrategie aufgestellt. Die bisherigen Untersuchungen waren allerdings losgelöst von

[26] Φ_2 wäre genauso gut geeignet, läßt sich aber schwerer überprüfen.

einem formalen logischen Fundament durchgeführt worden und stellen nicht sicher, daß eine *Implementierung* der Strategie tatsächlich nur korrekte Algorithmen generieren kann. Das Problem ist hierbei, daß eine "von-Hand" Codierung einer Strategie immer die Gefahr von Unterlassungsfehlern in sich birgt, die während des Syntheseprozesses nicht mehr auffallen, wenn sich diese Implementierung nicht auf Inferenzregeln eines formalen logischen Kalküls stützt. Daher ist es notwendig, Synthesestrategien in einen logischen Formalismus einzubetten, der die Korrektheit aller ausgeführten Syntheseschritte sicherstellt.

Wie können wir nun eine Entwurfsstrategie wie Strategie 5.5.25 in einen universellen logischen Formalismus wie den der Typentheorie integrieren, ohne dabei die Eleganz und Effizienz der informal beschriebenen Strategie zu verlieren? Wie können wir eine mit natürlichsprachigen Mitteln beschriebene Strategie in einer erheblich formaleren Sprache ausdrücken und dabei genauso natürlich und flexibel bleiben wie zuvor?

Für Strategien auf der Basis algorithmischer Theorien läßt sich diese Frage zum Glück relativ leicht beantworten, da hier auf der semi-formalen Ebene bereits eine Reihe von Vorarbeiten geleistet sind, welche die Grundlage des eigentlichen Syntheseverfahrens bilden. Es ist daher möglich, die entsprechenden Konzepte – wie zum Beispiel Globalsuchtheorien, Generalisierung und Filter – vollständig in der Typentheorie zu formalisieren und die Sätze über die Korrektheit schematischer Algorithmen als formale Theoreme der Typentheorie[27] zu beweisen. Ebenso kann man eine Wissensbank von vorgefertigten Teillösungen – zum Beispiel also konkrete Globalsuchtheorien und Wohlfundiertheitsfilter – dadurch aufstellen, daß man Lemmata über diese konkreten Objekte beweist und in der Library von NuPRL für eine spätere Verwendung während einer Synthese abspeichert. Mit Hilfe einer einfachen Taktik zur automatischen Anwendung von Theoremen kann man dann die Strategie innerhalb des formalen Fundaments (also z.B. in NuPRL) genauso elegant und effizient ablaufen lassen wie ihr informal entworfenes Gegenstück.

Gewonnen werden auf diese Art nicht nur eine Garantie für die Korrektheit aller Schritte sondern auch tiefere Einsichten über die von der Strategie beim Entwurf genutzten Zusammenhänge. Dies wollen wir im folgenden am Beispiel der im vorigen Abschnitt präsentierten Globalsuch-Strategie demonstrieren. Wir beginnen dazu mit einer Formalisierung aller relevanten Grundbegriffe, die bei der Synthese von Globalsuchalgorithmen eine Rolle spielen. Aufgrund der semi-formalen Definitionen des vorigen Abschnitts sind die meisten Definitionen sehr naheliegend und müssen nur präzise aufgeschrieben werden.

Definition 5.5.27 (Formalisierung der Grundkonzepte von Globalsuchalgorithmen)

$$\underline{\texttt{GS}} \equiv (\texttt{D:TYPES} \times \texttt{R:TYPES} \times \texttt{I:D}\to\mathbb{B} \times \texttt{O:D}\times\texttt{R}\to\mathbb{B}) \times \texttt{S:TYPES} \times \texttt{J:D}\times\texttt{S}\to\mathbb{B} \times \texttt{s}_0\texttt{:D}\to\texttt{S} \times \texttt{sat:R}\times\texttt{S}\to\mathbb{B} \times \texttt{split:D}\times\texttt{S}\to\texttt{Set(S)} \times \texttt{ext:S}\to\texttt{Set(R)}$$

$$\underline{split^k} \equiv \texttt{natind}(k;\ \lambda x,s.\{s\};\ n,sp_n.\ \lambda x,s.\bigcup\{sp_n(x,t) \mid t \in split(x,s)\})$$

$$\underline{G \texttt{ is a GS-theory}} \equiv \texttt{let } ((D,R,I,O),\ S,\ J,\ s_0,\ sat,\ split,\ ext) = G \texttt{ in}$$
$$\forall x\texttt{:D.}\ I(x) \Rightarrow s_0(x) \texttt{ hält} \wedge J(x,s_0(x))$$
$$\wedge\ \forall x\texttt{:D}.\forall s\texttt{:S.}\ I(x)\wedge J(x,s) \Rightarrow \forall t \in \texttt{split}(x,s).J(x,t)$$
$$\wedge\ \forall x\texttt{:D}.\forall z\texttt{:R.}\ I(x)\wedge O(x,z) \Rightarrow \texttt{sat}(z,s_0(x))$$
$$\wedge\ \forall x\texttt{:D}.\forall s\texttt{:S.}\ I(x)\wedge J(x,s) \Rightarrow \forall z\texttt{:R.}O(x,z) \Rightarrow \texttt{sat}(z,s) \Leftrightarrow \exists k\texttt{:}\mathbb{N}.\exists t \in \texttt{split}^k(x,s).\ z \in \texttt{ext}(t)$$

$$\underline{\texttt{Filters}(G)} \equiv \texttt{let...=}G\texttt{ in D}\times\texttt{S}\to\mathbb{B}$$

$$\underline{split_\Phi} \equiv \lambda x,s.\ \{\ t \mid t \in split(x,s) \wedge \Phi(x,t)\ \}$$

$$\underline{\Phi \texttt{ necessary for } G} \equiv \texttt{let...=}G\texttt{ in}\ \ \forall x\texttt{:D}.\forall s\texttt{:S.}\ \exists z\texttt{:R.}\ \texttt{sat}(z,s)\wedge O(x,z) \Rightarrow \Phi(x,s)$$

$$\underline{\Phi \texttt{ wf-filter for } G} \equiv \texttt{let...=}G\texttt{ in}\ \ \forall x\texttt{:D}.\forall s\texttt{:S.}\ I(x)\wedge J(x,s) \Rightarrow \exists k\texttt{:}\mathbb{N}.\ \texttt{split}_\Phi{}^k(x,s) = \emptyset$$

$$\underline{G \texttt{ generalizes } spec \texttt{ with } \theta} \equiv \texttt{let...=}G\texttt{ and (D',R',I',O')=}spec\texttt{ in}$$
$$R' \subset R \ \wedge\ \forall x\texttt{:D'.}\ I'(x) \Rightarrow I(\theta(x)) \wedge \forall z\texttt{:R'.}\ O'(x,z) \Rightarrow O(\theta(x),z)$$

$$\underline{\Phi_\theta} \equiv \lambda x,s.\Phi(\theta(x),s)$$

$$\underline{G_\theta(spec)} \equiv \texttt{let...=}G\texttt{ in}\ \ (\texttt{spec},\ S,\ \lambda x,s.J(\theta(x),s),\ \lambda x.s_0(\theta(x)),\ \texttt{sat},\ \lambda x,s.\texttt{split}(\theta(x),s),\ \texttt{ext})$$

[27] Hier erweist es sich als vorteilhaft, daß die Typentheorie ein Kalkül höherer Ordnung ist. Man kann daher ohne Bedenken Theoreme formulieren, in denen über Spezifikationen, Programme Prädikate, Globalsuchtheorien etc. quantifiziert wird.

Mit diesen formalisierten Begriffen kann man nun die Sätze und Lemmata des vorigen Abschnittes als formale Sätze der Typentheorie beweisen und dabei in den Beweisen nahezu das gleiche Abstraktionsniveau erreichen, wie dies bei den weniger formalen Gegenstücken der Fall ist. Diese Sätze und Lemmata lassen sich dann zu einem einzigen Theorem zusammenfassen, welches besagt, was genau die Voraussetzungen für die Erfüllbarkeit einer mengenwertigen Problemspezifikation sind. Dieses Theorem wird dann die Grundlage einer effizienten und verifizierten Realisierung der Strategie 5.5.25 innerhalb eines Beweissystems für einen logisch-formalen Kalkül[28] werden.

Satz 5.5.28 (Entwurf von Globalsuchalgorithmen: formale Voraussetzungen)

$\forall$spec:SPECS. let (D,R,I,O) = spec
in
FUNCTION F(x:D):Set(R) WHERE I(x) RETURNS $\{z \mid O(x,z)\}$ ist erfüllbar
$\Leftarrow$
$\exists$G:GS. G is a GS-Theory
$\wedge$ $\exists\theta$:D$\rightarrow$D$_G$. G generalizes spec with θ
$\wedge$ $\exists\Phi$:Filters(G). Φ wf-filter for G $\wedge$ Φ_θ necessary for G$_\theta$(spec)
$\wedge$ $\exists\Psi$:Filters(G$_\theta$(spec)). Ψ necessary for G$_\theta$(spec)

Beweis: Der formale Korrektheitsbeweis entspricht in seiner Struktur im wesentlichen dem Beweis von Satz 5.5.16, ist aber natürlich etwas aufwendiger, da hier keinerlei gedanklichen Abkürzungen zulässig sind. Auf dem Papier würde er mehr als 10 Seiten einnehmen, weil auch alle Zwischenlemmata über Generalisierung und Filter beweisen werden müssen (Details kann man in [Kreitz, 1992][Anhang C.5] finden). Wir beschränken uns hier daher auf eine grobe Beweisskizze.

Gemäß den Voraussetzungen (der rechten Seite von $\Leftarrow$) haben wir als Hypothesen:
– G:GS mit G = ((D',R',I',O'), S, J, s$_0$, sat, split, ext) und G is a GS-Theory
– θ:D$\rightarrow$D' mit G generalizes spec with θ
– Φ: D'$\times$S$\rightarrow\mathbb{B}$ mit Φ wf-filter for G und Φ_θ necessary for G$_\theta$(spec)
– Ψ: D$\times$S$\rightarrow\mathbb{B}$ mit Ψ necessary for G$_\theta$(spec)

Per Definition ist Erfüllbarkeit von Spezifikationen dasselbe wie die Existenz eines korrekten Lösungsprogramms. Wir geben als Lösung daher den Standard-Globalsuchalgorithmus in geschlossener Form an, wobei wir die Hilfsfunktion per Abstraktion einführen.

letrec f_{gs}(x,s) = $\{z \mid z \in ext(s) \wedge O(x,z)\}$
$\cup \bigcup\{f_{gs}(x,t) \mid t \in split(\theta(x),s) \wedge \Phi(\theta(x),t) \wedge \Psi(x,t)\}$
in
if $\Phi(\theta(x),s_0(\theta(x))) \wedge \Psi(x,s_0(\theta(x)))$ then $f_{gs}(x,s_0(\theta(x)))$ else $\emptyset$

Für den Nachweis der Korrektheit müssen wir nun alle Schritte des Beweises von Theorem 5.5.16 formal durchführen, wobei wir die Beweise von Satz 5.5.20 und Lemma 5.5.23 ebenfalls integrieren (bzw. separat ausführen) müssen. Bis darauf, daß der Beweis nun vollständig formal ist und nur mit Hilfe von Taktiken übersichtlich strukturiert werden kann, treten keine nennenswerten Unterschiede auf. □

Der Aufwand, der für den Beweis von Satz 5.5.28 getrieben werden muß, ist relativ hoch, wird sich aber bei der automatisierten Synthese von Programmen wieder auszahlen. Dadurch daß wir in der Lösung ein effizientes Algorithmenschema explizit als Extrakt-Term bereitgestellt und den Korrektheitsbeweis *ein für alle Mal* gegeben haben, können wir Globalsuchalgorithmen nun dadurch synthetisieren, daß wir im ersten Schritt den formalen Satz 5.5.28 mit Hilfe der Regeln cut und lemma einführen und auf das konkrete Programmierproblem anwenden. Als Teilziel bleibt dann nur noch der Nachweis der Existenz von G, θ, Φ und Ψ und ihrer Eigenschaften. Wir können also innerhalb des strengen formalen Inferenzsystems genauso effizient und elegant vorgehen, wie es in der informalen Strategie 5.5.25 beschrieben ist, ohne dabei einen zusätzlichen

[28] Auch wenn die Formalisierungen im wesentlichen auf der intuitionistischen Typentheorie des NuPRL Systems abgestützt werden, ist Theorem 5.5.28 weitestgehend unabhängig von der Typentheorie und könnte in ähnlicher Weise auch in anderen ähnlich mächtigen Systemen eingesetzt werden. Damit ist – zumindest im Prinzip – das Ziel erreicht, mit formalen logischen Kalkülen das natürliche logische Schließen auf verständliche Art widerspiegeln zu können.

Beweisaufwand[29] zu haben. Da zudem die wichtigsten allgemeinen GS-Theorien und ihre wf-Filter bereits als verifizierte Objekte der Wissensbank (d.h. als Lemmata der NuPRL-Bibliothek) vorliegen, wird auch der Nachweis der entstandenen Teilziele nicht mehr sehr schwierig ausfallen. Allein die Generalisierungseigenschaft und die Notwendigkeit von Filtern wird das Inferenzsystem belasten.

Eine Algorithmenentwurfstaktik, die auf der Basis von Satz 5.5.28 operiert, kann daher in wenigen großen Schritten zu einer vorgegebenen Spezifikation einen verifizierbar korrekten Globalsuchalgorithmus erstellen. Die folgende Beschreibung einer solchen Taktik ist das formale Gegenstück zu Strategie 5.5.25 und kann somit als verifizierte Implementierung dieser Strategie angesehen werden.

Man muß zunächst das Problem präzise formulieren und dann ein "Synthesetheorem" aufstellen, was nichts anderes besagt, als daß wir eine Lösung finden wollen. Nun wenden wir als erstes unser Theorem an und erhalten ein neues Unterziel, was sich aus den Voraussetzungen des Theorems ergibt. Konzeptionell ist nun der wichtigste Schritt vollzogen. Nun müssen noch die Parameter bestimmt werden und das kann entweder von Hand geschehen oder mit Unterstützung des Rechners.

Strategie 5.5.29 (Taktik zur formalen Synthese von Globalsuchalgorithmen)

Gegeben sei ein Syntheseproblem, formuliert als Beweisziel

$\vdash$ `FUNCTION F(x:`D`):`R `WHERE` I`(x) RETURNS {z |` O`(x,z)}` ist erfüllbar

1. *Wende das Synthesetheorem 5.5.28 wie eine Inferenzregel an.*

2. *Löse die entstandenen Unterziele – Bestimmung von* `G`*,* θ*,* Φ *und* Ψ *– wie folgt.*

   (a) *Wähle aus der Bibliothek eine GS-theorie* `G` *mit Ausgabetyp* `R` *oder einem Obertyp davon.*
   Beweise "`G is a GS-Theory`*" durch Aufruf des entsprechenden Lemmas.*

   (b) *Bestimme* θ *entweder durch Benutzerinteraktion oder durch einen Extraktion aus einem konstruktivem Beweis von "*`G generalizes (D,R,I,O)`*" (i.w. gezielte Suche nach anwendbaren Lemmata)*

   (c) *Wähle aus der Bibliothek eine Wohlfundiertheitsfilter* Φ *für* `G`*.*
   Beweise "Φ `wf-filter for G`*" durch Aufruf des entsprechenden Lemmas.*
   Beweise "Φ_θ `necessary for` G_θ`(spec)`*" durch gezielte Suche nach anwendbaren Lemmata*

   (d) *Bestimme heuristisch durch Suche nach anwendbaren Lemmata (mit Begrenzung der Suchtiefe) einen zusätzlichen notwendigen Filter* Ψ*, für den* Ψ `necessary for` G_θ`(spec)` *beweisbar ist.*

3. *Extrahiere eine Instanz des Standard-Globalsuchalgorithmus mit dem NuPRL-Extraktionsmechanismus. Da Taktiken nur auf verifizierten Theoreme und den Inferenzregeln der Typentheorie basieren, ist das generierte Programm korrekt.*

Am Beispiel des Costas-Arrays Problems (vergleiche Beispiel 5.5.26) wollen wir nun demonstrieren, wie die Synthese von Globalsuchalgorithmen innerhalb eines allgemeinen Beweisentwicklungssystems wie NuPRL durchgeführt werden kann. Nahezu alle der in Strategie 5.5.29 beschriebenen Teilschritte lassen sich durch den Einsatz von Taktiken automatisieren. Das hohe Abstraktionsniveau der Formalisierungen läßt es jedoch auch zu, die Synthese im wesentlichen interaktiv durchzuführen, wenn der Benutzer – wovon man ausgehen sollte – ein gewisses Verständnis für die zu lösenden Teilaufgaben[30] mitbringt.

[29] Taktiken, welche Globalsuchalgorithmen ohne Verwendung eines formalen Theorems wie Satz 5.5.28 herleiten, sind dagegen gezwungen, die Korrektheit des erzeugten Programms zur Laufzeit nachzuweisen, was in Anbetracht der komplizierten algorithmischen Struktur nahezu undurchführbar und zumindest sehr rechenaufwendig ist.

[30] Es ist sogar durchaus möglich, durch den eher interaktiven Umgang mit dem System ein wenig von der Methodik zu erlernen, die für einen systematischen Entwurf von Algorithmen erforderlich ist.

Es sei allerdings darauf hingewiesen, daß die Beispielsynthese vorläufig noch fiktiven Charakter hat, da die für die Synthese notwendigen Definitionen und Lemmata noch nicht in das NuPRL System eingebettet wurden und der hier beschriebene ZUstand noch nicht erreicht ist.

Beispiel 5.5.30 (Costas Arrays Synthese mit NuPRL)

Wir beginnen mit einer Formulierung des Problems und verwenden hierzu die in Beispiel 5.2.4 auf Seite 228 aufgestellte formale Spezifikation des Costas-Arrays Problems. Innerhalb von NuPRL wird diese Spezifikation zu dem Beweisziel eines Synthese-Theorems, welches mit Hilfe der in Strategie 5.5.29 beschriebenen Schritte beweisen werden soll.

```
⊢ FUNCTION Costas (n:ℤ):Set( Seq(ℤ) )   WHERE n≥1
    RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
  ist erfüllbar
```

Im Topknoten einer formalen Herleitung stehen wir nun vor der Aufgabe, eine Lösung für das Problem zu finden, und müssen hierzu – wie immer – eine Regel oder Taktik angeben. An dieser Stelle könnten wir viele verschiedene Wege einschlagen, entscheiden uns aber dafür, es mit Globalsuchalgorithmen zu versuchen. Daher geben wir eine Taktik an, welche das Theorem 5.5.28 instantiiert. Das System wendet diese Taktik an und hinterläßt als Teilziele die instantiierten Voraussetzungen des Theorems, wobei zugunsten der Lesbarkeit über die Spezifikation abstrahiert und Typinformation unterdrückt wird.

```
# top

⊢ FUNCTION Costas(n:ℤ):Set( Seq(ℤ) )  WHERE n≥1
    RETURNS {p | perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j))}
  ist erfüllbar

BY call 'Theorem_5.5.28'

1. ⊢  ∃G. G is a GS-Theory
     ∧ ∃θ. G generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for G ∧ Φθ  necessary for Gθ(spec)
     ∧ ∃Ψ. Ψ necessary for Gθ(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j)) )
```

Aus Sicht eines formalen Inferenzsystems war dies der schwierigste Schritt der Synthese, der aber durch den Einsatz von Theorem 5.5.28 drastisch vereinfacht werden konnte. In den folgenden Schritten müssen nun die Komponenten G, θ, Φ und Ψ bestimmt und ihre Eigenschaften überprüft werden.

Im nächsttieferen Knoten der Herleitung (bezeichnet mit `top 1`) müssen wir als erstes eine Globalsuchtheorie G bestimmen, die das Costas-Arrays Problem generalisiert. Diese muß gemäß der Generalisierungsbedingung eine allgemeine Suchstruktur für endliche Folgen über ganzen Zahlen beschreiben. Zudem muß sie in der Bibliothek bereits vorhanden sein. Die Erfahrung hat gezeigt, daß zur Lösung der meisten Routineprobleme insgesamt etwa 6–10 vorgefertigte Globalsuchtheorien ausreichen, von denen maximal 2 oder 3 eine allgemeine Suchstruktur für endliche Folgen charakterisieren. Unter diesen entscheidet das Zusatzkriterium, daß die Ausgabebedingung schwächer sein muß als die des Costas-Arrays Problems. Sollten dann immer noch mehrere Möglichkeiten bestehen, so gibt es mehrere fundamental verschiedene Suchstrukturen, die in dem zu erzeugenden Algorithmus zum Tragen kommen können. Die Auswahl zwischen diesen ist eine Entwurfs-Entscheidung, die eigentlich nur ein Benutzer fällen darf.

Selbst dann, wenn keinerlei automatische Unterstützung bei der Auswahl der Globalsuchtheorie bereitsteht, wird es für einen Benutzer nicht sehr schwer sein, sich für eine passende Globalsuchtheorie zu entscheiden. In diesem Fall ist dies die Theorie gs_seq_set(ℤ), die wir unter Verwendung der Regel `ex_i` [31] angeben. Nach Verarbeitung dieser Regel verbleiben zwei Teilziele. Zum einen müssen wir zeigen, daß gs_seq_set(ℤ) tatsächlich eine Globalsuchtheorie ist und zum zweiten müssen wir nun die anderen Komponenten passend hierzu bestimmen.

[31] Es wäre durchaus möglich, eine summarische Taktik mit Namen `INTRO` bereitzustellen, die anhand der äußeren Struktur der Konklusion bestimmt, daß in diesem Falle die Regel `ex_i` gemeint ist.

```
# top  1

⊢ ∃G. G is a GS-Theory
     ∧ ∃θ. G generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for G ∧ Φθ  necessary for Gθ(spec)
     ∧ ∃Ψ. Ψ necessary for Gθ(spec)
     where spec = ( ℤ, Seq(ℤ),  λn. n≥1,
                    λn,p. perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j)) )
BY ex_i  gs_seq_set(ℤ)  THEN  unfold 'perm'
1. ⊢ gs_seq_set(ℤ) is a GS-Theory
2. ⊢ ∃θ. gs_seq_set(ℤ) generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
         ∧  Φθ necessary for gs_seq_set(ℤ)θ(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)θ(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j)) )
```

Das erste Teilziel lösen wir durch Instantiierung des entsprechenden Lemmas der NuPRL-Bibliothek, ohne das die Globalsuchtheorie `gs_seq(ℤ)` nicht für eine Synthese zur Verfügung stehen würde. Für das zweite werden wir die Definition des Begriffes Permutation auflösen müssen, da wir auf die Bestandteile immer wieder zugreifen werden. Der Übersichtlichkeit halber haben wir dies in diesen Schritt integriert, könnten es aber auch erst im nachfolgenden Schritt tun.

In diesem nächsten Schritt müssen wir eine Transformation θ bestimmen, mit der wir die Generalisierungseigenschaft nachweisen können. Eine automatische Unterstützung hierfür ist möglich, da man hierzu nur die in Beispiel 5.5.19 beschriebenen Schritte durch eine Taktik ausführen lassen muß. Bei einer interaktien Synthese kann ein Benutzer jedoch auch ein intuitives Verständnis von der Generalisierungseigenschaft benuzten, welches besagt, daß die Ausgabebedingung von `gs_seq_set(ℤ)` nach Transformation mit θ aus der Ausgabebedingung des Problems folgen muß.

Sieht man sich die entsprechenden Komponenten von `gs_seq_set(ℤ)` genauer an, so stellt man fest, daß θ eine Zahl in eine Menge umwandeln muß. Im Falle von `gs_seq_set(ℤ)` sind die Elemente der aufgezählten endlichen Folgen aus dieser Eingabemenge zu wählen. In der Spezifikation heißt es dagegen, daß die Elemente der Folgen genau die Menge {1..n} bilden müssen. Es liegt also nahe, die Zahl `n` in die Menge {1..n} zu transformieren. Genau das geben wir als Wert für θ an.

```
# top  1 2

⊢ ∃θ. gs_seq_set(ℤ) generalizes spec with θ
     ∧ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
         ∧ Φθ necessary for gs_seq_set(ℤ)θ(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)θ(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j)) )
BY ex_i  λn.{1..n}
1. ⊢ gs_seq_set(ℤ) generalizes  spec with  λn.{1..n}
     where spec = ......
2. ⊢ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
       ∧ Φθ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ......
```

Von den beiden Teilzielen kann das erste durch eine einfache Beweistaktik gelöst werden, welche die Definition von `generalizes` auffaltet und zielgerichtet nach Lemmata über die vorkommenden Begriffe `range` und `sub` sucht.

Im zweiten Teilziel ist ein Filter Φ zu bestimmen, der die spezialisierte Globalsuchtheorie wohlfundiert machen kann. Da der Nachweis von Wohlfundiertheit üblicherweise einen komplizierten Induktionsbeweis benötigt, darf man diesen auf keinen Fall zur Laufzeit des Syntheseprozesses durchführen, sondern

muß vorgefertigte Filter in der Wissensbank ablegen. Erfahrungsgemäß reichen hierfür 3–4 Filter (siehe Beispiel 5.5.22) aus, um die meisten Routineprobleme zu behandeln. Unter diesen müssen wir einen auswählen, für den wir nachher Φ_θ `necessary for gs_seq_set(ℤ)`$_{\lambda n.\{1..n\}}$`(spec)` beweisen können. Eine automatische Unterstützung hierfür ist möglich, da man hierzu nur die in Beispiel 5.5.24 beschriebenen Schritte durch eine Taktik ausführen lassen muß. Es reicht jedoch auch aus, ein intuitives Verständnis des Begriffs "notwendig" zu verwenden: aus der Ausgabebedingung des Costas Arrays Problems und dem Erfüllbarkeitsprädikat zwischen Ausgabewerten und Deskriptoren muß die Gültigkeit des gewählten Filters folgen.

Da ein Präfix `V` einer Folge `p`, die keine doppelten Elemente hat, selbst wieder keine doppelten Elemente hat, liegt es nahe, den Filter Φ_3=`λS,V.nodups(V)` auszuwählen.

```
# top  1 2 2

⊢ ∃Φ. Φ wf-filter for gs_seq_set(ℤ)
     ∧ Φθ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     ∧ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j))) )
BY [ex_i λS,V.nodups(V)]
1. ⊢ λS,V.nodups(V) wf-filter for gs_seq_set(ℤ)
2. ⊢ λn,V.nodups(V) necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ......
3. ⊢ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ......
```

Von den drei entstandenen Teilzielen folgt das erste wiederum aus einem Lemma der NuPRL-Bibliothek. Für den Beweis des zweiten Ziels lösen wir alle Definitionen auf, um eine einfache Beweistaktik anwendbar zu machen. Diese Beweistaktik – wir haben sie `Prover` genannt – muß nur in der Lage sein, zielgerichtet nach Lemmata über die vorkommenden Begriffe zu suchen und diese schrittweise (wie z.B. in Beispiel 5.5.24) anzuwenden.

```
# top  1 2 2 2

⊢ λn,V.nodups(V) necessary for gs_seq_set(ℤ)λn.{1..n}(spec)
     where spec = ( ℤ, Seq(ℤ), λn. n≥1,
                    λn,p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j))) )
BY [undo_abstractions  THEN unfold ‘specialize‘  THEN unfold ‘necessary‘]
1. ⊢ ∀n.∀V. ∃p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j)) ∧ V⊑p
            ⇒ nodups(V)
     * BY [Prover]
```

Damit fehlt uns nur noch der zusätzliche Filter Ψ, der eigentlich nur aus Effizienzgründen benötigt wird, um den Suchraum des entstehenden Algorithmus frühzeitig zu begrenzen. Wie eben lösen wir dazu erst einmal die Definitionen auf und konzentrieren uns darauf, was unmittelbar aus den Ausgabebedingung des Costas Arrays Problems und dem Erfüllbarkeitsprädikat folgt.

Da wir die Eigenschaften `nodups(p)` und `range(p)={1..n}` bereits ausgenutzt haben, müssen wir nun Folgerungen aus der dritten Bedingung ziehen. Auch hier kann man sowohl eine maschinelle Unterstützung (mit Suche nach passenden Lemmata wie in Beispiel 5.5.24) als auch ein intuitives Verständnis des Problems einsetzen. An einer Skizze macht man sich schnell klar, daß jede Zeile in der Differenzentafel einer Folge `V⊑p` ein Präfix der entsprechenden Zeile in der Differenzentafel von `p` ist und dementsprechend auch die Eigenschaft übernimmt, keine doppelten Vorkommen zu haben.

2	4	1	6	5	3	p
-2	3	-5	1	2		Zeile 1
1	-2	-4	3			Zeile 2
-4	-1	-2				Zeile 3
-3	1					Zeile 4
-1						Zeile 5
						Zeile 6

2	4	1	6			V
-2	3	-5				Zeile 1
1	-2					Zeile 2
-4						Zeile 3
						Zeile 4
						Zeile 5
						Zeile 6

Die Deskriptorfolge muß also selbst ein Costas Array einer kleineren Ordnung sein. Wir wählen also Ψ entsprechend und überlassen den Rest des Beweises wieder der Beweisstrategie `Prover`. Diese wird wie im Falle von Φ_θ durch Suche nach geeigneten Lemmata in der Lage sein, die Notwendigkeit von Ψ nachzuweisen – allerdings mit einer deutlich größeren Anzahl von Inferenzen.

```
# top  1 2 2 3

⊢ ∃Ψ. Ψ necessary for gs_seq_set(ℤ)_{λn.{1..n}}(spec)
     where spec = ......
BY [undo_abstractions  THEN unfold ‘specialize‘  THEN unfold ‘necessary‘]
1. ⊢ ∃Ψ.∀n.∀V. ∃p. nodups(p) ∧ range(p)={1..n} ∧ ∀j∈dom(p).nodups(dtrow(p,j)) ∧ V⊑p
                ⇒ Ψ(n,V)
* BY [ex_i  λn,V. ∀j∈dom(V).nodups(dtrow(V,j)  THEN Prover]
```

Die formale Synthese ist nun beendet: alle Komponenten sind bestimmt und das Synthesetheorem über die Lösbarkeit des Costas Arrays Problems ist bewiesen. Nun kann aus dem Beweis eine Instanz des schematischen Algorithmus extrahiert werde, den wir in Satz 5.5.28 angegeben hatten. Diese beschreibt einen ersten lauffähigen, vor allem aber nachgewiesenermaßen korrekten Algorithmus, denn wir z.B. auch im NuPRL System auswerten könnten.

```
FUNCTION Costas(n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j))}
=
   letrec Costas_gs(n,V) =
        {p | p∈{V} ∧ perm(p,{1..n}) ∧ ∀j∈dom(p).nodups(dtrow(p,j))}
     ∪ ⋃{Costas_gs(n,W) | W∈{V·i|i∈{1..n}} ∧ nodups(W) ∧ ∀j∈dom(W).nodups(dtrow(W,j))}
   in
      if nodups([]) ∧ ∀j∈dom([]).nodups(dtrow([],j))
         then Costas_gs(n,[])   else ∅
```

Dieser Algorithmus entspricht im wesentlichen demjenigen, den wir bereits in Beispiel 5.5.26 auf informale Weise hergeleitet hatten.

Das Beispiel zeigt, daß es auch in einem vollständig formalen Rahmen möglich ist, gut strukturierte Algorithmen in wenigen großen Schritten zu entwerfen – selbst dann, wenn das Ausmaß der automatischen Unterstützung eher gering ist und ein Mensch diese Herleitung alleine steuern müsste. Der Vorteil gegenüber den Verfahren, die dasselbe ohne Abstützung auf einen formalen Kalkül erreichen, ist ein größtmögliches Maß an Sicherheit gegenüber logischen Fehlern bei der Erstellung von Software bei gleichzeitiger Erhaltung einer Kooperation zwischen Mensch und Maschine bei der Softwareentwicklung.

5.5.4 Divide & Conquer Algorithmen

Divide & Conquer ist eine der einfachsten und zugleich gebräuchlichsten Programmiertechniken bei der Verarbeitung strukturierter Datentypen wie Felder, Liste, Bäume, etc. Wie Globalsuchalgorithmen lösen Divide & Conquer Algorithmen ein Programmierproblem durch eine rekursive Reduktion der Problemstellung in

unabhängige Teilprobleme, die im Prinzip parallel weiterverarbeitet und dann zu einer Lösung des Gesamtproblems zusammengesetzt werden können. Während bei Globalsuchalgorithmen jedoch jedes dieser Teilprobleme zu einer eigenen Lösung führt ($\vee$-Reduktion, Reduktion des Ausgabebereichs), werden bei Divide & Conquer Algorithmen *alle* Teillösungen benötigt, um eine einzelne Gesamtlösung zusammenzusetzen ($\wedge$-Reduktion, Reduktion des Eingabebereichs).

Die Grundstruktur, die all diese Algorithmen gemeinsam haben, ist eine rekursive *Dekomposition* der Eingaben in "kleinere" Eingabewerte, auf denen der Algorithmus *rekursiv* aufgerufen werden kann, und ggf. weitere Werte, die durch eine Hilfsfunktion weiterverarbeitet werden. Die so erzielten Werte werden dann wieder zu einer Lösung des *Originalproblems* zusammengesetzt. Eingaben, die sich nicht zerlegen lassen, gelten als *primitiv* und müssen *direkt gelöst* werden. Formalisiert man diese Vorgehensweise in einem einheitlichen Algorithmenschema, so ergibt sich folgende Grundstruktur von Divide & Conquer Algorithmen.

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)
                   else (Compose ∘ G×F ∘ Decompose)(x)
```

Hierbei treten neben den Bestandteilen der eigentlichen Spezifikation die folgenden Komponenten auf.

- Ein Funktion `Decompose`:D → D'×D zur *Dekomposition* von Eingaben in Teilprobleme
- Eine *Hilfsfunktion* `G`:D'×R', die zusammen[32] mit einem rekursiven Aufruf von F auf die Dekomposition der Eingabewerte angewandt wird.
- Eine *Kompositions*funktion `Compose`: R'×R → R, welche rekursiv erzeugte Teillösungen (und die von G erzeugten Zusatzwerte) zu einer Lösung des Gesamtproblemms zusammensetzt.
- Ein *Kontrollprädikat* `primitive` auf D, welches Eingaben beschreibt, die sich nicht zerlegen lassen.
- Eine Funktion `Directly-solve`:D→R, welche für primitive Eingaben eine *direkte Lösung* berechnet.

Die Vielseitigkeit dieses Schemas entsteht vor allem durch die Hilfsfunktion G und ihre Ein- und Ausgabebereiche D' und R'. Sie kann im einfachsten Fall die Identitätsfunktion sein, welche die bei der Dekomposition erzeugten Hilfswerte – wie das erste Element einer endlichen Folge bei einer First/Rest-Zerlegung – überhaupt nicht weiterverarbeitet. Sie kann identisch mit der Funktion F sein, wenn Dekomposition einen Eingabewert in zwei gleichartige Werte zerlegt – wie etwa in linke und rechte Hälfte einer endlichen Folge bei einer ListSplit-Zerlegung. Aber auch kompliziertere Zerlegungen und komplexere Hilfsfunktionen sind denkbar.

6 Axiome mussen erfüllt sein, um die Korrektheit des obigen Algorithmenschemas sicherzustellen. Die meisten davon beschreiben die Spezifikationen der vorkommenden Teilfunktionen. Hinzu kommt ein Axiom über rekursive Zerlegbarkeit (*Strong Problem Reduction Principle* – kurz SPRP) und die Wohlfundiertheit der durch die Dekomposition induzierten Ordnung auf den Eingabewerten.

1. Die direkte Lösung ist korrekt für primitive Eingabwerte.
2. Die Ausgabebedingung ist rekursiv zerlegbar in Ausgabebedingungen für die Dekomposition, die Hilfsfunktion und die Komposition (*Strong Problem Reduction Principle*).
3. Die Dekompositionsfunktion erfüllt ihre Ausgabebedingung und verkleinert dabei die Eingabewerte.
4. Die Kompositionsfunktion erfüllt ihre Ausgabebedingung.
5. Die Hilfsfunktion erfüllt ihre Ausgabebedingung.
6. Die durch die Dekomposition induzierte Verkleinerungsrelation ist eine wohlfundierte Ordnung.

[32] Die Schreibweise G×F ist an dieser Stelle eine Abkürzung für λy',y.(G(y'),F(y)) und f∘g kürzt λx.f(g(x)) ab.

Präzisiert man die obengenannten Axiome durch logische Formeln, so läßt sich die Korrektheit des obengenannten Schemas für Divide & Conquer Algorithmen relativ leicht nachweisen.

Satz 5.5.31 (Korrektheit von Divide & Conquer Algorithmen)

Es seien spec=(D, R, I, O) *eine beliebige Spezifikation,* D' *und* R' *beliebige Datentypen,* Decompose:D → D'×D, G:D'×R', Compose: R'×R → R, primitive:D→𝔹 *und* Directly-solve:D→R. *Das Programmschema*

FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)
= if primitive(x) then Directly-solve(x) else (Compose ∘ G×F ∘ Decompose)(x)

ist korrekt, wenn es Prädikate O_D: D×D'×D → 𝔹, I':D'→𝔹, O': D'×R' → 𝔹, O_C: R'×R×R → 𝔹 *und eine Relation* ≻: D×D → 𝔹 *gibt, so daß die folgenden sechs Axiome erfüllt sind*

1. Directly-solve *erfüllt die Spezifikation*

   FUNCTION F_p(x:D):R WHERE I(x) ∧ primitive(x) RETURNS y SUCH THAT O(x,y)

2. *(SPRP:) Für alle* x,y∈D, y'∈D', z,t∈R *und* z'∈R' *gilt*

   O_D(x,y',y) ∧ O(y,z) ∧ O'(y',z') ∧ O_C(z,z',t) ⇒ O(x,t)

3. Decompose *erfüllt die Spezifikation*

   FUNCTION F_d(x:D):D'×D WHERE I(x) ∧ ¬primitive(x)
   RETURNS y',y SUCH THAT I'(y') ∧ I(y) ∧ x≻y ∧ O_D(x,y',y)

4. Compose *erfüllt die Spezifikation*

   FUNCTION F_c(z,z':R×R'):R RETURNS t SUCH THAT O_C(z,z',t)

5. G *erfüllt die Spezifikation*

   FUNCTION F_G(y':D'):R' WHERE I'(y') RETURNS z' SUCH THAT O'(y',z')

6. ≻ *ist eine wohlfundierte Ordnung auf* D

Beweis: Es seien D,R,I,O,D',R',Decompose,G,Compose,primitive,Directly-solve sowie O_D, I',O',O_C,≻ wie oben angegeben. Die Axiome 1–6 seien erfüllt und F sei definiert durch

F(x) = if primitive(x) then Directly-solve(x) else (Compose ∘ G×F ∘ Decompose)(x)

Da ≻ nach *Axiom 6* wohlfundiert ist, können wir durch strukturelle Induktion über (D,≻)[33] zeigen, daß für alle x∈D mit I(x) die Ausgabebedingung O(x, F(x)) erfüllt ist.

Es sei also x∈D mit I(x) und es gelte O(y, F(y)) für alle y∈D mit x≻y. Wir unterscheiden zwei Fälle.

- x ist 'primitiv'. Dann gilt F(x) = Directly-solve(x).
  Aus *Axiom 1* folgt dann unmittelbar O(x, F(x)).
- Es gilt ¬primitive(x). Dann gilt F(x) = (Compose ∘ G×F ∘ Decompose)(x).
  Nach *Axiom 3* berechnet Decompose(x) dann zwei Werte y'∈D' und y∈D mit den Eigenschaften I'(y'), O_D(x,y',y) und x≻y.
  Für y' ist *Axiom 5* anwendbar und die Berechnung von G liefert einen Wert z'∈R' mit O'(y',z')
  Für y ist die *Induktionsannahme* anwendbar und die Berechnung von F liefert einen Wert z∈R mit O(y,z).
  Auf die entstandenen Werte z und z' kann die Funktion Compose angewandt werden. Nach Axiom 4 liefert die Berechnung von Compose(z,z') einen Wert t mit O_C(z,z',t).
  Insgesamt gilt für die Werte x,y',y,z',z und t also O_D(x,y',y) ∧ O(y,z) ∧ O'(y',z') ∧ O_C(z,z',t)
  Damit ist Axiom 2 anwendbar und es folgt O(x,t)
  Da t identisch mit (Compose ∘ G×F ∘ Decompose)(x) ist, folgt t=F(x), also O(x,F(x))

Damit ist die Ausgabebedingung O(x, F(x)) für alle legalen Eingaben erfüllt.

Aufgrund des Prinzips der strukturellen Induktion über Wohlordnungen ist die Ausgabebedingung O(x, F(x)) somit für alle legalen Eingaben erfüllt. Hieraus folgt per Definition die Korrektheit der Funktion F. □

[33] Diese Form der Induktion, welche manchmal auf *wohlfundierte Ordnungsinduktion* genannt wird, basiert auf der Erkenntnis, daß eine Eigenschaft Q[x] für alle x∈D gilt wenn man zeigen kann, daß Q[x] aus der Annahme ∀y:D. x≻y ⇒ Q[y] folgt.

Satz 5.5.31 behandelt eigentlich nur den Fall, daß die Hilfsfunktion `G` verschieden von `F` ist. Andernfalls müsste die Eingabebedingung `I'` für `G` die Ordnungsbeziehung `x≻y'` enthalten, was formal nicht mehr ganz dem obigen Schema entspricht. Wir wollen dies an dieser Stelle jedoch nicht weiter vertiefen[34] sondern stattdessen einige Entwurfsstrategien ansprechen, welche auf Satz 5.5.31 und auf ähnlichen Sätzen über Divide & Conquer Algorithmen beruhen. Im Gegensatz zu der Strategie zur Entwicklung von Globalsuchalgorithmen enthalten diese Strategien jedoch deutlich größere Freiheitsgrade, die einer heuristischen Steuerung bedürfen, da man auf ein wesentlich geringeres Maß von vorgefertigten Informationen stützen muß. Sie wirken daher nicht ganz so zielgerichtet wie Strategie 5.5.25.

Alle Strategien haben gemeinsam, daß man zunächst mindestens eine der notwendigen Komponenten aus einer Wissensbank auswählt und dann daraus Spezifikationen und Lösungen der jeweils anderen Komponenten bestimmt. Dabei basiert die Herleitung der Spezifikationen meist auf einer Zerlegung des Problems in Teilprobleme mit dem Problemreduktionsprinzip (SPRP), während ihre Lösungen meist eine weitere Verwendung von Teilinformationen aus der Wissensbank oder einen erneuten Syntheseprozeß (der nicht nur auf Divide & Conquer beruht) verlangen. Die Erzeugung *abgeleiteter Vorbedingungen* (wie z.B. des Prädikats `primitive`) spielt hierbei eine wichtige Rolle. Wir wollen exemplarisch eine der Strategien im Detail beschreiben.

Strategie 5.5.32 (Synthese von Divide & Conquer Algorithmen)

Gegeben sei eine Problemspezifikation `FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)`.

1. *Wähle eine einfache Dekompositionsfunktion* `Decompose` *samt ihrer Spezifikation aus der Wissensbank und hierzu passend eine wohlfundierte Ordnungsrelation* $\succeq$ *auf* `D`.
   Hierdurch wird der Datentyp `D` und das Prädikat O_D ebenfalls festgelegt. Axiom 6 ist erfüllt.

2. *Bestimme die Hilfsfunktion* `G` *und ihre Spezifikation heuristisch. Wähle hierfür die Funktion* `F`*, falls* `D'=D` *gilt und sonst die Identitätsfunktion.*
   Hierdurch werden die Prädikate `O'` und `I'` ebenfalls festgelegt. Axiom 5 ist erfüllt.

3. *Verifiziere die Korrektheit von* `Decompose`*. Leite die zusätzlichen Vorbedingungen an Eingabewerte* `x∈D` *ab, unter denen die Spezifikation in Axiom 3 von* `Decompose` *erfüllt wird.*
   Hierdurch wird die Negation des Prädikates `primitive` heuristisch festgelegt.

4. *Konstruiere die Ausgabebedingung* O_C *des Kompositionsoperators durch Bestimmung der Vorbedingungen für die Gültigkeit der Formel* `O_D(x,y',y) ∧ O(y,z) ∧ O'(y',z') ⇒ O(x,t)` *und erfülle so Axiom 2.*
   Synthetisiere eine Funktion `Compose`*, welche Axiom 4 erfüllt.*
   Sollte `Compose` nicht in der Wissensbank zu finden sein, muß das Verfahren rekursiv aufgerufen werden.

5. *Synthetisiere eine Funktion* `Directly-solve`*, welche Axiom 1 erfüllt.*
   Die Funktion muß über einen einfachen Nachweis von `∀x:D. I(x) ∧ primitive(x) ⇒ ∃y:R. O(x,y)` konstruiert werden können. Ist dies nicht möglich, leite eine zusätzliche Vorbedingung für die Gültigkeit dieser Formel ab und starte das Verfahren erneut.

6. *Instantiiere das Divide & Conquer Schema mit den gefundenen Parametern.*

```
FUNCTION F(x:D):R   WHERE I(x) RETURNS y  SUCH THAT O(x,y)
= if primitive(x)  then Directly-solve(x)  else (Compose ∘ G×F ∘ Decompose)(x)
```

Viele dieser Schritte können mit Hilfe einer Technik zur Bestimmung von *Vorbedingungen* für die Gültigkeit einer logischen Formel, die in [Smith, 1985b, Section 3.2] ausführlich beschrieben ist, weitestgehend automatisiert werden. Die Synthese von `Compose` kann einen weiteren Aufruf des Verfahrens erfordern, bei dem eine andere Reihenfolge bei der Bestimmung der Komponenten von Divide & Conquer Algorithmen erfolgreicher sein mag als Strategie 5.5.32. Als alternative Vorgehensweisen werden hierfür in [Smith, 1985b, Section 7] vorgeschlagen.

[34] Ein etwas allgemeineres, allerdings nicht mehr ganz so einfach strukturiertes Schema für Divide & Conquer Algorithmen und den dazugehörigen Korrektheitsbeweis findet man in [Smith, 1982].

- Wähle einen einfachen Kompositionsoperator aus der Wissensbank. Konstruieren dann sukzessive die Hilfsfunktion, die Ordnungsrelation, den Dekompositionsoperator und zum Schluß die direkte Lösung für primitive Eingabewerte.
- Wähle einen einfachen Dekompositionsoperator und die Ordnung $\succ$ aus der Wissensbank. Konstruieren dann sukzessive den Kompositionsoperator, die Hilfsfunktion und zum Schluß die direkte Lösung für primitive Eingabewerte.

Wir wollend die Strategie 5.5.32 am Beispiel der Synthese eines Sortieralgorithmus illustrieren.

Beispiel 5.5.33 (Synthese eines Sortieralgorithmus)

Das Sortierproblem wird durch die folgenden formale Spezifikation beschrieben.

`FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) RETURNS S SUCH THAT SORT(L,S)`

wobei `SORT(`L`,`S`)` eine Abkürzung für `rearranges(`L`,`S`)` $\wedge$ `ordered(`S`)` ist.

1. Listen über ganzen Zahlen lassen sich auf mehrere Arten zerlegen. Wir könnten uns für eine `FirstRest`-Zerlegung, für eine `ListSplit`-Zerlegung in zwei (nahezu) gleiche Hälften, oder eine `SplitVal`-Zerlegung in Werte unter- bzw. oberhalb eines Schwellwertes entscheiden. In diesem Falle wählen wir die Funktion

   `ListSplit` $\equiv$ λ`L. ( [L[i]|i`$\in$`[1..|L|`$\div$`2] ], [L[i]|i`$\in$`[1+|L|`$\div$`2..|L|] ] )`

   Die zugehörige Ausgabebedingung O_D`(L,L`$_1$`,L`$_2$`)` lautet `L`$_1$`∘L`$_2$`=L` $\wedge$ `|L`$_1$`|=|L|`$\div$`2` $\wedge$ `|L`$_2$`|=(1+|L|)`$\div$`2`.

   Eine der wichtigsten wohlfundierten Ordnungen auf Listen besteht in einem Vergleich der Längen. Diese Ordnung ist nicht total, aber darauf kommt es ja nicht an. Wir wählen also λ`L,L'. |L|>|L'|`.

2. Entsprechend der Heuristik in Strategie 5.5.32 wählen wir die Funktion `sort` selbst als Hilfsfunktion. Damit werden die Ein- und Ausgabebedingungen `I` und `O` entsprechend übernommen.

3. Wir stellen nun die vollständige Spezifikation des Dekompositionsoperators – einschließlich der Vorbedingungen an `sort` und der *beiden* Ordnungsbedingungen `L`$\succ$`L`$_1$ $\wedge$ `L`$\succ$`L`$_2$ – auf und leiten die Vorbedingungen für die Korrektheit des Programms

   ```
   FUNCTION F_d(L:Seq(ℤ):Seq(ℤ)×Seq(ℤ)  RETURNS L₁,L₂
     SUCH THAT |L|>|L₁| ∧ |L|>|L₂| ∧ L₁∘L₂=L ∧ |L₁|=|L|÷2 ∧ |L₂|=(1+|L|)÷2
   =  ListSplit(L)
   ```

   her. Da `|L|>|L`$_1$`|` nur gelten kann, wenn `|L|>0`, also `L`$\neq$`[]` ist, erhalten wir als Beschreibung primitiver Eingaben das Prädikat λ`L. L=[]`.

4. Wir leiten nun die Vorbedingungen für die Gültigkeit von

   `L`$_1$`∘L`$_2$`=L` $\wedge$ `|L`$_1$`|=|L|`$\div$`2` $\wedge$ `|L`$_2$`|=(1+|L|)`$\div$`2` $\wedge$ `SORT(L`$_1$`,S`$_1$`)` , $\wedge$ `SORT(L`$_2$`,S`$_2$`)` $\Rightarrow$ `SORT(L,S)`

   ab. Dies liefert als Kompositionsbedingung `ordered(S)` $\wedge$ `range(S)=range(S`$_1$`)`$\cup$`range(S`$_2$`)`

   Die Synthese der Kompositionsfunktion

   ```
   FUNCTION F_c(S₁,S₂:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)  WHERE ordered(S₁)∧ordered(S₂)
     RETURNS S  SUCH THAT ordered(S) ∧ range(S)=range(S₁)∪range(S₂)
   ```

   benötigt einen rekursiven Aufruf des Verfahrens mit einer anderen Reihenfolge der Einzelschritte (siehe [Smith, 1985b, Section 7.2]) und liefert den `merge`-Operator als Kompositionsfunktion.

5. Nun müssen wir nur noch die direkte Lösung für primitive Eingaben bestimmen und überprüfen hierzu $\forall$`L:Seq(ℤ). L=[]` $\Rightarrow$ $\exists$`S:Seq(ℤ). SORT(L,S)`. Der Beweis ist nicht schwer zu führen und liefert `S=L`.

6. Wir instantiieren nun den schematischen Divide & Conquer Algorithmus, abstrahieren dabei über die beiden Resultate von `ListSplit(L)` und erhalten folgenden Sortieralgorithmus.

   ```
   FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) RETURNS S  SUCH THAT rearranges(L,S) ∧ ordered(S)
   =  if L=[] then L
              else let L₁,L₂=ListSplit(L)  in  merge(sort(L₁),sort(L₂))
   ```

Weitere Anwendungsbeispiele für eine Synthese von Divide & Conquer Algorithmen kann man in [Smith, 1983a, Smith, 1985a, Smith, 1987a] finden.

5.5.5 Lokalsuch-Algorithmen

Bei der Lösung von Optimierungsproblemen sind lokale Suchverfahren, oft auch als *Hillclimbing-Algorithmen* bezeichnet, eine beliebte Vorgehensweise. Hierbei wird ausgehend eine optimale Lösung eines Problems dadurch bestimmt, daß man eine beliebige Anfangslösung schrittweise verbessert, indem man die unmittelbare (lokale) Nachbarschaft einer Lösung nach besseren Lösungen absucht. Dabei hat die Nachbarschaftsstruktur des Suchraumes einen großen Einfluß auf das Verfahren. Kleine Nachbarschaften beschleunigen die Suche, können aber dazu führen, daß unbedeutende lokale Extremwerte als globale Optima verstanden werden. Dieses Problem tritt bei größeren Nachbarschaften nicht mehr auf, aber dafür muß man wiederum eine erheblich längere Zeit für die Suche in Kauf nehmen.

Die Konstruktion einer guten Nachbarschaftsstruktur spielt daher eine zentrale Rolle für den Erfolg von Lokalsuchalgorithmen. Hierbei sind zwei wichtige Eigenschaften zu berücksichtigen. Zum einen muß der Suchraum vollständig miteinander verbunden sein (*Konnektivität*), damit eine optimale Lösung von jeder beliebigen Lösung auch erreicht werden kann. Zum anderen müssen lokale Optima auch globale Optima sein (*Exaktheit*), damit lokale Suche nicht bei den falschen Lösungen steckenbleibt. Formalisiert man all diese Voraussetzuneg, so kann man Lokalsuchalgorithmen ebenfalls als algorithmische Theorie darstellen und ein schematisches Verfahren global verifizieren.

Die allgemeine Grundstruktur von Lokalsuchalgorithmen läßt sich durch folgendes Schema beschreiben.

$$
\begin{array}{l}
\texttt{FUNCTION } F_{opt}\texttt{(x:D):R} \quad \texttt{WHERE I(x)} \quad \texttt{RETURNS y} \\
\quad \texttt{SUCH THAT O(x,y)} \ \wedge\ \forall \texttt{t:R.}\ \texttt{O(x,t)} \Rightarrow \texttt{cost(x,y)} \leq \texttt{cost(x,t)} \\
= \ F_{LS}\texttt{(x,F(x))} \\
\\
\texttt{FUNCTION } F_{LS}\texttt{(x:D,z:R):R} \quad \texttt{WHERE I(x)} \wedge \texttt{O(x,y)} \quad \texttt{RETURNS y} \\
\quad \texttt{SUCH THAT O(x,y)} \ \wedge\ \forall \texttt{t} \in \texttt{N(x,y).}\ \texttt{O(x,t)} \Rightarrow \texttt{cost(x,y)} \leq \texttt{cost(x,t)} \\
= \ \texttt{if } \forall \texttt{t} \in \texttt{N(x,z).}\ \texttt{O(x,t)} \ \Rightarrow\ \texttt{cost(x,z)} \leq \texttt{cost(x,t)} \\
\quad\ \texttt{then z} \quad \texttt{else } F_{LS}\texttt{(x, arb(\{t | t} \in \texttt{N(x,z)} \wedge \texttt{O(x,t)} \wedge \texttt{cost(x,z)>cost(x,t)\}))}
\end{array}
$$

Dabei treten neben den üblichen Bestandteilen `D`, `R`, `I` und `O` einer Spezifikation folgende Komponenten auf

- Eine totale *Ordnung* $\boxed{\leq}$ auf einem Kostenraum $\mathcal{R}$.
- Eine *Kostenfunktion* $\boxed{\texttt{cost}}\texttt{:R} \to \mathcal{R}$, die jeder Lösung eine Bewertung zurodnet.
- Eine *Nachbarschaftsstruktur*funktion $\boxed{\texttt{N}}\texttt{: D} \times \texttt{R} \to \texttt{Set(R)}$, die jedem Element des Lösungsraumes eine Menge benachbarter Elemente zuordnet.
- Eine *Initiallösung* $\boxed{\texttt{F(x)}}$, die korrekt im Sinne der Ausgabebedingung `O` ist.

Wie bei den anderen Algorithmentheorien kann man nun Axiome für die Korrektheit von Lokalsuchalgorithmen aufstellen und durch logische Formeln präzisieren. Dabei kann die Konnektivität der Nachbarschaftsstruktur nur über eine Iteration der – durch `O` modifizierten – Nachbarschaftsstrukturfunktion erklärt werden, die analog zur Interation von `split` in Definition 5.5.15 wie folgt definiert ist.

Definition 5.5.34

Die k-fache Iteration $\underline{N_O^k}$ *einer Nachbarschaftsstruktur* $N\texttt{: D} \times \texttt{R} \to \texttt{Set(R)}$ *unter der Bedingung* $O\texttt{: D} \times \texttt{R} \to \mathbb{B}$ *ist definiert durch*

$$N_O^k\texttt{(x,y)} \ \equiv\ \texttt{if } k\texttt{=0 then } \{\texttt{y}\} \texttt{ else } \bigcup \{\, N^{\texttt{k}-1}\texttt{(x,t)} \mid \texttt{t} \in \texttt{N(x,y)} \wedge \texttt{O(x,t)} \,\}$$

Mit dieser Definition können wir nun einen Satz über die Korrektheit von Lokalsuchalgorithmen aufstellen.

Satz 5.5.35 (Korrektheit von Lokalsuchalgorithmen)

Es seien spec=(D, R, I, O) *eine beliebige Spezifikation,* $(\mathcal{R},\leq)$ *ein total geordneter (Kosten-)Raum,* cost:R$\rightarrow\mathcal{R}$, N:D×R$\rightarrow$Set(R) *und* F:D$\rightarrow$R. *Das Programmpaar*

FUNCTION F_{opt}(x:D):R WHERE I(x) RETURNS y
SUCH THAT O(x,y) $\wedge$ $\forall$t:R. O(x,t) $\Rightarrow$ cost(x,y)$\leq$cost(x,t)
= F_{LS}(x,F(x))

FUNCTION F_{LS}(x:D,z:R):R WHERE I(x) $\wedge$ O(x,y) RETURNS y
SUCH THAT O(x,y) $\wedge$ $\forall$t$\in$N(x,y). O(x,t) $\Rightarrow$ cost(x,y)$\leq$cost(x,t)
= if $\forall$t$\in$N(x,z). O(x,t) $\Rightarrow$ cost(x,z)$\leq$cost(x,t)
then z else F_{LS}(x, arb({t | t$\in$N(x,z) $\wedge$ O(x,t) $\wedge$ cost(x,z)>cost(x,t)}))

ist korrekt, wenn für alle x$\in$D *und* y,t$\in$R *die folgenden vier Axiome erfüllt sind*

1. *Korrektheit der Initiallösung:*
   F *erfüllt die Spezifikation* FUNCTION F(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y)
2. *Reflexivität der Nachbarschaftsstruktur:* I(x)$\wedge$O(x,y) $\Rightarrow$ y$\in$N(x,y)
3. *Exaktheit lokaler Optima:*
   I(x) $\wedge$ O(x,y) $\Rightarrow$ $\forall$t$\in$N(x,y). O(x,t) $\Rightarrow$ cost(x,y)$\leq$cost(x,t)
   $\Rightarrow$ $\forall$t:R. O(x,t) $\Rightarrow$ cost(x,y)$\leq$cost(x,t)
4. *Erreichbarkeit gültiger Lösungen:* I(x) $\wedge$ O(x,y) $\wedge$ O(x,t) $\Rightarrow$ $\exists$k:$\mathbb{N}$. t$\in\mathrm{N}_O^k$(x,y)

Auf einen Beweis und die Beschreibung einer konkreten Strategie zur Erzeugung von Lokalsuchalgorithmen wollen wir an dieser Stelle verzichten. Interessierte Leser seien hierfür auf [Lowry, 1988, Lowry, 1991] verwiesen.

5.5.6 Vorteile algorithmischer Theorien für Syntheseverfahren

Praktische Erfahrungen mit bisherigen Ansätzen zur Programmsynthese und vor allem der große Erfolg des KIDS Systems haben gezeigt, daß ein Verfahren zur Synthese von Programmen mit Hilfe von Algorithmenschemata eine Reihe praktischer Vorteile gegenüber anderen Syntheseverfahren hat. Diese liegen vor allem darin begründet, daß formale Schlüssen auf einem sehr *hohen Abstraktionsniveau* stattfinden und auf die Lösung von Teilproblemen abzielen, welche aus Erkenntnissen über Programmiertechniken erklärbar sind. Die Hauptvorteile sind ein *sehr effizientes Syntheseverfahren*, die Erzeugung *effizienter Algorithmenstrukturen*, ein *wissensbasiertes Vorgehen* und ein sehr starkes *formales theoretisches Fundament*.

Das Syntheseverfahren ist vor allem deshalb so effizient, weil der eigentliche Syntheseprozeß durch aufwendige theoretische Voruntersuchungen erheblich entlastet wird. Die tatsächliche Beweislast wird vom Ablauf des Syntheseverfahrens in seine Entwicklung – also in den Entwurf der Algorithmentheorien und ihren Korrektheitsbeweis – verlagert, wo sie nur ein einziges Mal durchgeführt werden muß. Zudem kann die Überprüfung der schwierigeren Axiome einer Algorithmentheorie oft dadurch entfallen, daß Techniken zur Verfeinerung vorgefertigter Teillösungen zur Verfügung stehen. Insgesamt ist ein solches Verfahren erheblich zielgerichteter als andere und somit auch leichter zu verstehen und leichter zu steuern – selbst dann, wenn nur wenig automatische Unterstützung bereitsteht. Die gewünschte *Kooperation zwischen Mensch und Computer* ist somit erstmalig erreichbar: der Mensch kann sich tatsächlich auf Entwurfsentscheidungen konzentrieren während die formalen Details durch den Computer (bzw. die Vorarbeiten) abgehandelt werden.

Auch die erzeugten Algorithmen können erheblich effizienter sein als solche, die zum Beispiel durch (reine) Extraktion aus konstruktiven Beweisen gewonnen werden können. Der Grund hierfür ist einfach: es wird eine sehr *effiziente Grundstruktur vorgegeben* und nach Überprüfung der Axiome instantiiert. Für die Erzeugung der Algorithmen können somit alle bekannten Erkenntnisse über gute Programmstrukturen eingesetzt werden. Ineffiziente Bestandteile, die sich in speziellen Fällen ergeben, bestehen im wesentlichen nur aus Redundanzen und können durch achträgliche Optimierungen beseitigt werden. Zudem ist es ohne weiteres möglich, die Schemata in nahezu jeder beliebigen Programmiersprache zu formulieren – also zum Beispiel das Schema für

Globalsuchalgorithmen aus Satz 5.5.16 auch in C, Pascal, Eiffel, C++, LISP, ML, Prolog oder einer Parallelverarbeitungssprache auszudrücken. An dem entsprechenden Satz und seinem Korrektheitsbeweis würde sich nur wenig ändern, da zusätzlich nur die konkrete Semantik der Programmiersprache in den Beweis integriert werden muß, die zentrale Beweisidee aber nur von der algorithmischen Struktur abhängt. In den anderen Syntheseparadigmen ist der Übergang auf eine andere Programmiersprache verhältnismäßig kompliziert, da hier beliebige Programme übersetzt werden müssen, ohne daß man von Erkenntnissen über deren Struktur Gebrauch machen kann.

Syntheseverfahren auf der Basis algorithmischer Theorien sind die einzigen Verfahren, bei denen man wirklich von einer Verarbeitung von *Wissen* sprechen kann, da hier Erkenntnisse über Algorithmen als Theoreme vorliegen. Während alle Syntheseverfahren in der Lage sind, Lemmata über verschiedene Anwendungsbereiche mehr oder weniger explizit zu verarbeiten, findet die Verarbeitung von Programmierwissen eigentlich nur bei der Anwendung algorithmischer Theorien statt. Bei andersartigen Verfahren ist Programmierwissen bestenfalls zu einem geringen Anteil – und auch nur implizit – in der Codierung der Synthesestrategie enthalten, welche die Anwendung elementarer Inferenz- bzw. Transformationsregeln steuert.

Das formale theoretische Fundament ist die Basis all dieser guten Eigenschaften. Es sichert die Korrektheit der erzeugten Algorithmen auf eine leicht zu verstehende Art und Weise und weist auch den Weg für eine unkomplizierte realisierung von Synthesestrategien mit einem Programm- und Beweisentwicklungssystem für einen universellen logischen Formalismus wie die intuitionistische Typentheorie.

5.6 Nachträgliche Optimierung von Algorithmen

Algorithmen, die durch automatisierte Syntheseverfahren aus formalen Spezifikationen erzeugt wurden, sind nur selten optimal, da ein allgemeines Syntheseverfahren nicht auf alle Details des Problems eingehen kann, welches durch den zu erzeugenden Algorithmus gelöst werden soll. Zwangsläufig bleiben nach der Synthese gewisse Redundanzen und ineffiziente Teilberechnungen zurück, die ein geschulter menschlicher Programmierer schnell als solche erkennen würde. Sie im Voraus zu vermeiden würde aber den eigentlichen Syntheseprozeß unnötig belasten, da die Suche nach anwendbaren Umformungen erheblich ausgedehnt und durch Heuristiken, welche die Effizienz bestimmter Teilkonstrukte bewerten, gesteuert werden müsste. Stattdessen bietet es sich an, *nachträgliche* Optimierungen vorzunehmen, die von einem Menschen gesteuert werden, und hierzu eine Reihe von Grundtechniken bereitzustellen, welche die Korrektheit der Optimierungsschritte garantieren.

Programmtransformationen zur Optimierung von Algorithmen sind ein großes Forschungsgebiet für sich, das an dieser Stelle nicht erschöpfend behandelt werden kann. Wir wollen uns stattdessen auf die Beschreibung einiger einfacher Optimierungstechniken konzentrieren, die sich gut in ein formales Synthesesystem integrieren lassen. Die meisten dieser Techniken sind bereits in dem KIDS System [Smith, 1990, Smith, 1991a] enthalten und haben sich als sehr wirksame Hilfsmittel bei der wissensbasierten Erzeugung effizienter Algorithmen aus formalen Spezifikationen erwiesen. Als Leitbeispiel verwenden wir eine nachträgliche Optimierung des Lösungsalgorithmus für das Costas Arrays Problem, den wir in Beispiel 5.5.26 auf Seite 266 synthetisch erzeugt hatten.

5.6.1 Simplifikation von Teilausdrücken

Die einfachste aller Optimierungstechniken ist die Simplifikation von Teilausdrücken des Programmtextes. Hierdurch sollen unnötig komplexe Ausdrücke durch logisch äquivalente, aber einfachere Ausdrücke ersetzt werden, wobei die Bedeutung dieser Ausdrucke und die algorithmische Struktur des Programms keine Rolle spielen darf. Aus logischer Sicht bestehen Simplifikationen aus einer Anwendung gerichteter Gleichungen als *Rewrite*-Regeln (z.B. durch Substitution oder Lemma-Anwendung). Dabei unterscheidet man *kontextunabhängige* und *kontextabhängige* Simplifikationen.

- Bei kontextunabhängigen (CI-) Simplifikationen wird ausschließlich der zu vereinfachende Teilausdruck

betrachtet und mit Hilfe von Rewrite-Regeln *ohne Vorbedingungen* so lange umgeschrieben, bis keine Vereinfachungsregel mehr angewandt werden kann. So wird zum Beispiel der Ausdruck $\texttt{a}\in\texttt{(x.[])}\circ\texttt{(b.L)}$ mit den Lemmata aus Appendix B.2 wie folgt schrittweise vereinfacht werden.

$$\texttt{a}\in\texttt{(x.[])}\circ\texttt{(b.L)} \mapsto \texttt{a}\in\texttt{x.([]}\circ\texttt{(b.L))} \mapsto \texttt{a}\in\texttt{x.(b.L)} \mapsto \texttt{a=x} \vee \texttt{a}\in\texttt{b.L} \mapsto \texttt{a=x} \vee \texttt{a=b} \vee \texttt{a}\in\texttt{L}$$

- Bei kontextabhängigen (CD-) Simplifikationen werden *bedingte Rewrite-Regeln* bzw. bedingte Gleichungen für die Vereinfachung eines Teilausdrucks eingesetzt. Hierbei spielt – wie der Name bereits andeutet – der Kontext des zu vereinfachenden Teilausdrucks eine wesentliche Rolle, denn hieraus ergibt sich, ob eine bedingte Gleichung wie $\texttt{a}\in\texttt{S} \wedge \texttt{p(a)} \Rightarrow \forall\texttt{x}\in\texttt{S}.\,\texttt{p(x)} \Leftrightarrow \forall\texttt{x}\in\texttt{S-a}.\,\texttt{p(x)}$ auf einen gegebenen Ausdruck überhaupt anwendbar ist.

  Der Kontext wird dabei im Voraus durch eine Analyse des abstrakten Syntaxbaumes bestimmt, in dem sich der Teilausdruck befindet, wobei Ausdrücke wie z.B. der Test einer Fallunterscheidung, benachbarte Konjunkte in einem generellen Set-Former $\{\,\texttt{f(x)} \mid \texttt{x}\in\texttt{S} \wedge \texttt{p(x)} \wedge \texttt{q(x)}\}$ oder die Vorbedingungen der WHERE-Klausel zum Kontext gehören.

  Eine der wesentlichen Aspekte der kontextabhängigen Simplifikation ist die Elimination redundanter Teilausdrücke im Programmcode wie zum Beispiel die Überprüfung einer Bedingung, die bereits in den Vorbedingungen des Programms auftaucht.

Üblicherweise wird eine Simplifikation in Kooperation mit einem Benutzer des Systems durchgeführt. Dieser gibt an, welche konkreten Teilausdrücke vereinfacht werden sollen und welcher Art die Vereinfachung sein soll. Hierdurch erspart man sich die aufwendige Suche nach simplifizierbaren Teilausdrücken, die einem Menschen beim ersten Hinsehen sofort auffallen. Wir wollen dies an unserem Leitbeispiel illustrieren.

Beispiel 5.6.1 (Costas Arrays Algorithmus: Simplifikationen)

In Beispiel 5.5.26 hatten wir den folgenden Algorithmus zur Lösung des Costas Arrays Problems durch Instantiierung eines Globalsuch-Schemas erzeugt.

$$\begin{array}{l}
\texttt{FUNCTION Costas (n:}\mathbb{Z}\texttt{):Set(Seq(}\mathbb{Z}\texttt{))} \;\; \texttt{WHERE n}\geq\texttt{1} \\
\quad \texttt{RETURNS \{p | perm(p,\{1..n\}) } \wedge\; \forall\texttt{j}\in\texttt{domain(p).nodups(dtrow(p,j))\}} \\
= \;\; \texttt{if } \boxed{\texttt{nodups([])} \wedge \forall\texttt{j}\in\texttt{domain([]).nodups(dtrow([],j))}} \\
\qquad \texttt{then Costas}_{gs}\texttt{(n,[])} \;\; \texttt{else } \emptyset
\end{array}$$

$$\begin{array}{l}
\texttt{FUNCTION Costas}_{gs}\texttt{ (n:}\mathbb{Z}\texttt{,V:Seq(}\mathbb{Z}\texttt{)):Set(Seq(}\mathbb{Z}\texttt{))} \\
\quad \texttt{WHERE n}\geq\texttt{1} \wedge \texttt{range(V)}\subseteq\texttt{\{1..n\}} \wedge \texttt{nodups(V)} \wedge \forall\texttt{j}\in\texttt{domain(V).nodups(dtrow(V,j))} \\
\quad \texttt{RETURNS \{p | perm(p,\{1..n\}) } \wedge\; \texttt{V}\sqsubseteq\texttt{p} \wedge \forall\texttt{j}\in\texttt{domain(p).nodups(dtrow(p,j))\}} \\
= \quad \texttt{\{p |} \boxed{\texttt{p}\in\texttt{\{V\}}} \wedge\; \texttt{perm(p,\{1..n\})} \wedge \forall\texttt{j}\in\texttt{domain(p).nodups(dtrow(p,j))\}} \\
\quad \cup \bigcup\{\,\texttt{Costas}_{gs}\texttt{(n,W)} \mid \boxed{\texttt{W}\in\texttt{\{V}\cdot\texttt{i}\mid\texttt{i}\in\texttt{\{1..n\}\}}} \wedge \texttt{nodups(W)} \wedge \forall\texttt{j}\in\texttt{domain(W).nodups(dtrow(W,j))}\,\}
\end{array}$$

In diesem Algorithmenpaar gibt es eine Reihe offensichtlicher Vereinfachungsmöglichkeiten, die im Programmtext umrandet wurden. So ist nach Lemma B.2.24.1 $\texttt{nodups([])}$ immer gültig[35] und ebenso $\forall\texttt{j}\in\texttt{domain([]).nodups(dtrow([],j))}$ nach Lemma B.2.21.1 und B.1.11.1. der gesamte umrandete Ausdruck vereinfacht sich somit zu true, was zur Folge hat, daß das Konditional

$$\texttt{if nodups([])} \wedge \forall\texttt{j}\in\texttt{domain([]).nodups(dtrow([],j))} \texttt{ then Costas}_{gs}\texttt{(n,[])} \texttt{ else } \emptyset$$

[35]Dieses Lemma zu finden ist nicht schwer, da der äußere Operator nodup und der innere Operator [] zusammen eindeutig bestimmen, welches Lemma überhaupt anwendbar ist. Bei einer entsprechenden Strukturierung der Wissensbank kann dieses Lemma in konstanter Zeit gefunden werden.

insgesamt zu $\texttt{Costas}_{gs}\texttt{(n,[])}$ vereinfacht werden kann. Auf ähnliche Weise lassen sich auch die anderen umrandeten Ausdrücke mit den Gesetzen endlicher Mengen und Folgen vereinfachen. Die Anwendung kontextunabhängiger Vereinfachungen führt somit zu dem folgenden vereinfachten Programmpaar.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs(n,[])

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ [nodups(V)] ∧ [[∀j∈domain(V).nodups(dtrow(V,j))]]
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
=    (if perm(V,{1..n}) ∧ ∀j∈domain(V).nodups(dtrow(V,j)) then {V} else ∅)
 ∪ ⋃{ Costas_gs(n,V·i) | i∈{1..n} ∧ [nodups(V·i)] ∧ [[∀j∈domain(V·i).nodups(dtrow(V·i,j))]] }
```

Das aufrufende Hauptprogramm kann nun nicht mehr weiter vereinfacht werden. In der Hilfsfunktion $\texttt{Costas}_{gs}$ bietet sich an vielen Stellen jedoch eine kontextabhängige Vereinfachung an, da im Programmcode eine Reihe von Bedingungen geprüft werden, die bereits in den Vorbedingungen genannt sind. Die zusammengehörigen Ausdrücke und ihre relevanten Kontexte sind durch gleichartige Markierungen gekennzichnet, wobei ein Kontext durchaus mehrfach Verwendung finden kann. So ist zum Beispiel der Ausdruck `perm(V,{1..n})` äquivalent zu `range(V)⊆{1..n} ∧ {1..n}⊆range(V) ∧ nodups(V)`. Da zwei dieser drei Klauseln bereits in den Vorbedingungen genannt sind, kann `perm(V,{1..n})` im Kontext der Vorbedingungen zu dem Ausdruck `{1..n}⊆range(V)` vereinfacht werden. Auf ähnliche Art können wir auch die anderen markierten Teilausdrücke vereinfachen und erhalten als Ergebnis kontextabhängiger Simplifikationen.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs(n,[])

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
=    (if {1..n}⊆range(V) then {V} else ∅)
 ∪ ⋃{ Costas_gs(n,V·i) | i∈{1..n}\range(V) ∧ ∀j∈domain(V). (V[|V|+1-j]-i)∉dtrow(V,j) }
```

5.6.2 Partielle Auswertung

Eine besondere Art der Vereinfachung bietet sich immer dann an, wenn in einem Ausdruck Funktionen mit einem konstantem (Teil-)Argument vorkommen. Dies legt den Versuch nahe, diese Funktion so weit wie möglich im voraus auszuwerten, also eine sogenannte *Partielle Auswertung* [Bjorner *et.al.*, 1988] oder *Spezialisierung* [Scherlis, 1981] des Programms bereits zur Entwurfszeit vorzunehmen. Dies erspart die entsprechenden Berechnungen zur Laufzeit des Programms.

Bei funktionalen Programmen besteht partielle Auswertung[36] aus dem simplen *Auffalten* einer Funktionsdefinition, gefolgt von einer Simplifikation. Dies wird so lange durchgeführt, wie konstante Teilausdrück vorhanden sind. Wir wollen dies an einem einfachen Beispiel illustrieren.

[36] Dies deckt natürlich nicht alle Möglichkeiten ab, bei denen eine partielle Auswertung möglich ist. Auch Operationen mit mehreren Argumenten, von denen nur eines konstant ist, können mit den aus der Literatur bekannten Techniken behandelt werden. Zudem ist bei imperativen Programmen die partielle Auswertung deutlich aufwendiger, da Zustände von Variablen berücksichtigt werden müssen.

Beispiel 5.6.2

Ein Teilausdruck der Form `|[4;5]∘V|`, wobei `V` ein beliebiger Programmausdruck – zum Beispiel eine Variable – ist, kann wie folgt partiell ausgewertet werden.

```
   |[4;5]∘V|                          Auffalten der Definition von concat
 = |4.([5]∘V)|                        Auffalten der Definition von concat
 = |4.(5.([]∘V))|                     Auffalten der Definition von concat
 = |4.(5.V)|                          Auffalten der Definition von length
 = 1 + |5.V|                          Auffalten der Definition von length
 = 1 + 1 + |V|                        arithmetische Simplifikation
 = 2 + |V|
```

Wie bei den Simplifikationen muß der auszuwertende Teilausdruck durch den Benutzer ausgewählt werden. In einfachen Fällen deckt sich die partielle Auswertung mit der kontextunabhängigen Simplifikation, da hier das Auffalten der Funktionsdefinition einer Lemma-Anwendung gleichkommt. Die wirklichen Stärken der partiellen Auswertung kommen daher erst bei der Behandlung neu erzeugter Funktionen zum Tragen.

5.6.3 Endliche Differenzierung

In manchen Programmen werden innerhalb einer Schleife oder einer Rekursion Teilausdrücke berechnet, deren Hauptargument die Schleifenvariable (bzw. ein Parameter des rekursiven Aufrufs) ist. Diese Berechnung kann vereinfacht werden, wenn ein Zusammenhang zwischen der Änderung der Schleifenvariablen und dem berechneten Teilausdruck festgestellt werden kann. In diesem Fall lohnt es sich nämlich, den Teilausdruck *inkrementell* zu berechnen, anstatt die Berechnung in jedem Schleifendurchlauf neu zu starten. Dies kann zu einer signifikanten Beschleunigung der Berechnung führen.

So verbraucht zum Beispiel die Berechnung von `{1..n}\range(V)` in der Hilfsfunktion $\texttt{Costas}_{gs}$ aus Beispiel 5.6.1 in etwa `|V|*n` Schritte. Berücksichtigt man jedoch, daß bei jedem rekursiven Aufruf von $\texttt{Costas}_{gs}$ die Variable `V` durch `V·i` ersetzt wird, so ließe sich die Berechnung dadurch vereinfachen, daß man eine neue Variable `Pool` einführt, welche die Menge `{1..n}\range(V)` inkrementell verwaltet. Bei einem rekursiven Aufruf braucht `Pool` dann nur entsprechend der Änderung von `V` in `Pool-i` umgewandelt werden, was erheblich schneller zu berechnen ist.

Die soeben angedeutete Technik, durch Einführung einer zusätzlichen Variablen eine inkrementelle Berechnung wiederkehrender Teilausdrücke zu ermöglichen, heißt in der Literatur *endliche Differenzierung* [Paige, 1981, Paige & Koenig, 1982], da eben nur noch die differentiellen Veränderungen während eines Rekursionsaufrufes berechnet werden müssen. Im Kontext funktionaler Programme läßt sich endliche Differenzierung in zwei elementarere Operationen zerlegen, nämlich in einer *Abstraktion über einen Teilausdruck* und eine anschließende Simplifikation.

- Die *Abstraktion eines Algorithmus* f mit der Variablen x *über einen Teilausdruck* $E[x]$ erzeugt einen neuen Algorithmus f', der zusätzlich zu den Variablen von f eine neue Variable c besitzt. Die Vorbedingung von f wird um $c{=}E[x]$ erweitert und im Funktionskörper wird jeder Aufruf der Gestalt $f(t[x])$ durch $f'(t[x],E[t[x]])$ ersetzt. In ähnlicher Weise wird auch jeder externe Aufruf der Gestalt $f(u)$ durch $f'(u,E[u])$ ersetzt.

  Eine Abstraktion transformiert also eine Menge von Programmen der Gestalt

  `FUNCTION` g... `WHERE ... RETURNS ... SUCH THAT...` $= \ldots f(u) \ldots$

  `FUNCTION` $f(x{:}D){:}R$ `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$
  $= \ldots E[x] \ldots f(t[x]) \ldots$

  in die folgende Menge von Programmen

  `FUNCTION` g... `WHERE ... RETURNS ... SUCH THAT...` $= \ldots f'(u,E[u]) \ldots$

  `FUNCTION` $f'(x{:}D,c{:}D'){:}R$ `WHERE` $I[x] \wedge c{=}E[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$
  $= \ldots E[x] \ldots f'(t[x],E[t[x]]) \ldots$

- Die anschließende Simplifikation beschränkt sich auf die Berücksichtigung der neu entstandenen Gleichung c=$E[x]$, die als Kontext der Teilausdrücke des Programmkörpers von f' hinzukommt.

  Kontextabhängige Simplifikation wird nun gezielt auf alle Teilausdrücke der Form $E[t[x]]$ angewandt, wobei vor allem versucht wird diese Teilausdrücke in die Form $t'[E[x]]$ umzuwandeln. Anschließend wird jedes Vorkommen von $E[x]$ zu c vereinfacht.

Wieder ist es der Benutzer, der den konkreten Teilausdruck $E[x]$ auswählt, mit dem endliche Differenzierung durchgeführt werden soll. Wir wollen dies an unserem Leitbeispiel illustrieren.

Beispiel 5.6.3 (Costas Arrays Algorithmus: Endliche Differenzierung)

Mit den in Beispiel 5.6.1 durchgeführten Vereinfachungen hatten wir folgende Version des Costas Arrays Algorithmus erzeugt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs(n,[])

FUNCTION Costas_gs (n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
=    (if {1..n}⊆range(V) then {V} else ∅)
 ∪ ⋃{ Costas_gs(n,V·i) | i∈{1..n}\range(V) ∧ ∀j∈domain(V). (V[|V|+1-j]-i)∉dtrow(V,j) }
```

Ein geübter Programmierer erkennt relativ schnell, daß die ständige Neuberechnung von `{1..n}\range(V)` den Algorithmus unnötig ineffizient werden läßt. Ebenso überflüssig ist die Berechnung von `|V|+1`, die sich ebenfalls nur inkrementell ändert. Beide Ausdrücke werden vom Benutzer gekennzeichnet. Nach (zweimaliger) Anwendung der endlichen Differenzierung ergibt sich dann folgendes Programmpaar.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs2(n,[],{1..n},1)

FUNCTION Costas_gs2 (n:ℤ,V:Seq(ℤ), Pool:Set(ℤ),Vsize:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
            ∧ Pool = {1..n}\range(V)        ∧ Vsize = |V|+1
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
= (if Pool=∅ then {V} else ∅)
 ∪ ⋃{ Costas_gs2(n,V·i,Pool−i,Vsize+1) | i∈Pool ∧ ∀j∈domain(V). (V[Vsize-j]-i)∉dtrow(V,j) }
```

Man beachte, daß hierbei `{1..n}⊆range(V)` in `{1..n}\range(V)=∅` umgewandelt und dann durch `Pool=∅` ersetzt wurde. Genauso könnte man in dem neuen Kontext auch `domain(V)` zu `{1..Vsize-1}` vereinfachen, was aber gezielt geschehen müsste und keinen großen Gewinn bringt.

Durch die endliche Differenzierung werden übrigens auch sinnvolle neue Datenstrukturen und Konzepte eingeführt, die im Algorithmus bisher nicht verarbeitet wurden. So ist zum Beispiel die Verwaltung eines Pools von Werten, die noch als mögliche Ergänzung der Folge `V` benutzt werden können, ein wichtiges Konzept, das auch einem menschlichen Programmierer schnell in den Sinn kommen könnte. Hier wurde es nun durch eine allgemeine Optimierungstechnik nachträglich in den Algorithmus eingefügt.

5.6.4 Fallanalyse

In Programmen, die Konditionale der Form `if` P `then` e_1 `else` e_2 vorkommen, mag es sinnvoll sein, die Fallunterscheidung mit P auf den gesamten Programmkörper auszuweiten, um so weitere kontextabhängige

Simplifikationen durchführen zu können. Durch eine solche Art der *Fallanalyse* könnte man zum Beispiel herausbekommen, daß die Funktion $\texttt{Costas}_{gs2}$ niemals aufgerufen wird, wenn `Pool=`$\emptyset$ ist, und die beiden mit $\cup$ verbundenen Teilausdrücke somit disjunkte Fälle beschreiben.

Die einfachste Technik der *Fallanalyse über einer Bedingung* P besteht darin, einen Teilausdruck E durch `if` P `then` E `else` E zu ersetzen und anschließend kontextabhängige Simplifikationen auf beiden Instanzen von E auszuführen. Auch dies wollen wir an unserem Leitbesipiel illustrieren.

Beispiel 5.6.4 (Costas Arrays Algorithmus: Fallanalyse)

Nach der endlichen Differenzierung hatte der Costas Arrays Algorithmus die folgende Form.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs2(n,[],{1..n},1)

FUNCTION Costas_gs2 (n:ℤ,V:Seq(ℤ), Pool:Set(ℤ),Vsize:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
            ∧ Pool = {1..n}\range(V)        ∧ Vsize = |V|+1
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
= (if [Pool=∅] then {V} else ∅)
 ∪ ⋃{ Costas_gs2(n,V·i,Pool−i,Vsize+1) | i∈Pool ∧ ∀j∈domain(V). (V[Vsize-j]-i)∉dtrow(V,j) }
```

Eine Fallanalyse des Programmkörpers von $\texttt{Costas}_{gs2}$ über der markierten Bedingung `Pool=`$\emptyset$ führt dazu, daß im positiven Fall der Ausdruck $\bigcup\{ \texttt{Costas}_{gs2}(\ldots.) \mid \texttt{i}\in\emptyset \wedge \ldots\}$ entsteht, der wiederum zu $\emptyset$ vereinfacht werden kann. Damit haben wir in beiden Fällen eine Vereinigung mit einer leeren Menge, was zu der folgenden vereinfachten Version des Algorithmus führt.

```
FUNCTION Costas (n:ℤ):Set(Seq(ℤ))  WHERE n≥1
  RETURNS {p | perm(p,{1..n}) ∧ ∀j∈domain(p).nodups(dtrow(p,j))}
=  Costas_gs2(n,[],{1..n},1)

FUNCTION Costas_gs2 (n:ℤ,V:Seq(ℤ), Pool:Set(ℤ),Vsize:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ ∀j∈domain(V).nodups(dtrow(V,j))
            ∧ Pool = {1..n}\range(V)        ∧ Vsize = |V|+1
  RETURNS {p | perm(p,{1..n}) ∧ V⊑p ∧ ∀j∈domain(p).nodups(dtrow(p,j)) }
= if Pool=∅
     then {V}
     else ⋃{ Costas_gs2(n,V·i,Pool−i,Vsize+1) | i∈Pool ∧ ∀j∈domain(V).(V[Vsize-j]-i)∉dtrow(V,j)}
```

5.6.5 Datentypverfeinerung

Mit den bisherigen Techniken haben wir vor allem die Möglichkeiten einer algorithmischen Optimierung synthetisierter Programme beschrieben. Darüberhinaus kann man aber noch weitere Effizienzsteigerungen dadurch erhalten, daß man für abstrakt definierte Datentypen wie Listen, Mengen, Graphen etc. und ihre Operationen eine Implementierung auswählt, die für den konkreten Algorithmus besonders geeignet ist.

So können endliche Mengen zum Beispiel durch Listen, Felder (Bitvektoren), Bäume, charakteristische Funktionen etc. implementiert werden. Listen wiederum können in vowärts oder rückwärts verketteter Form dargestellt werden. Welche dieser Implementierungen für eine konkrete Variable des Algorithmus gewählt werden sollte, hängt dabei im wesentlichen von den damit verbundenen Operationen ab. So empfiehlt sich eine rückwärts verkettete Liste immer dann, wenn viele Append-Operationen im Algorithmus vorkommen, was innerhalb von $\texttt{Costas}_{gs2}$ zum Beispiel für die Variable `V` gilt. Felder für endliche Mengen sind empfehlenswert, wenn diese – wie die Variable `Pool` – eine Maximalgröße nicht überschreiten und häufig die $\in$-Relation getestet wird.

Die Wahl einer konkreten Implementierung für ein abstrakt definiertes Konstrukt nennt man *Datentypverfeinerung* [Schonberg *et.al.*, 1981, Blaine & Goldberg, 1990a, Blaine & Goldberg, 1990b]. Diese Technik ist sehr sicher, wenn das Syntheseystem eine Reihe von Implementierungen für einen abstrakten Datentyp – sowie die eventuell nötigen Konversonsoperationen – bereithält, aus denen ein Benutzer dann separat für jede einzelne Variable eine Auswahl treffen kann. Da die richtige Wahl einer Implementierung sehr stark von einer Einschätzung der Häufigkeit bestimmter Operationen abhängt, kann sie bisher nur beschränkt automatisiert[37] werden.

5.6.6 Compilierung und sprachabhängige Optimierung

Die letzte Möglichkeit zu einer Optimierung von synthetisch erzeugten Algorithmen besteht in einer Übersetzung[38] der Algorithmen in eine konkrete Programmiersprache, die effizienter zu verarbeiten ist als die abstrakte mathematische Sprache, in der alle anderen Schritte stattgefunden haben. Dieser Schritt sollte erst dann ausgeführt werden, wenn alle anderen Möglichkeiten ausgeschöpft wurden, da eine algorithmische Optimierung nach einer Übersetzung erheblich schwerer ist. Sprachabhängige Optimierungen können im Anschluß daran durchgeführt werden, wobei man sich allerdings auf die in einen Compiler der Zielsprache eingebauten Möglichkeiten beschränken sollte.

5.7 Diskussion

Wir haben in diesem Kapitel die verschiedenen Techniken zur Erzeugung effizienter Algorithmen aus ihren formalen Spezifikationen vorgestellt und aufgezeigt, wie diese Methoden in einen allgemeinen logischen Formalismus wie der Typentheorie des NuPRL Systems integriert werden können. Damit sind wir unserem usprünglichem Ziel, durch Verfahren zur rechnergestützten Softwareentwicklung eine Antwort auf die Softwarekrise zu finden, einen deutlichen Schritt nähergekommen und mehr oder weniger bis an den derzeitigen Stand der Forschungen vorgedrungen.

Es hat sich herausgestellt, daß Programmsyntheseverfahren prinzipiell notwendig sind, um die derzeitige Praxis der Programmierung zu verbessern und im Hinblick auf die Erzeugung korrekter und wiederverwendbarer Software zu unterstützen. Trotz einiger spektakulärer Teilerfolge[39] sind Programmsynthesesysteme jedoch noch weit davon entfernt, Marktreife zu erlangen. Zu viele Fragen – vor allem im Hinblick au die Erzeugung formaler Spezifikationen und die Synthese komplerer Systeme – sind noch ungeklärt und das Gebiet des *wissensbasierten Software-Eingineering* ist mehr denn je ein zukunftsträchtiges Forschungsgebiet, das von fundamentaler Bedeutung für die Informatik ist.

Unter den bisherigen Ansätzen erscheint der Weg über algorithmische Theorien der sinnvollste zu sein, vor allem weil der die Stärken der Paradigmen "Beweise als Programme" und "Synthese durch Transformationen" auf einem hohen Abstraktionsniveau miteinander zu verbinden weiß und dem Ideal der *wissensbasierten* Programmsynthese am nächsten kommt. Bei diesem Ansatz muß allerdings der Gefahr begegnet werden,

[37] Dadurch, daß die Implementierung jedoch automatisch erstellt wird, bietet sich die Möglichkeit eines experimentellen Vorgehens an, bei dem man das Laufzeitverhalten verschiedener Implementierungen einer Variablen im konkreten Fall austestet und anhand dieser Erkenntnisse seine Wahl trifft. Die Testergebnisse können hierbei jedoch nur einen Hinweischarakter haben und eine gründliche Analyse des entstandenen Algorithmus nicht ersetzen.

[38] Die Sprache LISP ist hierfür am besten geeignet, da unsere abstrakte mathematische Sprache ebenfalls funktional ist und somit kein Paradigmenwechsel stattfinden müsste. Da LISP auf größeren Computern mittlerweile genauso schnell ist wie andere gängige Programmiersprachen, besteht auch kein Grund, eine andere Zielsprache zu wählen, solange der generierte Algorithmus nicht in ein festes Softwarepaket eingebaut werden soll. Die Übersetzung in andere Zielsprachen wird daher erst bei einem kommerziellen Einsatz von Programmsynthesesystemen relevant werden.

[39] Eine weitestgehend vom KIDS-System geleistete Neuentwicklung des Programms, mit dem z.B. die amerikanische Armee ihre Transporte bei Einsätzen in Übersee plant, konnte den bisher hierzu verwendeten Algorithmus bei realistischen Testdaten um den Faktor 2000 (!) beschleunigen (siehe [Smith & Parra, 1993]). Man vermutet, daß alleine durch diesen Einzelerfolg soviele Millionen von Dollar eingespart werden können, wie die Entwicklung von KIDS gekostet hat.

durch eine von logischen Kalkülen losgelöste "von Hand Codierung" der entsprechenden Spezialstrategien die Vorteile wieder aufs Spiel zu setzen. Im Endeffekt kann nur eine Integration dieses Ansatzes in ein formales Beweisentwicklungssystem wie NuPRL die notwendige Sicherheit und Flexibilität garantieren, die man sich von der rechnergestützten Softwareentwicklung erhofft.

Hierzu ist allerdings noch eine Menge Arbeit zu leisten. So muß zum Beispiel eine große *Wissensbank* aufgebaut werden, um eine Synthese von Algorithmen aus verschiedensten Anwendungsbereichen zu ermöglichen. Dies bedeutet nicht nur die Formalisierung und Verifikation von Wissen über die Standard-Anwendungsbereiche, mit denen sich die Informatik-Literatur beschäftigt,[40] sondern auch Techniken zur effizenten Verwendung dieses Wissens als Rewrite-Regeln und Methoden zur *effizienten Strukturierung der Wissensbank* in Teiltheorien, die einen schnellen und gezielten Zugriff auf dieses Wissen unterstützen. Auf der anderen Seite sind neben den einfachen Rewrite-Techniken, die an sich schon sehr viel zum praktischen Erfolg von Synthesesystemen beitragen, eine Reihe von *Beweisverfahren* in das allgemeine System zu integrieren. Hierzu gehören vor allem ein konstruktives prädikatenlogisches Beweisverfahren, allgemeine Induktionstechniken, und natürlich auch anwendungsspezifische Beweismethoden, welche die besonderen Denkweisen bestimmter Theorien widerspiegeln. All diese Techniken sind mehr oder weniger als Taktiken zu modellieren, wenn sie in ein einziges System integriert werden sollen. Mit einem derartigen Fundament können dann die in diesem Kapitel besprochenen *Syntheseverfahren und Optimierungstechniken* in das allgemeine System eingebettet werden. Mit der in Abschnitt 5.5.3 vorgestellten Vorgehensweise ist dies auf elegante und natürliche Art möglich: die Formalisierung der Strategien entspricht dann im wesentlichen ihrer Beschreibung auf dem Papier und benötigt keine gesonderte Codierung.

Im Hinblick auf *praktische Verwendbarkeit* müssen Programmsynthesesysteme viele einander scheinbar widersprechende Bedingungen erfüllen. So muß die interne Verarbeitung von Wissen natürlich formal korrekt sein, um die geforderte Sicherheit der erzeugten Software zu garantieren. Nach außen hin aber sollte so wenig formal wie möglich gearbeitet werden, um einem "normalen" benuzter den Umgang mit dem System zu ermöglichen. Der in Abschnitt 4.1.7.2 angesprochene Display-Mechanismus ist ein erster Ansatz, die Brücke zwischen diesen beiden Anforderungen zu bauen. Dies muß jedoch ergänzt werden um eine anschauliche graphische Unterstützung bei der Steuerung eines Syntheseprozesses, damit sich auch ungeübte Benutzer zügig in den Umgang mit einem Softwarentwicklungssystem einarbeiten können. In diesem Punkte sind wissenschaftliche Programme leider weit von dem entfernt, was man von einem modernen Benutzerinterface erwarten kann.

Möglichkeiten zur Mitarbeit an Systemen für eine rechnergestützte Softwareentwicklung gibt es also mehr als genug. Dies setzt allerdings ein gewisses Interesse für eine Verknüpfung von theoretische Grundlagen und praktischen Programmierarbeiten voraus, also Formale Denkweise, Kenntnis logischer Kalküle und Abstraktionsvermögen auf der einen Seite sowie Kreativität, Experimentierfreudigkeit und Ausdauer auf der anderen. Da Forschungstätigkeiten immer auch an die Grenzen des derzeit Machbaren und der eigenen Fähigkeiten heranführen, Unzulänglichkeiten der zur Verfügung stehenden Hilfsmittel schnell offenbar werden und sich Fortschritte nicht immer so schnell einstellen, wie man dies gerne hätte, ist auch ein gehöriges Maß von Frustrationstoleranz eine wesentliche Voraussetzung für einen Erfolg.

[40] Man müsste mindestens 500 Definitionen und 5000 Lemmata aufstellen, um ein Wissen darzustellen, was dem eines Informatik-Studenten nach dem Vordiplom gleichkommt.

Literaturverzeichnis

[Ackermann, 1954] W. Ackermann. *Solvable cases of the decision problem.* North–Holland, 1954.

[Aczel, 1978] Peter Aczel. The type theoretic interpretation of constructive set theory. In A. MacIntyre, L. Pacholski, and J. Paris, editors, *Logic Colloquium '77*, pages 55–66. North–Holland, 1978.

[Allen *et.al.*, 1990] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In John C. Mitchell, editor, *LICS-90 – Fifth Annual Symposium on Logic in Computer Science, 1990, Philadelphia, PA, June 1990*, pages 95–106. IEEE Computer Society Press, 1990.

[Allen, 1987a] Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In David Gries, editor, *LICS-87 – Second Annual Symposium on Logic in Computer Science, Ithaca, New York, USA, June 1987*, pages 215–224. IEEE Computer Society Press, 1987.

[Allen, 1987b] Stuart Allen. *A non-type-theoretic semantics for type-theoretic language.* PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, December 1987. TR 87-866.

[Altenkirch *et.al.*, 1994] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A user's guide to ALF.* University of Göteborg, 1994.

[Amadio & Cardelli, 1990] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. Technical Report 62, DIGITAL Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, August 1990.

[Anderson, 1970] R. Anderson. Completeness results for E-resolution. In *Spring Joint Computer Conference*, volume 36, pages 636–656. AFIPS Press, 1970.

[Andrews, 1971] Peter B. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36(3):414–432, September 1971.

[Andrews, 1986] Peter B. Andrews. *An Introduction to mathematical logic and Type Theory: To Truth through Proof.* Academic Press, Orlando, 1986.

[Backhouse *et.al.*, 1988a] Roland C. Backhouse, P. Chisholm, and Grant Malcolm. Do it yourself type theory (part I). *EATCS Bulletin*, 34:68–110, 1988. Also in *Formal aspects of computing* 1:19-84, 1989.

[Backhouse *et.al.*, 1988b] Roland C. Backhouse, P. Chisholm, and Grant Malcolm. Do it yourself type theory (part II). *EATCS Bulletin*, 35:205–245, 1988.

[Backhouse, 1984] Roland C. Backhouse. A note on subtypes in Martin-Löf's Theory of Types. Technical Report CSM-70, Computer Science Department, University of Essex, England, November 1984.

[Backhouse, 1985] Roland C. Backhouse. Algorithm development in Martin-Löf's Type Theory. Technical report, Computer Science Department, University of Essex, England, 1985.

[Backhouse, 1989] Roland C. Backhouse. Constructive Type Theory - an introduction. In Manfred Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series, Series F: Computer & System Sciences*, pages 9–62. Springer Verlag, 1989.

[Balzer *et.al.*, 1983] Robert Balzer, T. E. Cheatham jr, and Cordell Green. Software technology in the 1990's: Using a new paradigm. *Computer*, pages 39–45, November 1983.

[Balzer, 1985] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[Barendregt, 1981] Henk P. Barendregt. *The Lambda Calculus. Its syntax and semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North–Holland, 1981. (revised version 1984).

[Barstow, 1979] David R. Barstow. *Knowledge-Based Program Construction*. Elsevier Science Publishers B.V., New York, 1979.

[Basin, 1989] David A. Basin. Building theories in NuPRL. In A. R. Meyer and M. A. Taitslin, editors, *Logic at Botik 89*, number 363 in Lecture Notes in Computer Science, pages 12–25. Springer Verlag, 1989.

[Bates & Constable, 1985] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[Bates, 1981] Joseph L. Bates. *A logic for correct Program Development*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, 1981.

[Bauer & others, 1988] F. L. Bauer *et al. The Munich Project CIP: Volume II*, volume 292 of *Lecture Notes in Computer Science*. Springer Verlag, 1988.

[Bibel & Hörnig, 1984] Wolfgang Bibel and K. M. Hörnig. LOPS - a system based on a strategical approach to program synthesis. In Alan W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic program construction techniques*, chapter 3, pages 69–89. MacMillan, New York, 1984.

[Bibel *et.al.*, 1978] Wolfgang Bibel, Ulrich Furbach, and J. F. Schreiber. Strategies for the synthesis of algorithms. In K. Alber, editor, *Proceedings 5th GI Conference on Programming Languages*, number 12 in Informatik Fachberichte, pages 97–109, Berlin, 1978. Springer Verlag.

[Bibel *et.al.*, 1994] W. Bibel, S. Brüning, U. Egly, and T. Rath. Komet. In Alan Bundy, editor, *12th Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 783–787. Springer Verlag, 1994. System description.

[Bibel, 1978] Wolfgang Bibel. On strategies for the synthesis of algorithms. In D. Sleeman, editor, *Proceedings of AISB/GI Conference on Artificial Intelligence, Leeds University, England*, pages 22–27, 1978.

[Bibel, 1980] Wolfgang Bibel. Syntax–directed, semantics–supported program synthesis. *Artificial Intelligence*, 14(3):243–261, October 1980.

[Bibel, 1983] Wolfgang Bibel. Matings in matrices. *Communications of the Association for Computing Machinery*, 26(11):844–852, November 1983.

[Bibel, 1987] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg Verlag, second edition, 1987.

[Bibel, 1991] Wolfgang Bibel. Toward predicative programming. In Michael R. Lowry and Robert McCartney, editors, *Automating Software Design*, pages 405–424, Menlo Park, CA, 1991. AAAI Press / The MIT Press.

[Bibel, 1992] Wolfgang Bibel. *Deduktion – Automatisierung der Logik*, volume 6.2 of *Handbuch der Informatik*. R. Oldenbourg, München, Wien, 1992.

[Bishop, 1967] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967. revised and extended version: [**?**].

[Bjorner *et.al.*, 1988] D. Bjorner, A.P. Ershov, and N.D.Jones, editors. *Partial evaluation and mixed computation*. North–Holland, 1988.

[Blaine & Goldberg, 1990a] Lee Blaine and Allen Goldberg. DTRE—a semi-automatic transformation system. Technical report, Kestrel Institute, October 1990.

[Blaine & Goldberg, 1990b] Lee Blaine and Allen Goldberg. Verifiably correct data type refinement. Technical report, Kestrel Institute, November 1990.

[Bläsius *et.al.*, 1981] K. Bläsius, N. Eisinger, J. Siekmann, G. Smolka, A. Herold, and C. Walther. The Markgraf Karl refutation procedure. In *IJCAI-81 – 7th International Joint Conference on Artificial Intelligence, Vancouver, BC, Canada, August 1981*, pages 511–518, Los Altos CA, 1981. Morgan Kaufmann.

[Bledsoe, 1977] W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9(1):1–35, 1977.

[Boyer & Moore, 1979] Robert S. Boyer and J. Strother Moore. *A computational Logic*. Academic Press, New York, 1979.

[Bridges, 1979] Douglas Bridges. *Constructive Functional Analysis*. Pitman, 1979.

[Brouwer, 1908a] L. E. J. Brouwer. De onbetrouwbaarheid der logische principes. *Tijdschrift voor wijsbegeerte*, 2:152–158, 1908. English translation: "The unreliability of the logical principles", in [Brouwer, 1975], pp. 107–112.

[Brouwer, 1908b] L. E. J. Brouwer. Over de grondslagen der wiskunde. *Nieuw Arch. Wisk*, 2(8):326–328, 1908. English translation: "On the foundations of mathematics", in [Brouwer, 1975], pp. 105–107.

[Brouwer, 1924a] L. E. J. Brouwer. Intuitionistische Zerlegung mathematischer Grundbegriffe. *Jahresbericht Deutscher Mathematiker Verein*, 33:251–256, 1924.

[Brouwer, 1924b] L. E. J. Brouwer. Über die Bedeutung des Satzes vom ausgeschlossenen Dritten in der Mathematik, insbesondere in der Funktionentheorie. *J. reine und angewandte Mathematik*, pages 1–7, 1924.

[Brouwer, 1925] L. E. J. Brouwer. Zur Begründung der intuitionistischen Mathematik I. *Mathematische Annalen*, 93:244–257, 1925.

[Brouwer, 1926] L. E. J. Brouwer. Zur Begründung der intuitionistischen Mathematik II. *Mathematische Annalen*, 95:453–472, 1926.

[Brouwer, 1927] L. E. J. Brouwer. Zur Begründung der intuitionistischen Mathematik III. *Mathematische Annalen*, 96:451–488, 1927.

[Brouwer, 1975] L. E. J. Brouwer. *Collected Works*, volume 1. North–Holland, 1975.

[Broy & Pepper, 1981] Manfred Broy and Peter Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, SE-7(1):14–22, January 1981.

[Bruijn, 1980] N. G. De Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York, 1980.

[Bundy *et.al.*, 1990] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 647–648. Springer Verlag, 1990.

[Bundy, 1989] Alan Bundy. Automatic guidance of program synthesis proofs. In *Workshop on Automating Software Design, IJCAI-89, Kestrel Institute, Palo Alto*, pages 57–59, 1989.

[Burstall & Darlington, 1977] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[Cardone & Coppo, 1990] F. Cardone and M. Coppo. Two extendsions of Curry's type inference system. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 19–76. Academic Press, 1990.

[Chisholm, 1985] Paul Chisholm. Derivation of a parsing algorithm in Martin-Löf's theory of types. Technical report, Department of Computer Science, Heriot-Watt University, Edinburgh, Scotland, 1985.

[Chisholm, 1987] Paul Chisholm. Derivation of a parsing algorithm in Martin-Löf's Type Theory. *Science of Computer Programming*, 8:1–42, 1987.

[Chou & Gao, 1990] Shang-Ching Chou and Xiao-Shan Gao. Ritt-Wu's decomposition algorithm and geometry theorem proving. In M. E. Stickel, editor, *10^{th} Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 207–220. Springer Verlag, 1990.

[Church & Rosser, 1936] Alonzo Church and B. J. Rosser. Some properties of conversion. *Trans. Am. Math. Soc.*, 39:472–482, 1936.

[Church, 1940] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Clark & Sickel, 1977] Keith L. Clark and S. Sickel. Predicate logic: A calculus for the formal derivation of programs. In Raj Reddy, editor, *IJCAI-77 – 5^{th} International Joint Conference on Artificial Intelligence, Cambridge, MA, August 1977*, pages 419–420. Morgan Kaufman, Los Altos, CA, 1977.

[Cleaveland, 1987] Walter Rance Cleaveland. *Type-theoretic models of concurrency.* PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, 1987. TR 87-837.

[Constable & Howe, 1990a] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R. B. Banerjee, editor, *Formal techniques in Artificial Intelligence, a sourcebook*, pages 45–75. Elsevier Science Publishers B.V., 1990.

[Constable & Howe, 1990b] Robert L. Constable and Douglas J. Howe. Nuprl as a general logic. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 77–90. Academic Press, 1990.

[Constable & Mendler, 1985] Robert L. Constable and Nax Paul Mendler. Recursive definitions in Type Theory. In Rohit Parikh, editor, *Logics of Programs Conference, Brooklyn, NY, USA, June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer Verlag, 1985.

[Constable & Smith, 1987] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive Type Theory. In David Gries, editor, *LICS-87 – Second Annual Symposium on Logic in Computer Science, Ithaca, New York, USA, June 1987*, pages 183–193. IEEE Computer Society Press, 1987.

[Constable & Smith, 1988] Robert L. Constable and Scott Fraser Smith. Computational foundations of basic recursive function theory. In Yuri Gurevich, editor, *LICS-88 – Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland, July 1988*, pages 360–371. IEEE Computer Society Press, 1988.

[Constable & Smith, 1993] Robert L. Constable and Scott Fraser Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121:89–112, 1993.

[Constable *et.al.*, 1982] Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science.* Springer Verlag, 1982.

[Constable *et.al.*, 1985] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.

[Constable *et.al.*, 1986] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.

[Constable, 1983] Robert L. Constable. Programs as proofs. *Information Processing Letters*, 16(3):105–112, 1983.

[Constable, 1984] Robert L. Constable. Mathematics as programming. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs*, number 164 in Lecture Notes in Computer Science, pages 116–128. Springer Verlag, 1984.

[Constable, 1985] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. *Annals of Discrete Mathematics*, 24:21–38, 1985.

[Constable, 1988] Robert L. Constable. Themes in the development of programming logics circa 1963-1987. *Annual Reviews in Computer Science*, pages 147–165, 1988.

[Constable, 1989] Robert L. Constable. Assigning meaning to proofs: a semantic basis for problem solving environments. In Manfred Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series, Series F: Computer & System Sciences*, pages 63–94. Springer Verlag, 1989.

[Coquand & Huet, 1985] Thierry Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, pages 151–184, Linz, Austria, April 1985.

[Coquand & Huet, 1988] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Coquand & Paulin, 1988] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic, Tallin, USSR, December 1988*, number 417 in Lecture Notes in Computer Science, pages 50–66. Springer Verlag, 1988.

[Coquand, 1990] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.

[Costas, 1984] J. Costas. A study of a class of detection waveforms having nearly ideal range – doppler amibiguity properties. In *Proceedings of the IEEE*, volume 72, pages 996–1009, 1984.

[Cousineau & Huet, 1990] Guy Cousineau and Gerard Huet. The CAML primer. Rapports Techniques 122, Institut National de Recherche en Informatique et en Automatique, September 1990.

[Curry *et.al.*, 1958] Haskell B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume 1. North–Holland, 1958.

[Curry, 1970] Haskell B. Curry. *Outline of a formalist philosophy of mathematics.* North–Holland, 1970.

[Dahl *et.al.*, 1972] O.-J. Dahl, Edsger W. Dijkstra, and C.A.R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing.* Academic Press, 1972.

[Damas & Milner, 1982] Luis Damas and Robin Milner. Principal type schemes for functional programs. In 9^{th} *ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Darlington, 1975] John Darlington. Application of program transformation to program synthesis. In *IRIA Symposium on Proving and Improving programs*, pages 133–144, Arc-et-Senans, France, 1975.

[Davis & Putnam, 1960] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[de la Tour & Kreitz, 1992] Thierry Boy de la Tour and Christoph Kreitz. Building proofs by analogy via the Curry-Howard isomorphism. In A. Voronkov, editor, *LPAR'92, Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 202–213. Springer Verlag, 1992.

[Dershowitz, 1985] Nachum Dershowitz. Synthesis by completion. In A. Joshi, editor, *IJCAI-85 – 9th International Joint Conference on Artificial Intelligence, Los Angeles, August 1985*, pages 208–214, 1985.

[Dijkstra, 1976] Edsger W. Dijkstra. *A discipline of Programming.* Prentice Hall, 1976.

[Dummett, 1977] Michael Dummett. *Elements of Intuitionism.* Oxford Logic Series. Clarenden Press, Oxford, 1977.

[Felty & Miller, 1988] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 61–80. Springer Verlag, 1988.

[Felty & Miller, 1990] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In M. E. Stickel, editor, *10th Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 221–235. Springer Verlag, 1990.

[Felty, 1991] Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 215–251. Cambridge University Press, 1991.

[Franova & Kodratoff, 1991] Marta Franova and Yves Kodratoff. Solving "How to clear a block" with constructive matching methodology. In John Mylopoulos and Ray Reiter, editors, *IJCAI-91 – 12th International Joint Conference on Artificial Intelligence, Sidney, August 1991*, pages 232–237. Morgan Kaufmann Publishers, 1991.

[Franova, 1985] Marta Franova. A methodology for automatic programming based on the constructive matching strategy. In *EUROCAL 85*, pages 568–570. Springer Verlag, 1985.

[Frege, 1879] Gottlob Frege. *Begriffsschrift.* Verlag Louis Nebert, Halle, 1879.

[Frege, 1892] Gottlob Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, 100:25–50, 1892.

[Freyd, 1990] Peter Freyd. Recursive types reduced to inductive types. In John C. Mitchell, editor, *LICS-90 – Fifth Annual Symposium on Logic in Computer Science, 1990, Philadelphia, PA, June 1990*, pages 498–507. IEEE Computer Society Press, 1990.

[Fribourg, 1990] L. Fribourg. Extracting logic programs from proofs that use extended PROLOG execution and induction. In *7th International Conference on Logic Programming, Jerusalem*, pages 685–699. MIT Press, 1990.

[Gallier, 1986] Jean H. Gallier. *Logic for Computer Science: Foundations of Automated Theorem Proving.* Harper and Row, New York, 1986.

[Galmiche, 1990] Didier Galmiche. Constructive system for automatic program synthesis. *Theoretical Computer Science*, 71:227–239, 1990.

[Gentzen, 1935] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English Translation: "Investigations into logical deduction" in [**?**] pages 68-131.

[Girard *et.al.*, 1989] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[Girard, 1971] Jean-Yves Girard. Une extension de l'interpretation de godel a l'analyse, et son application a l'elimination des coupures dans l'analyse at la theorie des types. In J. Fenstad, editor, *second Scandinavian Logic Symposium*, pages 63–92. North–Holland, 1971.

[Girard, 1972] Jean-Yves Girard. *Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur*. PhD thesis, Universite Paris VII, 1972.

[Girard, 1986] Jean-Yves Girard. The system F of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[Girard, 1987] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Giunchiglia & Smaill, 1989] Fausto Giunchiglia and Alan Smaill. Reflection in constructive and non-constructive automated reasoning. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 6, pages 123–140. MIT Press, Cambridge Mass., 1989.

[Gordon *et.al.*, 1979] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[Gordon., 1985] M. Gordon. HOL: A machine oriented formalization of higher order logic. Technical Report 68, Cambridge University, 1985.

[Gordon., 1987] M. Gordon. HOL: A proof generating system for higher order logic. Technical Report 103, Cambridge University, 1987.

[Green, 1969] Cordell C. Green. An application of theorem proving to problem solving. In *IJCAI-69 – 1^{st} International Joint Conference on Artificial Intelligence, Washington, DC, May 1969*, pages 219–239, 1969.

[Gries, 1981] David Gries. *The science of programming*. Springer Verlag, 1981.

[Guttag & Horning, 1978] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, pages 27–52, 1978.

[Harper & Pollack, 1989] Robert Harper and Robert Pollack. Type checking, universe polymorphism and typical ambiguity in the calculus of constructions. In J. Diaz and F. Orejas, editors, *TAPSOFT '89 - International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 1989*, number 352 in Lecture Notes in Computer Science, pages 241–256. Springer Verlag, 1989.

[Harper, 1985] Robert W. Harper. *Aspects of the implementation of Type Theory*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, April 1985. TR 85-675.

[Hayashi, 1986] S. Hayashi. PX: a system extracting programs from proofs. In *IFIP Confereence on Formal Description of Programming Concepts*, pages 399–424, 1986.

[Hayes, 1987] I. Hayes. *Specification Case Studies*. Prentice-Hall, 1987.

[Heisel *et.al.*, 1988] M. Heisel, W. Reif, and W. Stephan. Implementing verification strategies in the KIV system. In E. Lusk and R. Overbeek, editors, *9^{th} Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 131–140. Springer Verlag, 1988.

[Heisel *et.al.*, 1990] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In M. E. Stickel, editor, *10^{th} Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 117–131. Springer Verlag, 1990.

[Heisel *et.al.*, 1991] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Formal software development in the KIV system. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 547–576, Menlo Park, CA, 1991. AAAI Press / The MIT Press.

[Heisel, 1989] Maritta Heisel. A formalization and implementation of Gries's program development method with the KIV environment. Technical Report 3/89, Fakultät für Informatik, Universität Karlsruhe, 1989.

[Heyting, 1931] Arend Heyting. Die intuitionistische Grundlegung der Mathematik. *Erkenntnis*, 2:106–115, 1931.

[Heyting, 1934] A. Heyting. *Mathematische Grundlagenforschung: Intuitionismus – Beweistheorie*. Springer Verlag, 1934.

[Heyting, 1971] Arend Heyting. *Intuitionism: An Introduction*. North–Holland, 3^{rd} edition, 1971.

[Hilbert & Bernays, 1934] David Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 1. Springer Verlag, 1934.

[Hilbert & Bernays, 1939] David Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 2. Springer Verlag, 1939.

[Hindley & Seldin, 1986] J. Roger Hindley and Jonathan P. Seldin. *Introduction to combinators and λ-calculus*. Cambridge University Press, 1986.

[Hindley, 1969] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Hindley, 1983] J. Roger Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[Hoare, 1972] Charles Antony Richard Hoare. Notes on data structuring. In *[Dahl* et.al.*, 1972]*, pages 83–174. Academic Press, 1972.

[Hoare, 1975] C. A. R. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, June 1975.

[Hogger, 1978] C. J. Hogger. Goal-oriented derivation of logic programs. In *7th Symposium on the Mathematical foundations of Computer Science*, Zakopane, Poland, September 3–8 1978. Polish Academy of Science.

[Hogger, 1981] C. J. Hogger. Derivation of logic programs. *Journal of the Association for Computing Machinery*, 28(2):372–392, April 1981.

[Horn, 1988] Christian Horn. Interactive program and proof development with NurPRL. Seminarbericht 97, Sektion Mathematik der Humboldt-Universität zu Berlin, PSF 1297, 1086 Berlin, 1988.

[Howard, 1980] W. Howard. The formulas-as-types notion of construction. In J. Roger Hindley and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, New York, 1980. First published 1969.

[Howe, 1986] Douglas J. Howe. Implementing number theory: An experiment with NuPRL. In J. H. Siekmann, editor, 8^{th} *Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer Verlag, 1986.

[Howe, 1987] Douglas J. Howe. The computational behaviour of girard's paradox. In David Gries, editor, *LICS-87 – Second Annual Symposium on Logic in Computer Science, Ithaca, New York, USA, June 1987*, pages 205–214. IEEE Computer Society Press, 1987.

[Howe, 1988] Douglas J. Howe. *Automating reasoning in an implementation of constructive Type Theory.* PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, June 1988. TR 88-925.

[Huet, 1986] Gérard Huet. Deduction and computation. In Wolfgang Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence — An Advanced Course*, volume 232 of *Lecture Notes in Computer Science*, pages 39–74. Springer Verlag, 1986.

[Jackson, 1993a] Paul Jackson. *NuPRL ML Manual.* Cornell University. Department of Computer Science, 1993.

[Jackson, 1993b] Paul Jackson. *The Nuprl Proof Development System, Version 4.1: Introductory Tutorial.* Cornell University. Department of Computer Science, 1993.

[Jackson, 1993c] Paul Jackson. *The Nuprl Proof Development System, Version 4.1: Reference Manual and User's Guide.* Cornell University. Department of Computer Science, 1993.

[Johnson & Feather, 1991] W. Lewis Johnson and Martin S. Feather. Using evolution transformations to construct specifications. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 65–92, Menlo Park, CA, 1991. AAAI Press / The MIT Press.

[Johnson, 1983] Scott D. Johnson. *A Computer System for checking Proofs.* UMI Press, New York, 1983. Ph.D. Thesis.

[Jones & Shaw, 1990] C. B. Jones and R. C. Shaw. *Case Studies in Systematic Software Development.* Prentice-Hall, 1990.

[Kant & Barstow, 1981] Elaine Kant and David Barstow. The refinement paradigm: the interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, 7(5):458–471, 1981.

[Kelly & Nonnenmann, 1991] Van E. Kelly and Uwe Nonnenmann. Reducing the complexity of formal specification acquisition. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 41–64, Menlo Park, CA, 1991. AAAI Press / The MIT Press.

[Kneale & Kneale, 1962] William Kneale and Martha Kneale. *The Development of Logic.* Clarendon Press, Oxford, 1962.

[Knoblock, 1987] Todd B. Knoblock. *Metamathematical extensibility in Type Theory.* PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, December 1987. TR 87-892.

[Knuth, 1968] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley, 1968.

[Knuth, 1972] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison Wesley, 1972.

[Knuth, 1975] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley, 1975.

[Kolmogorov, 1932] A. N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

[Kreitz, 1986] Christoph Kreitz. Constructive automata theory implemented with the NuPRL proof development system. Technical Report TR 86-779, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, August 1986.

[Kreitz, 1992] Christoph Kreitz. *METASYNTHESIS: Deriving Programs that Develop Programs.* Thesis for Habilitation, Technische Universität Darmstadt, FG Intellektik, 1992. Forschungsbericht AIDA–93–03.

[Krieg-Brückner *et.al.*, 1986] B. Krieg-Brückner, B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möricke, B. Weisgerber, A.D. McGettrick, I.G. Campbell, and G. Winterstein. PROgram development by SPECification and TRAnsformation. In *Proceedings ESPRIT Conference*, Brussels, 1986.

[Krieg-Brückner, 1989] B. Krieg-Brückner. Algebraic specifications and functionals for transformational program and meta-program development. In J. Diaz and F. Orejas, editors, *TAPSOFT '89 - International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 1989*, number 352 in Lecture Notes in Computer Science, pages 36–59. Springer Verlag, 1989.

[Lafontaine, 1990] Christine Lafontaine. Formalization of the VDM reification in the DEVA meta-calculus. the human-leucocyte-antigen case study. In *IFIP Working Conference on Programming Concepts and Methods*, 1990.

[Lakatos, 1976] Imre Lakatos. *Proofs and Refutations.* Cambridge University Press, Cambridge, 1976.

[Lau & Prestwich, 1990] K.K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In *7th International Conference on Logic Programming*, pages 667–684. MIT Press, 1990.

[Leivant, 1990] Daniel Leivant. Contracting proofs to programs. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 279–328. Academic Press, 1990.

[Letz *et.al.*, 1992] Reinhold Letz, Johann Schumann, Stephan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[Lowry & Duran, 1989] Michael R. Lowry and R. Duran. Knowledge-based software engineering. In Avron Barr, Paul R. Cohen, and Edward A. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Vol. IV*, chapter XX. Addison Wesley, 1989.

[Lowry, 1987] Michael R. Lowry. Algorithm synthesis through problem reformulation. In *AAAI-87 – 6^{th} AAAI National Conference on Artificial Intelligence, Seattle, July 1987*, Seattle, WA, July 13–17, 1987. Technical Report KES.U.87.10, Kestrel Institute, August 1987.

[Lowry, 1988] Michael R. Lowry. The structure and design of local search algorithms. In M. R. Lowry, R. McCartney, and D. R. Smith, editors, *Workshop on Automating Software Design. AAAI-88, St. Paul, MN, August 25, 1988*, pages 88–94, August 1988.

[Lowry, 1989] Michael R. Lowry. *Algorithm Synthesis through Problem Reformulation.* PhD thesis, Stanford University, 1989.

[Lowry, 1991] Michael R. Lowry. Automating the design of local search algorithms. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 515–546, Menlo Park, CA, 1991. AAAI Press / The MIT Press.

[Luo & Pollack, 1992] Zhaohui Luo and Robert Pollack. *The LEGO Proof Development System: A user's manual.* University of Edinburgh, 1992. LFCS Report ECS-LFVS-92-211.

[Luo, 1989] Zhaohui Luo. ECC: an extended calculus of constructions. In Rohit Parikh, editor, *LICS-89 – Fourth Annual Symposium on Logic in Computer Science, 1989*, pages 385–395. IEEE Computer Society Press, 1989.

[Luo, 1990] Zhaohui Luo. *An extended calculus of constructions.* PhD thesis, University of Edinburgh, 1990.

[Manna & Waldinger, 1975] Zohar Manna and Richard J. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6(2):175–208, 1975.

[Manna & Waldinger, 1979] Zohar Manna and Richard J. Waldinger. Synthesis: Dreams ⇒ programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, July 1979.

[Manna & Waldinger, 1980] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[Manna & Waldinger, 1985] Zohar Manna and Richard J. Waldinger. *The logical basis for computer programming*, volume I: Deductive Reasoning. Addison Wesley, 1985.

[Martin-Löf, 1970] Per Martin-Löf. A theory of types. unpublished manuscript, 1970.

[Martin-Löf, 1973] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, Amsterdam, 1973. North–Holland.

[Martin-Löf, 1982] Per Martin-Löf. Constructive mathematics and computer programming. In *6-th International Congress for Logic, Methodology and Philosophy of Science, 1979*, pages 153–175. North–Holland, 1982.

[Martin-Löf, 1984] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes*. Bibliopolis, Napoli, 1984.

[Martin-Löf, 1988] Per Martin-Löf. Mathematics of infinity. In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic, Tallin, USSR, December 1988*, number 417 in Lecture Notes in Computer Science, pages 146–197. Springer Verlag, 1988.

[Mauny, 1991] Michel Mauny. Functional programming using CAML. Technical Report 129, Institut National de Recherche en Informatique et en Automatique, May 1991.

[Mendler *et.al.*, 1986] Paul Francis Mendler, Prakash Panangaden, and Robert L. Constable. Infinite objects in Type Theory. Technical Report TR 86-743, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, March 1986.

[Mendler, 1987a] Paul Francis Mendler. *Inductive definition in Type Theory*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, September 1987. TR 87-870.

[Mendler, 1987b] Paul Francis Mendler. Recursive types and type constraints in second order lambda calculus. In David Gries, editor, *LICS-87 – Second Annual Symposium on Logic in Computer Science, Ithaca, New York, USA, June 1987*, pages 30–36. IEEE Computer Society Press, 1987.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Minsky, 1963] M. Minsky. Steps toward artificial intelligence. *Computers and Thought*, pages 406–450, 1963.

[Morris, 1969] J. B. Morris. E-resolution: an extension of resolution to include the equality relation. In *IJCAI-69 – 1st International Joint Conference on Artificial Intelligence, Washington, DC, May 1969*, pages 287–294. Morgan Kaufmann, 1969.

[Nelson & Oppen, 1979] Greg Nelson and Derek. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[Nelson & Oppen, 1980] Greg Nelson and Derek. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[Nerode, 1988] A. Nerode. Some lectures on intuitionistic logic. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, volume 1429 of *Lecture Notes in Mathematics*, pages 12–59. Springer Verlag, 1988.

[Neugebauer *et.al.*, 1989] Gerd Neugebauer, Bertram Fronhöfer, and Christoph Kreitz. XPRTS - an implementation tool for program synthesis. In D. Metzing, editor, *GWAI-89 – 13th German Workshop on Artificial Intelligence*, volume 216 of *Informatik Fachberichte*, pages 348–357. Springer Verlag, 1989.

[Newell *et.al.*, 1963] A. Newell, M. Shaw, and H. Simon. Empirical explorations with the logic theory machine. *Computers and Thought*, pages 109–133, 1963.

[Nordström & Smith, 1984] Bengt Nordström and Jan M. Smith. Propositions and specifications of programs in Martin-Löfs Type Theory. *BIT*, 24:288–301, 1984.

[Nordström *et.al.*, 1990] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löfs Type Theory. An introduction.* Clarendon Press, Oxford, 1990.

[Paige & Koenig, 1982] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[Paige, 1981] Robert A. Paige. *Formal Differentiation - A Program Synthesis Technique.* Artificial Intelligence, No. 6. UMI Research Press, Ann Arbor, MI, October 1981.

[Partsch & R., 1983] H. Partsch and Steinbrüggen R. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[Paulin-Mohring, 1989] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *16th ACM Symposium on Principles of Programming Languages*, pages 89–104, 1989.

[Paulson, 1986] Lawrence C. Paulson. Constructing recursion operators in intuituionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[Paulson, 1987] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[Paulson, 1989] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Paulson, 1990] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pitts, 1989] Andrew M. Pitts. Non-trivial power types can't be subtypes of polymorphic types. In Rohit Parikh, editor, *LICS-89 – Fourth Annual Symposium on Logic in Computer Science, 1989*, pages 6–13. IEEE Computer Society Press, 1989.

[Pollack, 1994] Robert Pollack. *The theory of LEGO – a proof checker for the extendend calculus of constructions.* PhD thesis, University of Edinburgh, 1994.

[Polya, 1945] G. Polya. *How to solve it.* Princeton University Press, Princeton, New Jersey, 1945.

[Prawitz, 1960] Dag Prawitz. An improved proof procedure. *Theoria*, 26:102–139, 1960.

[Prawitz, 1965] Dag Prawitz. *Natural Deduction: A Proof-Theoretical study.* Almquist & Wiksell, 1965.

[Quine, 1963] Willard Van Orman Quine. *Set Theory and its logic.* Belknap Press, Cambridge, MA, 1963.

[Reps & Teitelbaum, 1984] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings ACM SIGPLAN SIGOA Symposium on Practical Software Development Environments*, April 1984.

[Reynolds, 1981] John C. Reynolds. *The craft of programming.* Prentice Hall, Englewood Cliffs, N.J., 1981.

[Richter, 1978] Michel M. Richter. *Logikkalküle*, volume 43 of *Teubner Studienbücher Informatik.* B.G.Teubner, Stuttgart, 1978.

[Robinson & Wos, 1969] G. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. *Machine Intelligence*, 4:135–150, 1969.

[Robinson, 1965] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[Rogers, 1967] Hartley jr. Rogers. *Theory of recursive functions and effective computability.* MIT Press, Cambridge, MA, 1967.

[Russel, 1908] Bertrand Russel. Mathematical logic as based on a theory of types. *Am. J. of Math.*, 30:222–262, 1908.

[Salvesen & Smith, 1988] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's Type Theory. In Yuri Gurevich, editor, *LICS-88 – Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland, July 1988*, pages 384–391. IEEE Computer Society Press, 1988.

[Sannella & Tarlecki, 1987] Donald Sannella and Andrzej Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[Sannella & Tarlecki, 1988] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25(3):233–281, 1988.

[Sasaki, 1986] James T. Sasaki. *Extracting Efficient Code From Constructive Proofs.* PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, June 1986.

[Sato & Tamaki, 1989] T. Sato and H. Tamaki. First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8:605–627, 1989.

[Scherlis, 1981] William Scherlis. Program improvement by internal specialization. In 8^{th} *ACM Symposium on Principles of Programming Languages*, pages 41–49. Association for Computing Machinery, 1981.

[Schonberg *et.al.*, 1981] Edward Schonberg, Jacob Schwartz, and M. Sharir. An automatic technique for the selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.

[Schütte, 1977] Kurt Schütte. *Proof Theory.* Springer Verlag, 1977.

[Schwartzbach, 1986] Michael Ignatieff Schwartzbach. *A category theoretic analysis of predicative Type Theory.* PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, December 1986. TR 86-793.

[Scott, 1972] Dana Scott. Lattice theory, data types, and semantics. In R. Rustin, editor, *Formal semantics of Programming Languages*, pages 65–106. Prentice Hall, 1972.

[Scott, 1976] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–287, 1976.

[Silverman *et.al.*, 1988] J. Silverman, V. Vickers, and J. Mooney. On the number of Costas arrays as a function of array size. In *Proceedings of the IEEE*, volume 76, pages 851–853, 1988.

[Smith & Lowry, 1990] Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, October 1990. Report KES.U.89.3, Kestrel Institute.

[Smith & Parra, 1993] Douglas R. Smith and Eduardo A. Parra. Transformational approach to transportation scheduling. In *8th Knowledge-Based Software Engineering Conference, Chicago, Il, September 1993*, pages 60–68, 1993.

[Smith, 1982] Douglas R. Smith. Top-down synthesis of simple divide-and-conquer algorithms. Technical Report NPS52-82-011, Naval Postgraduate School, Monterey, California, 1982.

[Smith, 1983a] Douglas R. Smith. The structure of divide-and-conquer algorithms. Technical Report NPS52-83-002, Naval Postgraduate School, Monterey, California, March 1983.

[Smith, 1983b] Jan M. Smith. The identification of propositions and types in Martin-Löf's Type Theory: A programming example. In *International Conference on Foundations of Computation Theory, Borgholm, Sweden, August 1983*, pages 445–456. Springer Verlag, 1983.

[Smith, 1984] Jan M. Smith. An interpretation of Martin-Löf's Type Theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[Smith, 1985a] Douglas R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[Smith, 1985b] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, September 1985. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[Smith, 1987a] Douglas R. Smith. Application of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, June 1987.

[Smith, 1987b] Douglas R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, November 1987. Revised Version, July 1988.

[Smith, 1988] Scott Fraser Smith. *Partial objects in Type Theory*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, August 1988. TR 88-938.

[Smith, 1990] Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990. Special Issue on Formal Methods extended version: [**?**].

[Smith, 1991a] Douglas R. Smith. KIDS — a knowledge-based software development system. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 483–514, Menlo Park, CA, 1991. AAAI Press / The MIT Press.

[Smith, 1991b] Douglas R. Smith. Structure and design of dynamic programming algorithms. Technical report, Kestrel Institute, 1991.

[Smith, 1991c] Douglas R. Smith. Structure and design of problem reduction generators. In B. Möller, editor, *IFIP TC2 Working Conference on Constructing Programs from Specifications*, pages 91–124. Elsevier Science Publishers B.V., 1991.

[Smith, 1992] Douglas R. Smith. Constructing specification morphisms. Technical Report KES.U.92.1, Kestrel Institute, April 1992.

[Smith, 1993] Douglas R. Smith. Classification approach to design. Technical Report KES.U.93.4, Kestrel Institute, 1993.

[Stansifer, 1985] Ryan Stansifer. *Representing Constructive theories in High-level Programming languages*. PhD thesis, Cornell University. Department of Computer Science, Ithaca, NY 14853-7501, March 1985. TR 85-664.

[Stenlund, 1972] Sören Stenlund. *Combinators, λ-terms and Proof Theory.* D. Reidel, Dordrecht, The Netherlands, 1972.

[Suppes, 1972] Patrick Suppes. *Axiomatic Set Theory.* Dover Publications, 1972.

[Tait, 1967] William W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

[Takeuti, 1975] Gaisi Takeuti. *Proof Theory.* North–Holland, 1975.

[Teitelbaum & Reps, 1981] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A syntax-directed environment. *Communications of the Association for Computing Machinery*, 24(9), September 1981.

[Troelsta, 1969] Ann S. Troelsta. *Principles of Intuitionism.* Springer Verlag, 1969.

[Troelsta, 1977] Ann S. Troelsta. Aspects of constructive mathematics. In Jon Barwise, editor, *Handbook of mathematical logic*, chapter D.5, pages 974–1053. North–Holland, 1977.

[Turner, 1984] R. Turner. *Logics for Artificial Intelligence.* Ellis Horwood Ltd., 1984.

[van Benthem Jutting, 1977] L. S. van Benthem Jutting. *Checking Landau's Grundlagen in the AUTOMATH System.* PhD thesis, Eindhoven University of Technology, Eindhoven, Netherlands, March 1977. also: Mathematical Centre Tracts 83, Math. Centre, Amsterdam, 1979.

[van Dalen, 1986] Dirk van Dalen. Intuitionistic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, chapter III.4, pages 224–339. D. Reidel, 1986.

[van Heijenoort, 1967] J. van Heijenoort. *From Frege to Gödel: A Sourcebook of Mathematical Logic.* Harvard University Press, Cambridge, Massachusetts, 1967.

[Wallen, 1990] Lincoln Wallen. *Automated deduction in nonclassical logics.* MIT Press, 1990.

[Weber, 1990] Matthias Weber. Formalization of the Bird-Meertens algorithmic calculus in the DEVA meta-calculus. In *IFIP Working Conference on Programming Concepts and Methods*, 1990.

[Weis *et.al.*, 1990] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny, and Acsander Suarez. The CAML reference manual. Rapports Techniques 121, Institut National de Recherche en Informatique et en Automatique, September 1990.

[Whitehead & Russell, 1925] Alan N. Whitehead and Bertrand Russell. *Principia Mathematicae*, volume 1. Cambridge University Press, Cambridge, MA, 1925.

[Wirth, 1971] N. Wirth. Program development by stepwise refinement. *Communications of the Association for Computing Machinery*, 14(4):221–227, April 1971.

[Wos *et.al.*, 1984] Larry Wos, Russ Overbeek, Ewing Lusk, and J. Boyle. *Automated Reasoning.* Prentice Hall, 1984.

[Wos *et.al.*, 1990] L. Wos, S. Winker, W. McCune, R. Overbeek, E. Lusk, R. Stevens, and R. Butler. Automated reasoning contributes to mathematics and logic. In M. E. Stickel, editor, *10^{th} Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 485–499. Springer Verlag, 1990.

[Wu, 1986] Wen Tsün Wu. Basic principles of mechanical theorem proving in geometries. *Journal of Automated Reasoning*, 2(4):221–251, 1986.

Anhang A

Typentheorie: Syntax, Semantik, Inferenzregeln

A.1 Parametertypen

`variable` : Variablennamen (ML Datentyp `var`)
Zulässige Elemente sind Zeichenketten der Form $[\texttt{a}-\texttt{z}\,\texttt{A}-\texttt{Z}\,\texttt{0}-\texttt{9}\,\texttt{\_}\,\texttt{-}\,\texttt{\%}]^{+}$

`natural` : Natürliche Zahlen einschließlich der Null (ML Datentyp `int`).
Zulässige Elemente sind Zeichenketten der Form $\texttt{0} + [\texttt{1} - \texttt{9}][\texttt{0} - \texttt{9}]^{*}$.

`token` : Zeichenketten für Namen (ML Datentyp `tok`).
Zulässige Elemente bestehen aus allen Symbolen des NuPRL Zeichensatzes mit Ausnahme der Kontrollsymbole.

`string` : Zeichenketten für Texte (ML Datentyp `string`).
Zulässige Elemente bestehen aus allen Symbolen des NuPRL Zeichensatzes mit Ausnahme der Kontrollsymbole.

`level-expression` : Ausdrücke für das Level eines Typuniversums (ML Datentyp `level_exp`)
Die genaue Syntax wird in Abschnitt 3.2.3.1 bei der Diskussion der Universenhierarchie besprochen.

Die Namen für Parametertypen dürfen durch ihre ersten Buchstaben abgekürzt werden.

A.2 Operatorentabelle

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
	var{x:v}()	
	x	
int{}()	**natnum**{n:n}()	**ind**{}($\boxed{u}$; x, f_x.s; $base$; y, f_y.t)
	minus{}(**natnum**{n:n}())	**minus**{}($\boxed{u}$), **add**{}($\boxed{u}$;$\boxed{v}$), **sub**{}($\boxed{u}$;$\boxed{v}$)
		mul{}($\boxed{u}$;$\boxed{v}$), **div**{}($\boxed{u}$;$\boxed{v}$), **rem**{}($\boxed{u}$;$\boxed{v}$)
		int_eq($\boxed{u}$; $\boxed{v}$; s; t), **less**($\boxed{u}$; $\boxed{v}$; s; t)
$\mathbb{Z}$	n	ind($\boxed{u}$; x, f_x.s; $base$; y, f_y.t)
	$-n$	$-\boxed{u}$, $\boxed{u}+\boxed{v}$, $\boxed{u}-\boxed{v}$
		$\boxed{u}*\boxed{v}$, $\boxed{u}\div\boxed{v}$, $\boxed{u}$ rem $\boxed{v}$
		if $\boxed{u}=\boxed{v}$ then s else t, if $\boxed{u}<\boxed{v}$ then s else t
lt{}(u;v)	**Axiom**{}()	
$u<v$	Axiom	
void{}()		**any**{}(e)
void		any(e)
Atom{}()	**token**{$string$:t}()	**atom_eq**($\boxed{u}$; $\boxed{v}$; s; t)
Atom	¨$string$¨	if $\boxed{u}=\boxed{v}$ then s else t
U{j:l}()	(alle kanonischen Typen)	
U_j		
fun{}(S; x.T)	**lam**{}(x.t)	**apply**{}($\boxed{f}$;t)
x:$S\to T$	$\lambda x.t$	$\boxed{f}$ t
prod{}(S; x.T)	**pair**{}(s;t)	**spread**{}($\boxed{e}$; x,y.u)
x:$S\times T$	$\langle s,t\rangle$	let $\langle x,y\rangle = \boxed{e}$ in u
union{}(S;T)	**inl**{}(s), **inr**{}(t)	**decide**{}($\boxed{e}$; x.u; y.v)
$S+T$	inl(s), inr(t)	case $\boxed{e}$ of inl(x) $\mapsto u$ | inr(y) $\mapsto v$
equal{}(s;t;T)	**Axiom**{}()	
$s=t\in T$	Axiom	
list{}(T)	**nil**{}(), **cons**{}(t;l)	**list_ind**{}($\boxed{s}$; $base$; x,l,f_{xl}.t)
T list	[], $t.l$	list_ind($\boxed{s}$; $base$; x,l,f_{xl}.t)
set{}(S; x.T)	(Elemente von S)	
$\{x:S \mid T\}$		
quotient{}(T; x,y.E)	(Elemente von T)	
x,$y:T//E$		
rec{}(X.T_X)	(Elemente gemäß T_X)	**rec_ind**{}($\boxed{e}$; f,x.t)
rectype $X=T_X$		let* $f(x)=t$ in f($\boxed{e}$)
pfun{}(S; T)	**fix**{}(f,x.t)	**dom**{}($\boxed{f}$), **apply_p**{}($\boxed{f}$;t)
$S\not\to T$	letrec $f(x)=t$	dom($\boxed{f}$), $\boxed{f}$(t)

Man beachte, daß in der Implementierung von NuPRL 4.0 die konkreten Namen und Displayformen zum Teil etwas anders sind als hier angegeben. An der Anpassung wird gearbeitet.

A.3 Redizes und Kontrakta

Redex		Kontraktum
$(\lambda x\,.\,u)\ t$	$\xrightarrow{\beta}$	$u[t/x]$
let $\langle x\,,y\rangle = \langle s\,,t\rangle$ in u	$\xrightarrow{\beta}$	$u[s,t\,/\,x,y]$
case inl(s) of inl$(x)\mapsto u$ | inr$(y)\mapsto v$	$\xrightarrow{\beta}$	$u[s/x]$
case inr(t) of inl$(x)\mapsto u$ | inr$(y)\mapsto v$	$\xrightarrow{\beta}$	$v[t/y]$
ind$(0\,;\,x\,,f_x\,.\,s\,;\,base\,;\,y\,,f_y\,.\,t)$	$\xrightarrow{\beta}$	$base$
ind$(n\,;\,x\,,f_x\,.\,s\,;\,base\,;\,y\,,f_y\,.\,t)$	$\xrightarrow{\beta}$	$t[n,$ind$(n\text{-}1\,;\,x\,,f_x\,.\,s\,;\,base\,;\,y\,,f_y\,.\,t)\ /x,f_x]\quad,\quad (n>0)$
ind$(\text{-}n\,;\,x\,,f_x\,.\,s\,;\,base\,;\,y\,,f_y\,.\,t)$	$\xrightarrow{\beta}$	$s[\text{-}n,$ind$(\text{-}n\text{+}1\,;\,x\,,f_x\,.\,s\,;\,base\,;\,y\,,f_y\,.\,t)\ /y,f_y],\quad (n>0)$
$-i$	$\xrightarrow{\beta}$	*Die Negation von i (als Zahl)*
$i\text{+}j$	$\xrightarrow{\beta}$	*Die Summe von i und j*
$i\text{-}j$	$\xrightarrow{\beta}$	*Die Differenz von i und j*
$i\text{*}j$	$\xrightarrow{\beta}$	*Das Produkt von i und j*
$i\div j$	$\xrightarrow{\beta}$	0*, falls j=0; ansonsten die Integer-Division von i und j*
i rem j	$\xrightarrow{\beta}$	0*, falls j=0; ansonsten der Rest der Division von i und j*
if $i\text{=}j$ then s else t	$\xrightarrow{\beta}$	*s, falls $i=j$; ansonsten t*
if $i\text{<}j$ then s else t	$\xrightarrow{\beta}$	*s, falls $i<j$; ansonsten t*
list_ind$(\,[\,]\,;\,base\,;\,x\,,l\,,f_{xl}\,.\,t)$	$\xrightarrow{\beta}$	$base$
list_ind$(s\,.\,u\,;\,base\,;\,x\,,l\,,f_{xl}\,.\,t)$	$\xrightarrow{\beta}$	$t[s,u,$list_ind$(u\,;\,base\,;\,x\,,l\,,f_{xl}\,.\,t)\ /x,l,f_{xl}]$
if $u\text{=}v$ then s else t	$\xrightarrow{\beta}$	*s, falls $u=v$; ansonsten t*
let* $f(x)=t$ in $f(e)$	$\xrightarrow{\beta}$	$t[\lambda y\,.$ let* $f(x)=t$ in $f(y),\ e\ /\ f,x]$
(letrec $f(x)=t$) (u)	$\xrightarrow{\beta}$	$t[$letrec $f(x)=t,\ u\ /\ f,x]$
dom(letrec $f(x)=t$)	$\xrightarrow{\beta}$	$\lambda x\,.$ rectype $F=\mathcal{E}[[t]]$

A.4 Urteile

A.4.1 Typsemantik

$\mathbb{Z} = \mathbb{Z}$		
$i_1 < j_1 = i_2 < j_2$	falls	$i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$
$\mathsf{void} = \mathsf{void}$ $\mathsf{Atom} = \mathsf{Atom}$		
$\mathsf{U}_{j_1} = \mathsf{U}_{j_2}$	falls	$j_1 = j_2$ *(als natürliche Zahl)*
$x_1{:}S_1 \to T_1 = x_2{:}S_2 \to T_2$	falls	$S_1 = S_2$ und $T_1[s_1/x_1] = T_2[s_2/x_2]$ für alle Terme s_1, s_2 mit $s_1 = s_2 \in S_1$.
$T = S_2 \to T_2$	falls	$T = x_2{:}S_2 \to T_2$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1 \to T_1 = T$	falls	$x_1{:}S_1 \to T_1 = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1{:}S_1 \times T_1 = x_2{:}S_2 \times T_2$	falls	$S_1 = S_2$ und $T_1[s_1/x_1] = T_2[s_2/x_2]$ für alle Terme s_1, s_2 mit $s_1 = s_2 \in S_1$.
$T = S_2 \times T_2$	falls	$T = x_2{:}S_2 \times T_2$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1 \times T_1 = T$	falls	$x_1{:}S_1 \times T_1 = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$S_1 + T_1 = S_2 + T_2$	falls	$S_1 = S_2$ und $T_1 = T_2$.
$s_1 = t_1 \in T_1 = s_2 = t_2 \in T_2$	falls	$T_1 = T_2$ und $s_1 = s_2 \in T_1$ und $t_1 = t_2 \in T_1$.
$T_1\ \mathsf{list} = T_2\ \mathsf{list}$	falls	$T_1 = T_2$
$\{x_1 : S_1 \mid T_1\} = \{x_2 : S_2 \mid T_2\}$	falls	$S_1 = S_2$ und es gibt eine Variable x, die weder in T_1 noch in T_2 vorkommt, und zwei Terme p_1 und p_2 mit der Eigenschaft $p_1 \in \forall x{:}S_1\,.\,T_1[x/x_1] \Rightarrow T_2[x/x_2]$ und $p_2 \in \forall x{:}S_1\,.\,T_2[x/x_2] \Rightarrow T_1[x/x_1]$.
$T = \{S_2 \mid T_2\}$	falls	$T = \{x_2 : S_2 \mid T_2\}$ für ein beliebiges $x_2 \in \mathcal{V}$.
$\{S_1 \mid T_1\} = T$	falls	$\{x_1 : S_1 \mid T_1\} = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1, y_1 : T_1 // E_1 = x_2, y_2 : T_2 // E_2$	falls	$T_1 = T_2$ und es gibt (verschiedene) Variablen x, y, z, die weder in E_1 noch in E_2 vorkommen, und Terme p_1, p_2, r,s und t mit der Eigenschaft $p_1 \in \forall x{:}T_1\,.\,\forall y{:}T_1\,.\,E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ und $p_2 \in \forall x{:}T_1\,.\,\forall y{:}T_1\,.\,E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ und $r \in \forall x{:}T_1\,.\,E_1[x,x/x_1,y_1]$ und $s \in \forall x{:}T_1\,.\,\forall y{:}T_1\,.\,E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$ und $t \in \forall x{:}T_1\,.\,\forall y{:}T_1\,.\,\forall z{:}T_1\,.$ $E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$
$\mathsf{rectype}\ X_1 = T_{X1} = \mathsf{rectype}\ X_2 = T_{X2}$	falls	$T_{X1}[X/X_1] = T_{X2}[X/X_2]$ für alle Typen X
$S_1 \not\to T_1 = S_2 \not\to T_2$	falls	$S_1 = S_2$ und $T_1 = T_2$

A.4.2 Elementsemantik – Universen

Elementsemantik – Universen		
$\mathbb{Z} = \mathbb{Z} \in \mathsf{U}_j$		
$i_1{<}j_1 = i_2{<}j_2 \in \mathsf{U}_j$	falls	$i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$
$\mathsf{void} = \mathsf{void} \in \mathsf{U}_j$ $\mathsf{Atom} = \mathsf{Atom} \in \mathsf{U}_j$		
$\mathsf{U}_{j_1} = \mathsf{U}_{j_2} \in \mathsf{U}_j$	falls	$j_1{=}j_2{<}j$ *(als natürliche Zahl)*
$x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2] \in \mathsf{U}_j$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\to}T_2 \in \mathsf{U}_j$	falls	$T = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\to}T_1 = T \in \mathsf{U}_j$	falls	$x_1{:}S_1{\to}T_1 = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2] \in \mathsf{U}_j$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\times}T_2 \in \mathsf{U}_j$	falls	$T = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\times}T_1 = T \in \mathsf{U}_j$	falls	$x_1{:}S_1{\times}T_1 = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$S_1{+}T_1 = S_2{+}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1{=}T_2 \in \mathsf{U}_j$.
$s_1 = t_1 \in T_1 = s_2 = t_2 \in T_2 \in \mathsf{U}_j$	falls	$T_1{=}T_2 \in \mathsf{U}_j$ und $s_1{=}s_2 \in T_1$ und $t_1{=}t_2 \in T_1$.
$T_1\,\mathsf{list} = T_2\,\mathsf{list} \in \mathsf{U}_j$	falls	$T_1 = T_2 \in \mathsf{U}_j$
$\{x_1 : S_1 \mid T_1\} = \{x_2 : S_2 \mid T_2\} \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s/x_1] \in \mathsf{U}_j$ sowie $T_2[s/x_2] \in \mathsf{U}_j$ gilt für alle Terme s mit $s \in S_1$ und es gibt eine Variable x, die weder in T_1 noch in T_2 vorkommt, und zwei Terme p_1 und p_2 mit der Eigenschaft $p_1 \in \forall x{:}S_1\,.\,T_1[x/x_1] \Rightarrow T_2[x/x_2]$ und $p_2 \in \forall x{:}S_1\,.\,T_2[x/x_2] \Rightarrow T_1[x/x_1]$
$T = \{S_2 \mid T_2\} \in \mathsf{U}_j$	falls	$T = \{x_2 : S_2 \mid T_2\} \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$\{S_1 \mid T_1\} = T \in \mathsf{U}_j$	falls	$\{x_1 : S_1 \mid T_1\} = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1, y_1 : T_1 // E_1 = x_2, y_2 : T_2 // E_2 \in \mathsf{U}_j$	falls	$T_1 = T_2 \in \mathsf{U}_j$ und für alle Terme s, t mit $s \in T_1$ und $t \in T_1$ gilt $E_1[s,t/x_1,y_1] \in \mathsf{U}_j$ sowie $E_2[s,t/x_2,y_2] \in \mathsf{U}_j$ und es gibt (verschiedene) Variablen x, y, z, die weder in E_1 noch in E_2 vorkommen, und Terme p_1, p_2, r,s und t mit der Eigenschaft $p_1 \in \forall x{:}T_1\,.\forall y{:}T_1\,.\,E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ und $p_2 \in \forall x{:}T_1\,.\forall y{:}T_1\,.\,E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ und $r \in \forall x{:}T_1\,.\,E_1[x,x/x_1,y_1]$ und $s \in \forall x{:}T_1\,.\forall y{:}T_1\,.\,E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$ und $t \in \forall x{:}T_1\,.\forall y{:}T_1\,.\forall z{:}T_1\,.$ $\quad E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$
$\mathsf{rectype}\ X_1 = T_{X1} = \mathsf{rectype}\ X_2 = T_{X2} \in \mathsf{U}_j$	falls	$T_{X1}[X/X_1] = T_{X2}[X/X_2] \in \mathsf{U}_j$ für alle Terme X mit $X \in \mathsf{U}_j$
$S_1 \not\to T_1 = S_2 \not\to T_2 \in \mathsf{U}_j$	falls	$S_1 = S_2 \in \mathsf{U}_j$ und $T_1 = T_2 \in \mathsf{U}_j$

A.4.3 Elementsemantik

$i = i \in \mathbb{Z}$		
$\mathsf{Axiom} = \mathsf{Axiom} \in s{<}t$	falls	es ganze Zahlen i und j gibt, wobei i kleiner als j ist, für die gilt $s \xrightarrow{l} i$ und $t \xrightarrow{l} j$
$s = t \in \mathsf{void}$		*gilt niemals !*
$\text{¨}string\text{¨} = \text{¨}string\text{¨} \in \mathsf{Atom}$		
$\lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$	falls	$x{:}S{\to}T$ Typ und $t_1[s_1/x_1] = t_2[s_2/x_2] \in T[s_1/x]$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S$.
$\langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S{\times}T$	falls	$x{:}S{\times}T$ Typ und $s_1{=}s_2 \in S$ und $t_1{=}t_2 \in T[s_1/x]$.
$\mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S{+}T$	falls	$S{+}T$ Typ und $s_1{=}s_2 \in S$.
$\mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S{+}T$	falls	$S{+}T$ Typ und $t_1{=}t_2 \in T$.
$\mathsf{Axiom} = \mathsf{Axiom} \in s = t \in T$	falls	$s{=}t \in T$
$[] = [] \in T\,\mathsf{list}$	falls	T Typ
$t_1.l_1 = t_2.l_2 \in T\,\mathsf{list}$	falls	T Typ und $t_1 = t_2 \in T$ und $l_1 = l_2 \in T\,\mathsf{list}$
$s = t \in \{x{:}S \mid T\}$	falls	$\{x{:}S \mid T\}$ Typ und $s = t \in S$ und es gibt einen Term p mit der Eigenschaft $p \in T[s/x]$
$s = t \in x, y : T//E$	falls	$x, y : T//E$ Typ und $s \in T$ und $t \in T$ und es gibt einen Term p mit der Eigenschaft $p \in E[s,t/x,y]$
$s = t \in \mathsf{rectype}\ X = T_X$	falls	$\mathsf{rectype}\ X = T_X$ Typ und $s = t \in T_X[\mathsf{rectype}\ X = T_X\ /\ X]$
$\mathsf{letrec}\ f_1(x_1) = t_1 = \mathsf{letrec}\ f_2(x_2) = t_2 \in S{\not\to}T$	falls	$S{\not\to}T$ Typ und für alle Terme s_1 und s_2 mit $s_1{=}s_2 \in S$ $\{x{:}S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1) = t_1)(x)\}$ $= \{x{:}S \mid \mathsf{dom}(\mathsf{letrec}\ f_2(x_2) = t_2)(x)\}$ und $t_1[\mathsf{letrec}\ f_1(x_1) = t_1,\ s_1\ /\ f_1, x_1]$ $= t_2[\mathsf{letrec}\ f_2(x_2) = t_2,\ s_2\ /\ f_2, x_2] \in T$

A.5 Inferenzregeln

A.5.1 Ganze Zahlen

$\Gamma \vdash \mathbb{Z} \in \mathsf{U}_j$ [Ax]
by intEq

$\Gamma \vdash n \in \mathbb{Z}$ [Ax]
by natnumEq

$\Gamma \vdash \mathsf{ind}(u_1; x_1, f_{x1}.s_1; base_1; y_1, f_{y_1}.t_1) = \mathsf{ind}(u_2; x_2, f_{x2}.s_2; base_2; y_2, f_{y_2}.t_2) \in T[u_1/z]$ [Ax]
by indEq z T
$\Gamma \vdash u_1 = u_2$ [Ax]
$\Gamma, x{:}\mathbb{Z}, v{:}\, x{<}0, f_x{:}T[(x{+}1)/z] \vdash s_1[x, f_x / x_1, f_{x1}] = s_2[x, f_x / x_2, f_{x2}] \in T[x/z]$ [Ax]
$\Gamma \vdash base_1 = base_2 \in T[0/z]$ [Ax]
$\Gamma, x{:}\mathbb{Z}, v{:}\, 0{<}x, f_x{:}T[(x{-}1)/z] \vdash t_1[x, f_x / y_1, f_{y_1}] = t_2[x, f_x / y_2, f_{y_2}] \in T[x/z]$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** n]
by natnumI n

$\Gamma, z{:}\mathbb{Z}, \Delta \vdash C$
[**ext** $\mathsf{ind}(z; x, f_x.s[\mathsf{Axiom}/v]; base; x, f_x.t[\mathsf{Axiom}/v])$]
by intE i
$\Gamma, z{:}\mathbb{Z}, \Delta, x{:}\mathbb{Z}, v{:}\, x{<}0, f_x{:}C[(x{+}1)/z] \vdash C[x/z]$ [**ext** s]
$\Gamma, z{:}\mathbb{Z}, \Delta \vdash C[0/z]$ [**ext** $base$]
$\Gamma, z{:}\mathbb{Z}, \Delta, x{:}\mathbb{Z}, v{:}\, 0{<}x, f_x{:}C[(x{-}1)/z] \vdash C[x/z]$ [**ext** t]

$\Gamma \vdash -s_1 = -s_2$ [Ax]
by minusEq
$\Gamma \vdash s_1 = s_2$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** $-s$]
by minusI
$\Gamma \vdash \mathbb{Z}$ [**ext** s]

$\Gamma \vdash s_1{+}t_1 = s_2{+}t_2$ [Ax]
by addEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** $s{+}t$]
by addI
$\Gamma \vdash \mathbb{Z}$ [**ext** s]
$\Gamma \vdash \mathbb{Z}$ [**ext** t]

$\Gamma \vdash s_1{-}t_1 = s_2{-}t_2$ [Ax]
by subEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** $s{-}t$]
by subI
$\Gamma \vdash \mathbb{Z}$ [**ext** s]
$\Gamma \vdash \mathbb{Z}$ [**ext** t]

$\Gamma \vdash s_1{*}t_1 = s_2{*}t_2$ [Ax]
by mulEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** $s{*}t$]
by mulI
$\Gamma \vdash \mathbb{Z}$ [**ext** s]
$\Gamma \vdash \mathbb{Z}$ [**ext** t]

$\Gamma \vdash s_1 \div t_1 = s_2 \div t_2$ [Ax]
by divEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]
$\Gamma \vdash t_1 \neq 0$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** $s \div t$]
by divI
$\Gamma \vdash \mathbb{Z}$ [**ext** s]
$\Gamma \vdash \mathbb{Z}$ [**ext** t]

$\Gamma \vdash s_1 \,\mathsf{rem}\, t_1 = s_2 \,\mathsf{rem}\, t_2$ [Ax]
by remEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]
$\Gamma \vdash t_1 \neq 0$ [Ax]

$\Gamma \vdash \mathbb{Z}$ [**ext** $s \,\mathsf{rem}\, t$]
by remI
$\Gamma \vdash \mathbb{Z}$ [**ext** s]
$\Gamma \vdash \mathbb{Z}$ [**ext** t]

$\Gamma \vdash \mathsf{if}\ u_1{=}v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ t_1 = \mathsf{if}\ u_2{=}v_2\ \mathsf{then}\ s_2\ \mathsf{else}\ t_2 \in T$ [Ax]
by int_eqEq
$\Gamma \vdash u_1 = u_2$ [Ax]
$\Gamma \vdash v_1 = v_2$ [Ax]
$\Gamma, v{:}\, u_1{=}v_1 \vdash s_1 = s_2 \in T$ [Ax]
$\Gamma, v{:}\, u_1{\neq}v_1 \vdash t_1 = t_2 \in T$ [Ax]

$\Gamma \vdash \mathsf{if}\ u_1{<}v_1\ \mathsf{then}\ s\ \mathsf{else}\ t = \mathsf{if}\ u_2{<}v_2\ \mathsf{then}\ s_2\ \mathsf{else}\ t_2 \in T$ [Ax]
by lessEq
$\Gamma \vdash u_1 = u_2$ [Ax]
$\Gamma \vdash v_1 = v_2$ [Ax]
$\Gamma, v{:}\, u_1{<}v_1 \vdash s_1 = s_2 \in T$ [Ax]
$\Gamma, v{:}\, u_1{\geq}v_1 \vdash t_1 = t_2 \in T$ [Ax]

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; \mathit{base}; y, f_y.t) = t_2 \in T$ [Ax]
by indRedDown
$\Gamma \vdash t[i,\mathsf{ind}(i+1; x, f_x.s; \mathit{base}; y, f_y.t)\ /x, f_x] = t_2 \in T$ [Ax]
$\Gamma \vdash i<0$ [Ax]

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; \mathit{base}; y, f_y.t) = t_2 \in T$ [Ax]
by indRedUp
$\Gamma \vdash t[i,\mathsf{ind}(i-1; x, f_x.s; \mathit{base}; y, f_y.t)\ /y, f_y] = t_2 \in T$ [Ax]
$\Gamma \vdash 0<i$ [Ax]

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; \mathit{base}; y, f_y.t) = t_2 \in T$ [Ax]
by indRedBase
$\Gamma \vdash \mathit{base} = t_2 \in T$ [Ax]
$\Gamma \vdash i=0$ [Ax]

$\Gamma \vdash \mathsf{if}\ u=v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by int_eqRedT
$\Gamma \vdash s = t_2 \in T$ [Ax]
$\Gamma \vdash u=v$ [Ax]

$\Gamma \vdash \mathsf{if}\ u=v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by int_eqRedF
$\Gamma \vdash t = t_2 \in T$ [Ax]
$\Gamma \vdash u\neq v$ [Ax]

$\Gamma \vdash \mathsf{if}\ u<v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by lessRedT
$\Gamma \vdash s = t_2 \in T$ [Ax]
$\Gamma \vdash u<v$ [Ax]

$\Gamma \vdash \mathsf{if}\ u<v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ [Ax]
by lessRedF
$\Gamma \vdash t = t_2 \in T$ [Ax]
$\Gamma \vdash u\geq v$ [Ax]

$\Gamma \vdash 0 \leq s \mathsf{\ rem\ } t \ \wedge\ s \mathsf{\ rem\ } t < t$ [Ax]
by remBounds1
$\Gamma \vdash 0 \leq s$ [Ax]
$\Gamma \vdash 0 < t$ [Ax]

$\Gamma \vdash 0 \leq s \mathsf{\ rem\ } t \ \wedge\ s \mathsf{\ rem\ } t < -t$ [Ax]
by remBounds2
$\Gamma \vdash 0 \leq s$ [Ax]
$\Gamma \vdash t < 0$ [Ax]

$\Gamma \vdash s \mathsf{\ rem\ } t \leq 0 \ \wedge\ s \mathsf{\ rem\ } t > -t$ [Ax]
by remBounds3
$\Gamma \vdash s \leq 0$ [Ax]
$\Gamma \vdash 0 < t$ [Ax]

$\Gamma \vdash s \mathsf{\ rem\ } t \leq 0 \ \wedge\ s \mathsf{\ rem\ } t > t$ [Ax]
by remBounds4
$\Gamma \vdash s \leq 0$ [Ax]
$\Gamma \vdash t < 0$ [Ax]

$\Gamma \vdash s = (s \div t) * t + (s \mathsf{\ rem\ } t)$ [Ax]
by divremSum
$\Gamma \vdash s \in \mathbb{Z}$ [Ax]
$\Gamma \vdash t \neq 0$ [Ax]

$\Gamma \vdash s_1<t_1 = s_2<t_2 \in \mathsf{U}_j$ [Ax]
by ltEq
$\Gamma \vdash s_1 = s_2$ [Ax]
$\Gamma \vdash t_1 = t_2$ [Ax]

$\Gamma \vdash \mathsf{Axiom} \in s<t$ [Ax]
by axiomEq_lt
$\Gamma \vdash s<t$ [Ax]

A.5.2 Void

$\Gamma \vdash \mathsf{void} \in \mathsf{U}_j$ [Ax]
by voidEq

$\Gamma \vdash \mathsf{any}(s) = \mathsf{any}(t) \in T$ [Ax]
by anyEq
$\Gamma \vdash s = t \in \mathsf{void}$ [Ax]

$\Gamma,\ z\colon \mathsf{void},\ \Delta \vdash C$ [**ext** $\mathsf{any}(z)$]
by voidE i

A.5.3 Universen

$\Gamma \vdash \mathsf{U}_j \in \mathsf{U}_k$ [Ax]
by univEq *

$\Gamma \vdash T \in \mathsf{U}_k$ [Ax]
by cumulativity j *
$\Gamma \vdash T \in \mathsf{U}_j$ [Ax]

*: *Die Regel ist nur anwendbar, wenn $j < k$ ist.*

A.5.4 Atom

$\Gamma \vdash$ Atom = Atom $\in \mathsf{U}_j$ ⌊Ax⌋
by atomEq

$\Gamma \vdash$ ¨*string*¨ $\in$ Atom ⌊Ax⌋
by tokEq

$\Gamma \vdash$ Atom ⌊**ext** ¨*string*¨⌋
by tokI ¨*string*¨

$\Gamma \vdash$ if u_1=v_1 then s_1 else t_1
= if u_2=v_2 then s_2 else t_2 $\in T$ ⌊Ax⌋
by atom_eqEq
$\Gamma \vdash u_1$=u_2 $\in$ Atom ⌊Ax⌋
$\Gamma \vdash v_1$=v_2 $\in$ Atom ⌊Ax⌋
Γ, v: u_1=v_1 $\in$ Atom $\vdash s_1 = s_2 \in T$ ⌊Ax⌋
Γ, v: $\neg(u_1$=$v_1 \in$ Atom$) \vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash$ if u=v then s else $t = t_2 \in T$ ⌊Ax⌋
by atom_eqRedT
$\Gamma \vdash s = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash u$=$v \in$ Atom ⌊Ax⌋

$\Gamma \vdash$ if u=v then s else $t = t_2 \in T$ ⌊Ax⌋
by atom_eqRedF
$\Gamma \vdash t = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash \neg(u$=$v \in$ Atom$)$ ⌊Ax⌋

A.5.5 Funktionenraum

$\Gamma \vdash x_1$:$S_1 \to T_1 = x_2$:$S_2 \to T_2 \in \mathsf{U}_j$ ⌊Ax⌋
by functionEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ ⌊Ax⌋
Γ, x:$S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x$:$S \to T$ ⌊Ax⌋
by lambdaEq j
Γ, x':$S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ ⌊Ax⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash x$:$S \to T$ ⌊**ext** $\lambda x'.t$⌋
by lambdaI j
Γ, x':$S \vdash T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash f_1\, t_1 = f_2\, t_2 \in T[t_1/x]$ ⌊Ax⌋
by applyEq x:$S \to T$
$\Gamma \vdash f_1 = f_2 \in x$:$S \to T$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in S$ ⌊Ax⌋

Γ, f: x:$S \to T$, $\Delta \vdash C$ ⌊**ext** t[$f\, s$, Axiom / y, v]⌋
by functionE i s
Γ, f: x:$S \to T$, $\Delta \vdash s \in S$ ⌊Ax⌋
Γ, f: x:$S \to T$, y:$T[s/x]$,
v: $y = f\, s \in T[s/x]$, $\Delta \vdash C$ ⌊**ext** t⌋

$\Gamma \vdash (\lambda x.t)\, s = t_2 \in T$ ⌊Ax⌋
by applyRed
$\Gamma \vdash t[s/x] = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash f_1 = f_2 \in x$:$S \to T$ ⌊**ext** t⌋
by functionExt j x_1:$S_1 \to T_1$ x_2:$S_2 \to T_2$
Γ, x':$S \vdash f_1\, x' = f_2\, x' \in T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma \vdash f_1 \in x_1$:$S_1 \to T_1$ ⌊Ax⌋
$\Gamma \vdash f_2 \in x_2$:$S_2 \to T_2$ ⌊Ax⌋

Γ, f: $S \to T$, $\Delta \vdash C$ ⌊**ext** $t[f\, s, /y]$⌋
by functionE_indep i
Γ, f: $S \to T$, $\Delta \vdash S$ ⌊**ext** s⌋
Γ, f: $S \to T$, y:T, $\Delta \vdash C$ ⌊**ext** t⌋

A.5.6 Produktraum

$\Gamma \vdash x_1{:}S_1{\times}T_1 \;=\; x_2{:}S_2{\times}T_2 \;\in\; \mathsf{U}_j$ [Ax]
by productEq
$\Gamma \vdash S_1 \;=\; S_2 \;\in\; \mathsf{U}_j$ [Ax]
$\Gamma,\; x'{:}S_1 \vdash T_1[x'/x_1] \;=\; T_2[x'/x_2] \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash \langle s_1,t_1\rangle \;=\; \langle s_2,t_2\rangle \;\in\; x{:}S{\times}T$ [Ax]
by pairEq j
$\Gamma \vdash s_1 \;=\; s_2 \;\in\; S$ [Ax]
$\Gamma \vdash t_1 \;=\; t_2 \;\in\; T[s_1/x]$ [Ax]
$\Gamma,\; x'{:}S \vdash T[x'/x] \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash x{:}S{\times}T$ [**ext** $\langle s,t\rangle$]
by pairI j s
$\Gamma \vdash s \;\in S$ [Ax]
$\Gamma \vdash T[s/x]$ [**ext** t]
$\Gamma,\; x'{:}S \vdash T[x'/x] \;\in\; \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{let}\ \langle x_1,y_1\rangle = e_1\ \mathsf{in}\ t_1$
$\quad = \mathsf{let}\ \langle x_2,y_2\rangle = e_2\ \mathsf{in}\ t_2 \;\in\; C[e_1/z]$ [Ax]
by spreadEq z C $x{:}S{\times}T$
$\Gamma \vdash e_1 \;=\; e_2 \;\in\; x{:}S{\times}T$ [Ax]
$\Gamma,\; s{:}S,\; t{:}T[s/x],\; y{:}\; e_1{=}\langle s,t\rangle \in x{:}S{\times}T$
$\quad \vdash t_1[s,t/x_1,y_1] = t_2[s,t/x_2,y_2] \;\in\; C[\langle s,t\rangle/z]$ [Ax]

$\Gamma,\; z{:}\; x{:}S{\times}T,\; \Delta \vdash C$ [**ext** $\mathsf{let}\ \langle s,t\rangle = z\ \mathsf{in}\ u$]
by productE i
$\Gamma,\; z{:}\; x{:}S{\times}T,\; s{:}S,\; t{:}T[s/x],\; \Delta[\langle s,t\rangle/z]$
$\quad \vdash C[\langle s,t\rangle/z]$ [**ext** u]

$\Gamma \vdash \mathsf{let}\ \langle x,y\rangle = \langle s,t\rangle\ \mathsf{in}\ u \;=\; t_2 \;\in\; T$ [Ax]
by spreadRed
$\Gamma \vdash u[s,t/x,y] \;=\; t_2 \;\in\; T$ [Ax]

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \;\in\; S{\times}T$ [Ax]
by pairEq_indep
$\Gamma \vdash s_1 \;=\; s_2 \;\in\; S$ [Ax]
$\Gamma \vdash t_1 \;=\; t_2 \;\in\; T$ [Ax]

$\Gamma \vdash S{\times}T$ [**ext** $\langle s,t\rangle$]
by pairI_indep
$\Gamma \vdash S$ [**ext** s]
$\Gamma \vdash T$ [**ext** t]

A.5.7 Disjunkte Vereinigung (Summe)

$\Gamma \vdash S_1{+}T_1 = S_2{+}T_2 \in \mathsf{U}_j$ [Ax]
by unionEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S{+}T$ [Ax]
by inlEq j
$\Gamma \vdash s_1 = s_2 \in S$ [Ax]
$\Gamma \vdash T \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S{+}T$ [Ax]
by inrEq j
$\Gamma \vdash t_1 = t_2 \in T$ [Ax]
$\Gamma \vdash S \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}(x_1) \mapsto u_1 \mid \mathsf{inr}(y_1) \mapsto v_1$
$= \mathsf{case}\ e_2\ \mathsf{of}\ \mathsf{inl}(x_2) \mapsto u_2 \mid \mathsf{inr}(y_2) \mapsto v_2$
$\in C[e_1/z]$ [Ax]
by decideEq z C $S{+}T$
$\Gamma \vdash e_1 = e_2 \in S{+}T$ [Ax]
$\Gamma,\ s{:}S,\ y{:}\ e_1{=}\mathsf{inl}(s) \in S{+}T$
$\vdash u_1[s/x_1] = u_2[s/x_2] \in C[\mathsf{inl}(s)/z]$ [Ax]
$\Gamma,\ t{:}T,\ y{:}\ e_1{=}\mathsf{inr}(t) \in S{+}T$
$\vdash v_1[t/y_1] = v_2[t/y_2] \in C[\mathsf{inr}(t)/z]$ [Ax]

$\Gamma \vdash \mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$
$= t_2 \in T$ [Ax]
by decideRedL
$\Gamma \vdash u[s/x] = t_2 \in T$ [Ax]

$\Gamma \vdash \mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$
$= t_2 \in T$ [Ax]
by decideRedR
$\Gamma \vdash v[t/y] = t_2 \in T$ [Ax]

$\Gamma \vdash S{+}T$ [**ext** $\mathsf{inl}(s)$]
by inlI j
$\Gamma \vdash S$ [**ext** s]
$\Gamma \vdash T \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash S{+}T$ [**ext** $\mathsf{inr}(t)$]
by inrI j
$\Gamma \vdash T$ [**ext** t]
$\Gamma \vdash S \in \mathsf{U}_j$ [Ax]

$\Gamma,\ z{:}S{+}T,\ \Delta \vdash C$
[**ext** $\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$]
by unionE i
$\Gamma,\ z{:}S{+}T,\ x{:}S,\ \Delta[\mathsf{inl}(x)/z]$
$\vdash C[\mathsf{inl}(x)/z]$ [**ext** u]
$\Gamma,\ z{:}S{+}T,\ y{:}T,\ \Delta[\mathsf{inr}(y)/z]$
$\vdash C[\mathsf{inr}(y)/z]$ [**ext** v]

A.5.8 Gleichheit

$\Gamma \vdash\ s_1 = t_1 \in T_1\ =\ s_2 = t_2 \in T_2\ \in\ \mathsf{U}_j$ [Ax]
by equalityEq
$\Gamma \vdash\ T_1\ =\ T_2\ \in\ \mathsf{U}_j$ [Ax]
$\Gamma \vdash\ s_1\ =\ s_2\ \in\ T_1$ [Ax]
$\Gamma \vdash\ t_1\ =\ t_2\ \in\ T_1$ [Ax]

$\Gamma \vdash\ \mathsf{Axiom}\ \in\ s = t \in T$ [Ax]
by axiomEq
$\Gamma \vdash\ s\ =\ t\ \in\ T$ [Ax]

$\Gamma,\ z\colon s = t \in T,\ \Delta \vdash\ C$ [**ext** u]
by equalityE i
$\Gamma,\ z\colon s = t \in T,\ \Delta[\mathsf{Axiom}/z]$
$\vdash\ C[\mathsf{Axiom}/z]$ [**ext** u]

$\Gamma,\ x{:}T,\ \Delta \vdash\ x\ \in\ T$ [Ax]
by hypEq i

$\Gamma \vdash\ s = t \in T$ [Ax]
by symmetry
$\Gamma \vdash\ t = s \in T$ [Ax]

$\Gamma \vdash\ s = t \in T$ [Ax]
by transitivity t'
$\Gamma \vdash\ s = t' \in T$ [Ax]
$\Gamma \vdash\ t' = t \in T$ [Ax]

$\Gamma \vdash\ C[s/x]$ [**ext** u]
by subst j $s = t \in T$ x C
$\Gamma \vdash\ s = t \in T$ [Ax]
$\Gamma \vdash\ C[t/x]$ [**ext** u]
$\Gamma,\ x{:}T \vdash\ C\ \in\ \mathsf{U}_j$ [Ax]

A.5.9 Listen

$\Gamma \vdash\ T_1\,\mathsf{list}\ =\ T_2\,\mathsf{list}\ \in\ \mathsf{U}_j$ [Ax]
by listEq
$\Gamma \vdash\ T_1\ =\ T_2\ \in\ \mathsf{U}_j$ [Ax]

$\Gamma \vdash\ []\ \in\ T\,\mathsf{list}$ [Ax]
by nilEq j
$\Gamma \vdash\ T\ \in\ \mathsf{U}_j$ [Ax]

$\Gamma \vdash\ T\,\mathsf{list}$ [**ext** $[]$]
by nilI j
$\Gamma \vdash\ T\ \in\ \mathsf{U}_j$ [Ax]

$\Gamma \vdash\ t_1.l_1\ =\ t_2.l_2\ \in\ T\,\mathsf{list}$ [Ax]
by consEq
$\Gamma \vdash\ t_1\ =\ t_2\ \in\ T$ [Ax]
$\Gamma \vdash\ l_1\ =\ l_2\ \in\ T\,\mathsf{list}$ [Ax]

$\Gamma \vdash\ T\,\mathsf{list}$ [**ext** $t.l$]
by consI
$\Gamma \vdash\ T$ [**ext** t]
$\Gamma \vdash\ T\,\mathsf{list}$ [**ext** l]

$\Gamma \vdash\ \mathsf{list\_ind}(s_1;\ base_1;\ x_1, l_1, f_{xl1}.t_1)$
$=\ \mathsf{list\_ind}(s_2;\ base_2;\ x_2, l_2, f_{xl2}.t_2)\ \in\ T[s_1/z]$ [Ax]
by list_indEq z T $S\,\mathsf{list}$
$\Gamma \vdash\ s_1\ =\ s_2\ \in\ S\,\mathsf{list}$ [Ax]
$\Gamma \vdash\ base_1\ =\ base_2\ \in T[[]/z]$ [Ax]
$\Gamma,\ x{:}S,\ l{:}S\,\mathsf{list},\ f_{xl}{:}T[l/z] \vdash$
$t_1[x, l, f_{xl}\,/\,x_1, l_1, f_{xl1}]\ =\ t_1[x, l, f_{xl}\,/\,x_2, l_2, f_{xl2}]$
$\in\ T[x.l/z]$ [Ax]

$\Gamma,\ z\colon T\,\mathsf{list},\ \Delta \vdash\ C$
[**ext** $\mathsf{list\_ind}(z;\ base;\ x, l, f_{xl}.t)$]
by listE i
$\Gamma,\ z\colon T\,\mathsf{list},\ \Delta \vdash\ C[[]/z]$ [**ext** $base$]
$\Gamma,\ z\colon T\,\mathsf{list},\ \Delta,\ x{:}T,\ l{:}T\,\mathsf{list},\ f_{xl}{:}C[l/z]$
$\vdash\ C[x.l/z]$ [**ext** t]

$\Gamma \vdash\ \mathsf{list\_ind}([];\ base;\ x, l, f_{xl}.t)\ =\ t_2\ \in\ T$ [Ax]
by list_indRedBase
$\Gamma \vdash\ base\ =\ t_2\ \in\ T$ [Ax]

$\Gamma \vdash\ \mathsf{list\_ind}(s.u;\ base;\ x, l, f_{xl}.t)\ =\ t_2\ \in\ T$ [Ax]
by list_indRedUp
$\Gamma \vdash\ t[s, u, \mathsf{list\_ind}(u;\ base;\ x, l, f_{xl}.t)\ /x, l, f_{xl}]$
$=\ t_2\ \in\ T$ [Ax]

A.5.10 Teilmengen

$\Gamma \vdash \{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\} \in \mathsf{U}_j$ [Ax]
by setEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ [Ax]
$\Gamma, x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash s = t \in \{x{:}S \mid T\}$ [Ax]
by elementEq j
$\Gamma \vdash s = t \in S$ [Ax]
$\Gamma \vdash T[s/x]$ [Ax]
$\Gamma, x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash \{x{:}S \mid T\}$ [**ext** s]
by elementI j s
$\Gamma \vdash s \in S$ [Ax]
$\Gamma \vdash T[s/x]$ [Ax]
$\Gamma, x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ [Ax]

$\Gamma, z{:}\{x{:}S \mid T\}, \Delta \vdash C$ [**ext** $(\lambda y.t)\, z$]
by setE i
$\Gamma, z{:}\{x{:}S \mid T\}, y{:}S, [\![v]\!]{:}T[y/x], \Delta[y/z] \vdash C[y/z]$ [**ext** t]

$\Gamma \vdash s = t \in \{S \mid T\}$ [Ax]
by elementEq_indep
$\Gamma \vdash s = t \in S$ [Ax]
$\Gamma \vdash T$ [Ax]

$\Gamma \vdash \{S \mid T\}$ [**ext** s]
by elementI_indep
$\Gamma \vdash S$ [**ext** s]
$\Gamma \vdash T$ [Ax]

A.5.11 Quotienten

$\Gamma \vdash x_1, y_1 : T_1//E_1 = x_2, y_2 : T_2//E_2 \in \mathsf{U}_j$ [Ax]
by quotientEq_weak
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]
$\Gamma, x{:}T_1, y{:}T_1 \vdash E_1[x,y/x_1,y_1] = E_2[x,y/x_2,y_2] \in \mathsf{U}_j$ [Ax]
$\Gamma, x{:}T_1, y{:}T_1 \vdash E_1[x,x/x_1,y_1]$ [Ax]
$\Gamma, x{:}T_1, y{:}T_1, v{:}E_1[x,y/x_1,y_1] \vdash E_1[y,x/x_1,y_1]$ [Ax]
$\Gamma, x{:}T_1, y{:}T_1, z{:}T_1, v{:}E_1[x,y/x_1,y_1], v'{:}E_1[y,z/x_1,y_1] \vdash E_1[x,z/x_1,y_1]$ [Ax]

$\Gamma \vdash x_1, y_1 : T_1//E_1 = x_2, y_2 : T_2//E_2 \in \mathsf{U}_j$ [Ax]
by quotientEq
$\Gamma \vdash x_1, y_1 : T_1//E_1 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash x_2, y_2 : T_2//E_2 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]
$\Gamma, v{:}\, T_1 = T_2 \in \mathsf{U}_j, x{:}T_1, y{:}T_1 \vdash E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ [Ax]
$\Gamma, v{:}\, T_1 = T_2 \in \mathsf{U}_j, x{:}T_1, y{:}T_1 \vdash E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ [Ax]

$\Gamma \vdash s = t \in x, y : T//E$ [Ax]
by memberEq_weak j
$\Gamma \vdash x, y : T//E \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash s = t \in T$ [Ax]

$\Gamma \vdash x, y : T//E$ [**ext** t]
by memberI j
$\Gamma \vdash x, y : T//E \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T$ [**ext** t]

$\Gamma \vdash s = t \in x, y : T//E$ [Ax]
by memberEq j
$\Gamma \vdash x, y : T//E \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash s \in T$ [Ax]
$\Gamma \vdash t \in T$ [Ax]
$\Gamma \vdash E[s,t/x,y]$ [Ax]

$\Gamma, v{:}\, s = t \in x, y : T//E, \Delta \vdash C$ [**ext** u]
by quotient_eqE i j
$\Gamma, v{:}\, s = t \in x, y : T//E, [\![v']\!]{:}E[s,t/x,y], \Delta \vdash C$ [**ext** u]
$\Gamma, v{:}\, s = t \in x, y : T//E, \Delta, x'{:}T, y'{:}T \vdash E[x',y'/x,y] \in \mathsf{U}_j$ [Ax]

$\Gamma, z{:}\, x, y : T//E, \Delta \vdash s = t \in S$ [Ax]
by quotientE i j
$\Gamma, z{:}\, x, y : T//E, \Delta, x'{:}T, y'{:}T \vdash E[x',y'/x,y] \in \mathsf{U}_j$ [Ax]
$\Gamma, z{:}\, x, y : T//E, \Delta \vdash S \in \mathsf{U}_j$ [Ax]
$\Gamma, z{:}\, x, y : T//E, \Delta, x'{:}T, y'{:}T, v{:}E[x',y'/x,y] \vdash s[x'/z] = t[y'/z] \in S[x'/z]$ [Ax]

A.5.12 Induktive Typen

$\Gamma \vdash$ rectype $X_1 = T_{X1}$ = rectype $X_2 = T_{X2} \in \mathsf{U}_j$ [Ax]
by recEq
$\Gamma,\ X{:}\mathsf{U}_j \vdash T_{X1}[X/X_1] = T_{X2}[X/X_2] \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash s = t \in$ rectype $X = T_X$ [Ax]
by rec_memEq j
$\Gamma \vdash s = t \in T_X[\text{rectype } X = T_X / X]$ [Ax]
$\Gamma \vdash$ rectype $X = T_X \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash$ rectype $X = T_X$ [**ext** t]
by rec_memI j
$\Gamma \vdash T_X[\text{rectype } X = T_X / X]$ [**ext** t]
$\Gamma \vdash$ rectype $X = T_X \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash$ let* $f_1(x_1) = t_1$ in $f_1(e_1)$
= let* $f_2(x_2) = t_2$ in $f_2(e_2) \in T[e_1/z]$ [Ax]
by rec_indEq z T rectype $X = T_X$ j
$\Gamma \vdash e_1 = e_2 \in$ rectype $X = T_X$ [Ax]
$\Gamma \vdash$ rectype $X = T_X \in \mathsf{U}_j$ [Ax]
$\Gamma,\ P{:}\ (\text{rectype } X = T_X) \to \mathbb{P}_j,$
$f{:}\ (y{:}\{x{:}\text{rectype } X = T_X \mid P(x)\} \to T[y/z]),$
$x{:}\ T_X[\{x{:}\text{rectype } X = T_X \mid P(x)\} / X]$
$\vdash t_1[f, x / f_1, x_1] = t_2[f, x / f_2, x_2] \in T[x/z]$ [Ax]

$\Gamma,\ z{:}$ rectype $X = T_X,\ \Delta \vdash C$
[**ext** let* $f(x) = t[\lambda y.\Lambda / P]$ in $f(z)$]
by recE i j
$\Gamma,\ z{:}$ rectype $X = T_X,\ \Delta$
$\vdash$ rectype $X = T_X \in \mathsf{U}_j$ [Ax]
$\Gamma,\ z{:}$ rectype $X = T_X,\ \Delta$
$P{:}\ (\text{rectype } X = T_X) \to \mathbb{P}_j,$
$f{:}\ (y{:}\{x{:}\text{rectype } X = T_X \mid P(x)\} \to C[y/z]),$
$x{:}\ T_X[\{x{:}\text{rectype } X = T_X \mid P(x)\} / X]$
$\vdash C[x/z]$ [**ext** t]

$\Gamma,\ z{:}$ rectype $X = T_X,\ \Delta \vdash C$ [**ext** $t[z/x]$]
by recE_unroll i
$\Gamma,\ z{:}$ rectype $X = T_X,\ \Delta,$
$x{:}\ T_X[\text{rectype } X = T_X / X],$
$v{:}\ z{=}x \in T_X[\text{rectype } X = T_X / X] \vdash C[x/z]$ [**ext** t]

A.5.13 Rekursive Funktionen

$\Gamma \vdash S_1 \rightharpoonup T_1 = S_2 \rightharpoonup T_2 \in \mathsf{U}_j$ [Ax]
by pfunEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ [Ax]
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ [Ax]

$\Gamma \vdash$ (letrec $f_1(x_1) = t_1$)
= (letrec $f_2(x_2) = t_2$) $\in S \rightharpoonup T$ [Ax]
by fixEq j
$\Gamma \vdash$ letrec $f_1(x_1) = t_1 \in S \rightharpoonup T$ [Ax]
$\Gamma \vdash$ letrec $f_2(x_2) = t_2 \in S \rightharpoonup T$ [Ax]
$\Gamma \vdash \{x{:}S \mid \text{dom}(\text{letrec } f_1(x_1) = t_1)(x)\}$
$= \{x{:}S \mid \text{dom}(\text{letrec } f_2(x_2) = t_2)(x)\} \in \mathsf{U}_j$ [Ax]
$\Gamma,\ y{:}\{x{:}S \mid \text{dom}(\text{letrec } f_1(x_1) = t_1)(x)\}$
$\vdash$ (letrec $f_1(x_1) = t_1$) (y)
= (letrec $f_2(x_2) = t_2$) $(y) \in T$ [Ax]

$\Gamma \vdash$ letrec $f(x) = t \in S \rightharpoonup T$ [Ax]
by fixMem j
$\Gamma \vdash S \rightharpoonup T \in \mathsf{U}_j$ [Ax]
$\Gamma,\ f'{:}S \rightharpoonup T,\ x'{:}S \vdash \mathcal{E}[\![\, t[f', x' / f, x] \,]\!] \in \mathsf{U}_j$ [Ax]
$\Gamma,\ f'{:}S \rightharpoonup T,\ x'{:}S,\ \mathcal{E}[\![\, t[f', x' / f, x] \,]\!]$
$\vdash t[f', x' / f, x] \in T$ [Ax]

$\Gamma \vdash f_1(t_1) = f_2(t_2) \in T$ [Ax]
by apply_pEq $S \rightharpoonup T$
$\Gamma \vdash f_1 = f_2 \in S \rightharpoonup T$ [Ax]
$\Gamma \vdash t_1 = t_2 \in \{x{:}S \mid \text{dom}(f_1)(x)\}$ [Ax]

$\Gamma,\ f{:}S \rightharpoonup T,\ \Delta \vdash C$ [**ext** $t[f(s), \text{Axiom} / y, v]$]
by pfunE i s
$\Gamma,\ f{:}S \rightharpoonup T,\ \Delta \vdash s \in \{x{:}S \mid \text{dom}(f_1)(x)\}$ [Ax]
$\Gamma,\ f{:}S \rightharpoonup T,\ \Delta,\ y{:}T,\ v{:}\ y{=}f(s) \in T \vdash C$ [**ext** t]

$\Gamma \vdash$ (letrec $f(x) = t)\ (u) = t_2 \in T$ [Ax]
by apply_pRed
$\Gamma \vdash t[\text{letrec } f(x) = t,\ u\ /\ f, x] = t_2 \in T$ [Ax]

$\Gamma \vdash \text{dom}(f_1) = \text{dom}(f_2) \in S \to \mathbb{P}_j$ [Ax]
by domEq $S \rightharpoonup T$
$\Gamma \vdash f_1 = f_2 \in S \rightharpoonup T$ [Ax]
$\Gamma \vdash S \rightharpoonup T \in \mathsf{U}_j$ [Ax]

A.5.14 Strukturelle Regeln und sonstige Regeln

$\Gamma,\ x{:}T,\ \Delta \vdash T$ ⌊**ext** x⌋
by `hypothesis` i

$\Gamma \vdash T$ ⌊**ext** t⌋
by `intro` t
$\Gamma \vdash t \in T$ ⌊Ax⌋

$\Gamma,\Delta \vdash C$ ⌊**ext** $(\lambda x.t)\ s$⌋
by `cut` i T
$\Gamma,\Delta \vdash T$ ⌊**ext** s⌋
$\Gamma,x{:}T,\Delta \vdash C$ ⌊**ext** t⌋

$\Gamma,\ x{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋
by `thin` i
$\Gamma,\Delta \vdash C$ ⌊**ext** t⌋

$\Gamma \vdash C$ ⌊**ext** t⌋
by `compute` $tagC$
$\Gamma \vdash C{\downarrow}_{tagC}$ ⌊**ext** t⌋

$\Gamma \vdash C$ ⌊**ext** t⌋
by `rev_compute` $tagC$
$\Gamma, \vdash C{\uparrow}_{tagC}$ ⌊**ext** t⌋

$\Gamma,\ z{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋
by `computeHyp` i $tagT$
$\Gamma,\ z{:}T{\downarrow}_{tagT},\ \Delta \vdash C$ ⌊**ext** t⌋

$\Gamma,\ z{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋
by `rev_computeHyp` i $tagT$
$\Gamma,\ z{:}T{\uparrow}_{tagT},\ \Delta \vdash C$ ⌊**ext** t⌋

$\Gamma \vdash C$ ⌊**ext** t⌋
by `unfold` *def-name*
$\Gamma \vdash C_{\downarrow}$ ⌊**ext** t⌋

$\Gamma \vdash C_{\downarrow}$ ⌊**ext** t⌋
by `fold` *def-name*
$\Gamma \vdash C$ ⌊**ext** t⌋

$\Gamma,\ z{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋
by `unfoldHyp` i *def-name*
$\Gamma,\ z{:}T_{\downarrow},\ \Delta \vdash C$ ⌊**ext** t⌋

$\Gamma,\ z{:}T_{\downarrow},\ \Delta \vdash C$ ⌊**ext** t⌋
by `foldHyp` i *def-name*
$\Gamma,\ z{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋

$\Gamma,\ z{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋
by `replaceHyp` i S j
$\Gamma,\ z{:}S,\ \Delta \vdash C$ ⌊**ext** t⌋
$\Gamma,\ z{:}T,\ \Delta \vdash T = S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash C$ ⌊**ext** t⌋
by `lemma` *theorem-name*

$\Gamma \vdash t \in T$ ⌊Ax⌋
by `extract` *theorem-name*

$\Gamma \vdash C$ ⌊**ext** $t[\sigma]$⌋
by `instantiate` Γ' C' σ
$\Gamma' \vdash C'$ ⌊**ext** t⌋

$\Gamma \vdash C$ ⌊**ext** $t[y/x]$⌋
by `rename` y x
$\Gamma[x/y] \vdash C[x/y]$ ⌊**ext** t⌋

$\Gamma \vdash s = t \in T$ ⌊Ax⌋
by `equality`

$\Gamma \vdash C$ ⌊**ext** t⌋
by `arith` j
$\Gamma \vdash s_1 \in \mathbb{Z}$ ⌊Ax⌋
$\vdots$

Anhang B

Konservative Erweiterungen der Typentheorie und ihre Gesetze

Achtung: Dieser Teil wurde relativ zügig aus einer alten Quelle zusammengestellt und könnte divese kleine Fehler enthalten. Für Hinweise bin ich dankbar.

B.1 Endliche Mengen

Der Typ der endlichen Mengen ist ein generischer Datentyp, der auf der Basis der Konzepte `Set, =, ∅, +` und $\in$ eingeführt werden kann. Die einfachste Form einer Implementierung ist eine Simulation durch Listen modulo einer neuen Definition der Gleichheit.

BASISKONZEPTE

Definition B.1.1 (Beschränkte Boole'sche Quantoren)

$\forall x \in S.p_x \equiv$ `list_ind(`S`; true; a,_,pS'.pS'` $\wedge p_x$`[a/`x`])`
$\exists x \in S.p_x \equiv$ `list_ind(`S`; false; a,_,pS'.pS'` $\vee p_x$`[a/`x`])`

Definition B.1.2 (Implementierung endlicher Mengen)

$\emptyset \equiv$ `[]`
`+` $\equiv \lambda$`a,S. a.S`
$\in_\alpha \equiv \lambda$`a,S.` $\exists$`x`$\in$`S.x`$=_\alpha$`a`
$=_{\mathtt{Set}(\alpha)} \equiv \lambda$`S,T.` $(\forall$`a`$\in$`S. a`$\in_\alpha$`T)` $\wedge$ $(\forall$`a'`$\in$`T. a'`$\in_\alpha$`S)`
`Set(`α`)` $\equiv$ `(S,T):`α`list //` $S_{=_{\mathtt{Set}(\alpha)}}T$

Der Index α zeigt an, daß eine Operation von der Gleichheit auf dem Typ α abhängt und somit nur dann berechenbar ist, wenn es hierfür eine boolesche Entscheidungsfunktion gibt. Der Übersichtlichkeit halber wird er ab sofort unterdrückt.

Lemma B.1.3 Axiome endlicher Mengen

$\forall\alpha$`:TYPES.` $\forall$`S:Set(`α`).` $\forall$`x,a:`α`.` $\forall$`P:PROP(Set(`α`)).`

1. `Set(`α`)` $\in$ `TYPES`
2. $\emptyset$ $\in$ `Set(`α`)`
3. `+` $\in$ `Set(`α`)`$\times\alpha\rightarrow$`Set(`α`)`
4. $\in$ $\in \alpha\times$`Set(`α`)`$\rightarrow\mathbb{B}$
5. `a` $\notin \emptyset$
6. `x` $\in$ `(S+a)` $\Leftrightarrow$ `(x=a` $\vee$ `x` $\in$ `S)`
7. `(S+a)+x = (S+x)+a`
8. `(S+a)+a = S+a`
9. `(P(`$\emptyset$`)` $\wedge$ $\forall$`S:Set(`α`).`$\forall$`a:`α`.P(S)`$\Rightarrow$`P(S+a))` $\Rightarrow$ $\forall$`S:Set(`α`).P(S)`

Lemma B.1.4 Finite Constructability and Extensionality

$\forall\alpha$:TYPES.$\forall$S,S':Set(α).
1. S=$\emptyset$ $\vee$ $\exists$a:α.$\exists$S':Set(α).(a$\notin$S' $\wedge$ S=S'+a)
2. S=S' $\Leftrightarrow$ $\forall$a:α.(a$\in$S $\Leftrightarrow$ a$\in$S')

ABGELEITETE KONZEPTE

Die weiteren Operationen auf endlichen Mengen hängen von der oben gegebenen speziellen Implementierung nicht ab, da sie sich i.w. durch $\emptyset$, +, $\in$, Gleichheit und den Induktionsoperator list_ind beschreiben lassen.

Definition B.1.5 (Set Notation)

if S=$\emptyset$ then t else t'	$\equiv$ list_ind(S; t; _,_,_,t')
let S=S'+a $\in$ exp	$\equiv$ list_ind(S; ∞; a,S',_,exp)
let S=$\{a\}$ $\in$ exp	$\equiv$ let S=S'+a $\in$ if S'=$\emptyset$ then exp else ∞
let S=$\{a,a'\}$ $\in$ exp	$\equiv$ let S=S'+a' $\in$ let S'=$\{a\}$ $\in$ exp
let S=$\{a,a',a''\}$ $\in$ exp	$\equiv$ let S=S'+a'' $\in$ let S'=$\{a,a'\}$ $\in$ exp
$\lambda\{a\}.exp$	$\equiv$ λS.let S=$\{a\}$ $\in$ exp
$\lambda\{a,a'\}.exp$	$\equiv$ λS.let S=$\{a,a'\}$ $\in$ exp
$\lambda\{a,a',a'\}.exp$	$\equiv$ λS.let S=$\{a,a',a'\}$ $\in$ exp

Definition B.1.6 (Set Operations)

empty?	$\equiv$ λS. if S=$\emptyset$ then true else false
$\subseteq$	$\equiv$ λS,S'. $\forall$x$\in$S.x$\in$S'
$\{$*list-exp*$\}$	$\equiv$ *list-exp*.nil
$\{i..j\}$	$\equiv$ ind(j-i; _,_.$\emptyset$; $\{j\}$; diff,j-set.j-set+(j-diff))
$\{f_x \mid x\in S\wedge p_x\}$	$\equiv$ list_ind(S; $\emptyset$; a,_,GSF. if p_x[a/x] then GSF+f_x[a/x] else GSF
$\{f_x \mid x\in S\}$	$\equiv$ $\{f_x \mid x\in S \wedge$ true$\}$
$\lvert S\rvert$	$\equiv$ list_ind(S; 0; a,S',card. if a$\in$S' then card else card+1)
-	$\equiv$ λS,a. $\{$x$\mid$x$\in$S$\wedge$x$\neq$a$\}$
$\cup$	$\equiv$ λS,S'. list_ind(S'; S; a,_,union.union+a)
$\cap$	$\equiv$ λS,S'. $\{$x$\mid$x$\in$S$\wedge$x$\in$S'$\}$
\	$\equiv$ λS,S'. $\{$x$\mid$x$\in$S$\wedge$x$\notin$S'$\}$
$\bigcup$	$\equiv$ λFAMILY. list_ind(FAMILY; $\emptyset$; S,FAM,Union.Union$\cup$S)
$\bigcap$	$\equiv$ λFAMILY. list_ind(FAMILY; ∞; S,FAM,inter. if empty?(FAM) then S else inter$\bigcap$S)
arb	$\equiv$ λS. list_ind(S; ∞; a,_,_. a)
map	$\equiv$ λf,S. $\{$f(x)$\mid$x$\in$S$\}$
reduce	$\equiv$ λop,S. list_ind(S; ∞; a,S',redS'. if empty?(S') then a else if a$\in$S' then redS' else op(redS',a))
T=$S\uplus S'$	$\equiv$ T=$S\cup S'$ $\wedge$ empty?($S\cap S'$)

Die unten angegebenen Lemmata beschreiben die Grundgesetze der soeben definierten Operationen. Im wesentlichen zeigen sie, wie eine Operation über diverse andere distributiert. Sie können daher zur Vereinfachung eingesetzt werden. Wir gruppieren sie gemäß der jeweils äußersten Operation.

Lemma B.1.7 Operation Signatures

$\forall\alpha,\beta$:TYPES. $\forall$p:$\alpha\to\mathbb{B}$. $\forall$f:$\alpha\to\beta$.

1. empty? $\in$ Set(α)$\to\mathbb{B}$
2. λS. $\exists$x$\in$S.p(x) $\in$ Set(α)$\to\mathbb{B}$
3. λS. $\forall$x$\in$S.p(x) $\in$ Set(α)$\to\mathbb{B}$
4. $\subseteq$ $\in$ Set(α)$^2\to\mathbb{B}$
5. λi,j. {i..j} $\in$ $\mathbb{Z}^2\to$Set($\mathbb{Z}$)
6. λS. {f(x)|x$\in$S $\wedge$ p(x)} $\in$ Set(α)$\to$Set(β)
7. λS. |S| $\in$ Set(α)$\to\mathbb{N}$
8. - $\in$ Set(α)$\times\alpha\to$Set(α)
9. $\cup$ $\in$ Set(α)$^2\to$Set(α)
10. $\cap$ $\in$ Set(α)$^2\to$Set(α)
11. $\setminus$ $\in$ Set(α)$^2\to$Set(α)
12. $\bigcup$ $\in$ Set(Set(α))$\to$Set(α)
13. $\bigcap$ $\in$ Set(Set(α))$\not\to$Set(α
14. arb $\in$ Set(α)$\to\alpha$
15. map $\in$ ($\alpha\to\beta$)$\times$Set(α)$\to$Set(β)
16. reduce $\in$ ($\alpha^2\to\alpha$)$\times$Set(α)$\not\to\alpha$

Lemma B.1.8 Element addition

$\forall\alpha$:TYPES.$\forall$S:Set(α).$\forall$a:α.

1. (S+a)$\neq\emptyset$
2. a$\in$S $\Rightarrow$ S+a=S
3. a$\notin$S $\Rightarrow$ S+a$\neq$S
4. a$\in$S $\Rightarrow$ (S-a)+a = S

Lemma B.1.9 Membership

$\forall\alpha,\beta$:TYPES.$\forall$a,a':α.$\forall$S,S':Set(α).$\forall$FAM:Set(Set(α)).$\forall$p:$\alpha\to\mathbb{B}$.$\forall$f:$\alpha\to\beta$.$\forall$b:β. $\forall$i,j,k:$\mathbb{Z}$.

1. a'$\in${a} $\Leftrightarrow$ a'=a
2. k$\in${i..j} $\Leftrightarrow$ (i$\leq$k $\wedge$ k$\leq$j)
3. b$\in${f(x)|x$\in$S$\wedge$p(x)} $\Leftrightarrow$ $\exists$x$\in$S. p(x) $\wedge$ b=f(x)
4. a$\in${x|x$\in$S$\wedge$p(x)} $\Leftrightarrow$ a$\in$S $\wedge$ p(a)
5. a$\in$S-a' $\Leftrightarrow$ a$\in$S $\wedge$ a$\neq$a'
6. a$\in$S$\cup$S' $\Leftrightarrow$ a$\in$S $\vee$ a$\in$S
7. a$\in$S$\cap$S' $\Leftrightarrow$ a$\in$S $\wedge$ a$\in$S'
8. a$\in$S$\setminus$S' $\Leftrightarrow$ a$\in$S $\wedge$ a$\notin$S'
9. a$\in\bigcup$FAM $\Leftrightarrow$ $\exists$S$\in$FAM. a$\in$S
10. a$\in\bigcap$FAM $\Leftrightarrow$ $\forall$S$\in$FAM. a$\in$S
11. arb(S)$\in$S
12. b$\in$map(f,S) $\Leftrightarrow$ $\exists$x$\in$S.b=f(x)

Lemma B.1.10 empty?

$\forall\alpha,\beta$:TYPES.$\forall$a:α.$\forall$S,S':Set(α).$\forall$FAM:Set(Set(α)).$\forall$p:$\alpha\to\mathbb{B}$.$\forall$f:$\alpha\to\beta$.

1. empty?(S) $\Leftrightarrow$ S=$\emptyset$
2. empty?(S) $\Leftrightarrow$ |S|=0
3. S$\subseteq$S' $\Rightarrow$ empty?(S') $\Rightarrow$ empty?(S)
4. $\neg$empty?(S+a)
5. $\neg$empty?({a})
6. empty?({f(x)|x$\in$S$\wedge$p(x)}) $\Leftrightarrow$ empty?(S) $\vee$ $\forall$x$\in$S.$\neg$p(x))
7. empty?(S-a) $\Leftrightarrow$ empty?(S) $\vee$ S={a}
8. empty?(S$\cup$S') $\Leftrightarrow$ empty?(S) $\wedge$ empty?(S')
9. empty?(S) $\vee$ empty?(S') $\Rightarrow$ empty?(S$\cap$S')
10. empty?(S) $\Rightarrow$ empty?(S$\setminus$S')
11. empty?($\bigcup$FAM) $\Leftrightarrow$ $\forall$S$\in$FAM. empty?(S)
12. $\exists$S$\in$FAM. empty?(S) $\Rightarrow$ empty?($\bigcap$FAM)

Lemma B.1.11 Limited Universal Quantifier

$\forall\alpha,\beta$:TYPES.$\forall$a:α.$\forall$S,S':Set(α).$\forall$FAM:Set(Set(α)).$\forall$p,q:$\alpha\to\mathbb{B}$.$\forall$f:$\alpha\to\beta$.

1. $\forall x\in\emptyset.\ p(x)$
2. $\forall x\in S+a.\ p(x) \quad\Leftrightarrow\quad p(a)\wedge\forall x\in S.p(x)$
3. $\forall x\in\{a\}.\ p(x) \quad\Leftrightarrow\quad p(a)$
4. $S\subseteq S' \ \Rightarrow\ \forall x\in S'.\ p(x) \ \Rightarrow\ \forall x\in S.\ p(x)$
5. $\forall x\in\{f(y)\,|\,y\in S\wedge q(y)\}.\ p(x) \ \Leftrightarrow\ \forall y\in S.\ q(y)\Rightarrow p(f(y))$
6. $\forall x\in S-a.\ p(x)\ \wedge\ p(a) \quad\Rightarrow\quad \forall x\in S.p(x)$

6a. $a\in S \quad\Rightarrow\quad \forall x\in S-a.\ p(x)\ \wedge\ p(a) \quad\Leftrightarrow\quad \forall x\in S.p(x)$

7. $\forall x\in S\cup S'.\ p(x) \quad\Leftrightarrow\quad \forall x\in S.p(x)\ \wedge\ \forall x\in S'.p(x)$
8. $\forall x\in S\cap S'.\ p(x) \quad\Leftarrow\quad \forall x\in S.p(x)\ \vee\ \forall x\in S'.p(x)$
9. $\forall x\in S\setminus S'.\ p(x)\ \wedge\ \forall x\in S'.p(x) \quad\Rightarrow\quad \forall x\in S.p(x)$
10. $\forall x\in\bigcup FAM.\ p(x) \quad\Leftrightarrow\quad \forall S\in FAM.\ \forall x\in S.p(x)$
11. $\forall x\in\bigcap FAM.\ p(x) \quad\Leftarrow\quad \exists S\in FAM.\ \forall x\in S.p(x)$

Lemma B.1.12 Limited Existential Quantifier

$\forall\alpha,\beta$:TYPES.$\forall$a:α.$\forall$S,S':Set(α).$\forall$FAM:Set(Set(α)).$\forall$p,q:$\alpha\to\mathbb{B}$.$\forall$f:$\alpha\to\beta$.

1. $\neg\exists x\in\emptyset.\ p(x)$
2. $\exists x\in S+a.\ p(x) \quad\Leftrightarrow\quad p(a)\vee\exists x\in S.p(x)$
3. $\exists x\in\{a\}.\ p(x) \quad\Leftrightarrow\quad p(a)$
4. $S\subseteq S' \ \Rightarrow\ \exists x\in S.\ p(x) \ \Rightarrow\ \exists x\in S'.\ p(x)$
5. $\exists x\in\{f(y)\,|\,y\in S\wedge q(y)\}.\ p(x) \ \Leftrightarrow\ \exists y\in S.\ q(y)\wedge p(f(y))$
6. $\exists x\in S-a.\ p(x) \quad\Leftarrow\quad \neg p(a)\wedge\exists x\in S.p(x)$
7. $\exists x\in S\cup S'.\ p(x) \quad\Leftrightarrow\quad \exists x\in S.p(x)\ \vee\ \exists x\in S'.p(x)$
8. $\exists x\in S\cap S'.\ p(x) \ \Leftrightarrow\ \exists x\in S.p(x)\ \wedge\ \exists x\in S'.p(x)$
9. $\exists x\in S\setminus S'.\ p(x) \ \Leftarrow\ \exists x\in S.p(x)\ \wedge\ \nexists x\in S'.p(x)$
10. $\exists x\in\bigcup FAM.\ p(x) \quad\Leftrightarrow\quad \exists S\in FAM.\ \exists x\in S.p(x)$
11. $\exists x\in\bigcap FAM.\ p(x) \ \Leftrightarrow\ \exists x{:}\alpha.\ p(x)\wedge\forall S\in FAM.x\in S$

Lemma B.1.13 Subset

$\forall\alpha,\beta$:TYPES.$\forall$a:α.$\forall$S,S',S'':Set(α).$\forall$FAM:Set(Set(α)).$\forall$p:$\alpha\to\mathbb{B}$.$\forall$f:$\alpha\to\beta$.$\forall$i,j,k,l:$\mathbb{Z}$.

1. $\emptyset\subseteq S$
2. $S\subseteq\emptyset \ \Leftrightarrow$ empty?(S)
3. $S+a\subseteq S' \quad\Leftrightarrow\quad S\subseteq S'\ \wedge\ a\in S'$
4. $\{a\}\subseteq S \quad\Leftrightarrow\quad a\in S$
5. $\{i..j\}\subseteq\{k..l\} \quad\Leftrightarrow\quad k\leq i\wedge j\leq l$
6. $S\subseteq S' \ \Rightarrow\ \{f(x)\,|\,x\in S\wedge p(x)\}\subseteq\{f(x)\,|\,x\in S'\wedge p(x)\}$
7. $S-a\subseteq S$
8. $S\subseteq S\cup S'$

8a. $S\cup S'\subseteq S'' \ \Leftrightarrow\ S\subseteq S''\ \wedge\ S'\subseteq S''$

9. $S\cap S'\subseteq S$
10. $S\setminus S'\subseteq S$
11. $S\subseteq S$
12. $S\subseteq S'\ \wedge\ S'\subseteq S'' \quad\Rightarrow\quad S\subseteq S''$
13. $S\subseteq S'\ \wedge\ S'\subseteq S \quad\Leftrightarrow\quad S=S'$

Lemma B.1.14 Integer Subset

$\forall$i,j:$\mathbb{Z}$.

1. $i>j \quad\Rightarrow$ empty?($\{i..j\}$)
2. $\{i..i\} = \{i\}$
3. $\{i+1..j\} = \{i..j\}-i$
4. $i-1\leq j \ \Rightarrow\ \{i-1..j\} = \{i..j\}+(i-1)$
5. $\{i..j-1\} = \{i..j\}-j$
6. $i\leq j+1 \ \Rightarrow\ \{i..j+1\} = \{i..j\}+(j+1)$
7. $\{i+1..j+1\} = \{k+1\,|\,k\in\{i..j\}\}$

Lemma B.1.15 General Set Former

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$S,S':Set($\alpha$).$\forall$a:$\alpha$.$\forall$FAM:Set(Set($\alpha$)).$\forall$f,f':$\alpha\to\beta$.$\forall$p,q:$\alpha\to\mathbb{B}$.$\forall$g:$\beta\to\gamma$.$\forall$q:$\beta\to\mathbb{B}$.

1. $\{f(x)|x\in\emptyset \wedge p(x)\} = \emptyset$
2. $\neg p(a) \Rightarrow \{f(x)|x\in S+a \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\}$
3. $p(a) \Rightarrow \{f(x)|x\in S+a \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\}+f(a)$
4. $\neg p(a) \Rightarrow \{f(x)|x\in\{a\} \wedge p(x)\} = \emptyset$
5. $p(a) \Rightarrow \{f(x)|x\in\{a\} \wedge p(x)\} = \{f(a)\}$
6. $\{g(y)\,|\,y\in\{f(x)|x\in S\wedge p(x)\} \wedge q(y)\} = \{g(f(x))|x\in S\wedge p(x)\wedge q(f(x))\}$
7. $\neg p(a) \Rightarrow \{f(x)|x\in S-a \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\}$
8. $p(a) \Rightarrow \{f(x)|x\in S-a \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\}-f(a)$
9. $\{f(x)|x\in S\cup S' \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\}\cup\{f(x)|x\in S' \wedge p(x)\}$
10. $\{f(x)|x\in S\cap S' \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\}\cap_\beta\{f(x)|x\in S' \wedge p(x)\}$
11. $\{f(x)|x\in S\setminus S' \wedge p(x)\} = \{f(x)|x\in S \wedge p(x)\} \setminus \{f(x)|x\in S' \wedge p(x)\}$
12. $\{f(x)|x\in\bigcup FAM \wedge p(x)\} = \bigcup\{\{f(x)|x\in S \wedge p(x)\}\,|\,S\in FAM\}$
13. $\{f(x)|x\in\bigcap FAM \wedge p(x)\} = \bigcap\{\{f(x)|x\in S \wedge p(x)\}\,|\,S\in FAM\}$
14. $\{x|x\in S \wedge x\neq a\} = S-a$
15. $\{x|x\in S \wedge x\in S'\} = S\cap S'$
16. $\{x|x\in S \wedge x\notin S'\} = S\setminus S'$
17. $\forall x\in S.p(x)\Rightarrow f(x)=f'(x) \Rightarrow \{f(x)|x\in S\wedge p(x)\}=\{f'(x)|x\in S\wedge p(x)\}$
18. $\forall x\in S.\,p(x)\Leftrightarrow q(x) \Rightarrow \{f(x)|x\in S\wedge p(x)\}=\{f(x)|x\in S\wedge q(x)\}$

Lemma B.1.16 Cardinality

$\forall\alpha,\beta$:TYPES.$\forall$S,S',T:Set(α).$\forall$a:α.$\forall$f:$\alpha\to\beta$.$\forall$i,j:$\mathbb{Z}$.

1. $|\emptyset| = 0$
2. $a\notin S \Rightarrow |S+a| = |S|+1$
3. $|\{a\}| = 1$
4. $S\subseteq S' \Rightarrow |S|\leq|S'|$
5. $i\leq j \Rightarrow |\{i..j\}| = j-i+1$
6. $a\in S \Rightarrow |S-a| = |S|-1$
7. $T=S\uplus S' \Rightarrow |T| = |S|+|S'|$
8. $|map(f,S)| = |S|$

Lemma B.1.17 Element Deletion

$\forall\alpha,\beta$:TYPES.$\forall$S,S':Set(α).$\forall$a,a':α.

1. $\emptyset-a = \emptyset$
2. $a\notin S \Rightarrow ((S+a)-a = S)$
3. $a\neq a' \Rightarrow ((S+a)-a' = (S-a')+a)$
4. $\{a\}-a = \emptyset$
5. $a\neq a' \Rightarrow \{a\}-a' = \{a\}$
6. $a\notin S \Rightarrow S-a = S$

Lemma B.1.18 Union

$\forall\alpha,\beta$:TYPES.$\forall$S,S',S'':Set(α).$\forall$a,a':α.$\forall$FAM,FAM':Set(Set(α)).$\forall$f:$\alpha\to\beta$.$\forall$p,q:$\alpha\to\mathbb{B}$.$\forall$i,j,k:$\mathbb{Z}$.

1. $S\cup\emptyset = S$
2. $S\cup(S'+a) = (S\cup S')+a$
3. $S\cup\{a\} = S+a$
4. $i\leq j\wedge j<k \Rightarrow \{i..j\}\cup\{j+1..k\} = \{i..k\}$
5. $\{f(x)|x\in S\wedge p(x)\} \cup \{f(x)|x\in S\wedge q(x)\} = \{f(x)|x\in S\wedge p(x)\vee q(x)\}$
6. $S\cup(S'\cap S'') = (S\cup S')\cap(S\cup S'')$
7. $\bigcup FAM \cup \bigcup FAM' = \bigcup(FAM\cup FAM')$
8. $S\cup S = S$
9. $S\cup S' = S'\cup S$
10. $S\cup(S'\cup S'') = (S\cup S')\cup S''$
11. $S\cup(S\cap S') = S$

Lemma B.1.19 Intersection

$\forall\alpha,\beta$:TYPES.$\forall$S,S',S'':Set(α).$\forall$a,a':α.$\forall$FAM,FAM':Set(Set(α)).$\forall$f:$\alpha\to\beta$.$\forall$p,q:$\alpha\to\mathbb{B}$.$\forall$i,j,k:$\mathbb{Z}$.

1. S$\cap\emptyset$ = $\emptyset$
2. a$\in$S $\Rightarrow$ (S$\cap$(S'+a) = (S$\cap$S')+a)
3. a$\notin$S $\Rightarrow$ S$\cap$(S'+a) = S$\cap$S'
4. a$\in$S $\Rightarrow$ (S$\cap${a} = {a})
5. a$\notin$S $\Rightarrow$ S$\cap${a} = $\emptyset$
6. (i$\leq$j $\wedge$ j$\leq$k $\wedge$ k$\leq$j') $\Rightarrow$ {i..k}$\cap${j..j'} = {j..k}
7. {f(x)|x$\in$S $\wedge$ p(x)} $\cap$ {f(x)|x$\in$S $\wedge$ q(x)} = {f(x)|x$\in$S $\wedge$ p(x) $\wedge$ q(x)}
8. S$\cap$(S'$\cup$S'') = (S$\cap$S')$\cup$(S$\cap$S'')

8a S$\cap$(S'-a) = (S$\cap$S')-a

9. $\bigcap$FAM $\cap$ $\bigcap$FAM' = $\bigcap$(FAM$\cup$FAM')
10. S$\cap$S = S
11. S$\cap$S' = S'$\cap$S
12. S$\cap$(S'$\cap$S'') = (S$\cap$S')$\cap$S''
13. S$\cap$(S$\cup$S') = S

Lemma B.1.20 Set Difference

$\forall\alpha,\beta$:TYPES.$\forall$S,S',S'':Set(α).$\forall$a,a':α.$\forall$FAM,FAM':Set(Set(α)).$\forall$f:$\alpha\to\beta$.$\forall$p,q:$\alpha\to\mathbb{B}$.$\forall$i,j,k:$\mathbb{Z}$.

1. S\$\emptyset$ = S
2. S\(S'+a) = (S\S')-a
3. S\{a} = S-a
4. $\emptyset$\S' = $\emptyset$
5. a$\notin$S' $\Rightarrow$ ((S+a)\S' = (S\S')+a)
6. a$\in$S' $\Rightarrow$ ((S+a)\S' = S\S')
7. a$\notin$S $\Rightarrow$ ({a}\S = {a})
8. a$\in$S $\Rightarrow$ ({a}\S = $\emptyset$)
9. {f(x)|x$\in$S $\wedge$ p(x)} \ {f(x)|x$\in$S $\wedge$ q(x)} = {f(x)|x$\in$S $\wedge$ p(x) $\wedge\neg$q(x)}
10. S\(S'$\cup$S'') = (S\S')\S''
11. S\S = $\emptyset$
12. S$\subseteq$S' $\Rightarrow$ S\S' = $\emptyset$

Lemma B.1.21 Union of a family of sets

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$FAM:Set(Set($\alpha$)).$\forall$S:Set($\alpha$).$\forall$T:Set($\beta$).$\forall$F:$\alpha\to$Set($\beta$).$\forall$g:$\beta\to\gamma$.
$\forall$p:$\alpha\to\mathbb{B}$.$\forall$q:$\beta\to\mathbb{B}$.$\forall$S:Set(α).$\forall$c:γ.

1. $\bigcup\emptyset$ = $\emptyset$
2. $\bigcup$(FAM+S) = $\bigcup$FAM$\cup$S
3. $\bigcup$({S}) = S
4. $\bigcup${ {g(y)|y$\in$F(x)$\wedge$q(y)} |x$\in$S $\wedge$ p(x)} = {g(y)|y$\in\bigcup${F(x)|x$\in$S $\wedge$ p(x)} $\wedge$ q(y)})
5. c$\in\bigcup${ {g(y)|y$\in$F(x) $\wedge$ q(y)} | x$\in$S $\wedge$ p(x)} $\Leftrightarrow$ $\exists$x$\in$S. p(x) $\wedge$ $\exists$y$\in$F(x). q(y) $\wedge$ c=g(y)
6. S$\in$FAM $\Rightarrow$ $\bigcup$(FAM-S) = $\bigcup$FAM\S
7. $\bigcup${ $\bigcup${f(x,y)|x$\in$S$\wedge$p(x)}|y$\in$T $\wedge$ q(y)} = $\bigcup${ $\bigcup${f(x,y)|y$\in$T$\wedge$q(y)}|x$\in$S $\wedge$ p(x)}
8. $\bigcup${ {G(y)|y$\in$F(x)$\wedge$q(y)} |x$\in$S $\wedge$ p(x)} = { $\bigcup${G(y)|y$\in$F(x)$\wedge$q(y)} |x$\in$S $\wedge$ p(x)}

Lemma B.1.22 Intersection of a family of sets

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$F:$\alpha\to$Set($\beta$).$\forall$FAM:Set(Set($\alpha$)).$\forall$g:$\beta\to\gamma$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$q:$\beta\to\mathbb{B}$.$\forall$S:Set($\alpha$).

1. $\bigcap${S} = S
2. $\bigcap$(FAM+S) = $\bigcap$FAM$\cap$S
3. $\bigcap${ {g(y)|y$\in$F(x)$\wedge$q(y)} |x$\in$S $\wedge$ p(x)} = {g(y)|x$\in\bigcap${F(x)|x$\in$S $\wedge$ p(x)} $\wedge$ q(y)})

Lemma B.1.23 Arbitrary Selection

$\forall\alpha$:TYPES.$\forall$a:α.

1. arb({a}) = a

Lemma B.1.24 map-operation

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$f:$\alpha\to$Set($\beta$).$\forall$g:$\beta\to\gamma$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$S,S':Set($\alpha$).$\forall$a:$\alpha$.

1. map(f,$\emptyset$) = $\emptyset$
2. map(f,S+a) = map(f,S)+f(a)
3. map(f,{a}) = {f(a)}
4. map(g,{f(x)|x$\in$S $\wedge$ p(x)}) = {g(f(x))|x$\in$S $\wedge$ p(x)})
5. map(f,S-a) = map(f,S)-f(a)
6. map(f,S$\cup$S') = map(f,S)$\cup$map(f,S')
7. map(f,S$\cap$S') = map(f,S)$\cap$map(f,S')
8. map(f,S\S') = map(f,S)\map(f,S')

Lemma B.1.25 Reduce operation

$\forall\alpha$:TYPES.$\forall$bop:$\alpha^2\to\alpha$.$\forall$S:Set(α).$\forall$a:α.$\forall$FAM:Set(Set(α)).

1. reduce(bop,{a}) = a
2. $\neg$empty?(S) $\wedge$ a$\notin$S $\Rightarrow$ reduce(bop,S+a) = bop(reduce(bop,S),a)
3. $\neg$empty?(FAM) $\Rightarrow$ reduce($\cup$,FAM) = $\bigcup$FAM
4. $\neg$empty?(FAM) $\Rightarrow$ reduce($\cap$,FAM) = $\bigcap$FAM

B.2 Endliche Folgen

Der Typ der endlichen Mengen ist ein generischer Datentyp, der auf der Basis der Konzepte `Seq, =, [], cons, first` und`rest` eingeführt werden kann. Bis auf kleine Erweiterungen entspricht der dem NuPRL Listenkonstruktor.

BASISKONZEPTE

Definition B.2.1 (Implementierung endlicher Folgen)

`[]`	$\equiv$ `nil`
`cons`	$\equiv$ λ`a,L. (a.L)`
`list_ind(`L`;`t_b`;`a`,`L'`,`$FL'.t_{ind}$`)`	$\equiv$ `list_ind(`L`;`t_b`;`a`,`L'`,`$FL'.t_{ind}$`)`
`first`	$\equiv$ λ`L.list_ind(L;`∞`; a,_,_. a)`
`rest`	$\equiv$ λ`L.list_ind(L; []; _,L',_. L')`
`=`	$\equiv$ λ`L,L'.(list_ind(L;`λ`L.list_ind(L; true; _,_,_. false);` `a,_,EQ.`λ`L1.a=first(L1)`$\wedge$`EQ(rest(L1))) (L'))`
`Seq(`α`)`	$\equiv$ α `list`

Lemma B.2.2 Axioms of Finite Sequences

$\forall\alpha$`:TYPES.`$\forall$`L,L':Seq(`α`).`$\forall$`a,a':`α`. `$\forall$`P:PROP(Seq(`α`)).`

1. `Seq(`α`)` $\in$ `TYPES`
2. `[]` $\in$ `Seq(`α`)`
3. `cons` $\in$ $\alpha\times$`Seq(`α`)`$\rightarrow$`Seq(`α`)`
4. `first` $\in$ `Seq(`α`)`$\not\rightarrow\alpha$
5. `rest` $\in$ `Seq(`α`)`$\rightarrow$`Seq(`α`)`
6. `[]`$\neq$`a.L`
7. `a.L=a'.L'` $\Leftrightarrow$ `(a=a'` $\wedge$ `L=L' )`
8. `first(a.L)=a`
9. `rest(a.L)=L`
10. $\forall$`g:`β`.`$\forall$`h:(`$\beta\times$`Seq(`α`)`$\times\alpha$`)`$\rightarrow\beta$`.`$\exists$`f:Seq(`α`)`$\rightarrow\beta$`.`
`f([])=g` $\wedge$ $\forall$`L:Seq(`α`).`$\forall$`a:`α`. f(a.L)=h(f(L),L,a)`
11. `(P([])` $\wedge$ $\forall$`L:Seq(`α`).`$\forall$`a:`α`.P(L)`$\Rightarrow$`P(a.L))` $\Rightarrow$ $\forall$`L:Seq(`α`).P(L)`

Lemma B.2.3 Finite Constructability and Sequence Equality

$\forall\alpha$`:TYPES.`$\forall$`L,L':Seq(`α`).`

1. $\forall\alpha$`:TYPES.`$\forall$`L:Seq(`α`).` `L=[]` $\vee$ $\exists$`a:`α`.`$\exists$`L':Seq(`α`). L=a.L'`
2. `L=L'` $\Leftrightarrow$ `first(L)=first(L')` $\wedge$ `rest(L)=rest(L')`

ABGELEITETE KONZEPTE

Definition B.2.4 (Sequence Notation)

`if` L`=[] then` t `else` t'	$\equiv$ `list_ind(`L`; `t`; _,_,_,`t'`)`
`let` L`=`$a.L'$ $\in exp$	$\equiv$ `list_ind(`L`; `∞`; `a`,`L'`,_,`exp`)`
`let` L`=[`a`]` $\in exp$	$\equiv$ `let` L`=`a`.L'` $\in$ `if L'=[] then` exp `else` ∞
`let` L`=[`a`,`a'`]` $\in exp$	$\equiv$ `let` L`=`a'`.L'` $\in$ `let L'=[`a`]` $\in exp$
`let` L`=[`a`,`a'`,`a''`]` $\in exp$	$\equiv$ `let` L`=`a''`.L'` $\in$ `let L'=[`a`,`a'`]` $\in exp$
λ`[`a`].`exp	$\equiv$ λ`L.let L=[`a`]` $\in exp$
λ`[`a`,`a'`].`exp	$\equiv$ λ`L.let L=[`a`,`a'`]` $\in exp$
λ`[`a`,`a'`,`a'`].`exp	$\equiv$ λ`L.let L=[`a`,`a'`,`a'`]` $\in exp$

Definition B.2.5 (Sequence Operations)

null?	$\equiv \lambda$L. list_ind(L; true; _,_,_.false)
[*list-exp*]	$\equiv$ *list-exp*.nil
$[i..j]$	$\equiv$ ind(j-i; _,_.[]; [j]; diff,j-seq. (j-diff).j-seq)
$[f_x \mid x \in L \wedge p_x]$	$\equiv$ list_ind(L; []; a,_,GSF. if p_x[a/x] then f_x[a/x].GSF else GSF)
$[f_x \mid x \in L]$	$\equiv [f_x \mid x \in L \wedge$ true$]$
$\lvert L \rvert$	$\equiv$ list_ind(L; 0; _,_,card. card+1)
$L[i]$	$\equiv$ list_ind(L; λj.∞; a,_,jth-of. λj. if j=1 then a else jth-of(j-1)) (i)
last	$\equiv \lambda$L. L[|L|]
$\cdot$	$\equiv \lambda$L,a. list_ind(L; [a]; a',_.app. a'.app)
ins	$\equiv \lambda$L,j,a. [if i<j then L[i] else if i=j then a else L[i-1] | i $\in$ [1..|L|+1]]
del	$\equiv \lambda$L,j. [if i<j then L[i] else L[i+1] | i $\in$ [1..|L|-1]]
$\circ$	$\equiv \lambda$L,L'. list_ind(L; L'; a,_,conc. a.conc)
rev	$\equiv \lambda$L. [L[|L|-i] | i $\in$ [0..|L|-1]]
domain	$\equiv \lambda$L. {1..|L|}
range	$\equiv \lambda$L. {L[i]|i $\in$ domain(L)}
map	$\equiv \lambda$f,L. [f(x)|x $\in$ L]
reduce	$\equiv \lambda$op,L. list_ind(L; ∞; a,L',redL'. if null?(L') then a else op(redL',a))
$L_{[i..j]}$	$\equiv$ [L[k] | k $\in$ [$i..j$]]
$\in$	$\equiv \lambda$a,L. $\exists$x $\in$ range(L).x=a
$\sqsubseteq$	$\equiv \lambda$L,L'. |L| $\leq$ |L'| $\wedge$ $\forall$i $\in$ domain(L). L[i]=L'[i]
$L_{<g}$	$\equiv$ [x|x $\in L \wedge$ x<g]
$L_{\geq g}$	$\equiv$ [x|x $\in L \wedge$ x$\geq g$]
find	$\equiv \lambda$g,L. min{j|j $\in$ domain(L) $\wedge$ L[j]=g}
-	$\equiv \lambda$L,g.del(L,find(g,L))
nodups	$\equiv \lambda$L. $\forall$i $\in$ domain(L).$\forall$j $\in$ {i+1..|L|}. L[i]$\neq$L[j]
perm	$\equiv \lambda$L,S. nodups(L) $\wedge$ range(L)=S
rearranges	$\equiv \lambda$L,L'. $\exists$I:Seq($\mathbb{Z}$).perm(I,domain(L)) $\wedge$ L'=[L[k] | k $\in$ I]
insert	$\equiv \lambda$L,g.$L_{<g} \circ$ g.L
Seq$^+(\alpha)$	$\equiv$ {L:Seq(α) | $\neg$empty?(L)}
arb	$\equiv \lambda$L. first(L)

Lemma B.2.6 Operation Signatures

$\forall \alpha, \beta$:TYPES.$\forall$f:$\alpha \to \beta$.$\forall$p:$\alpha \to \mathbb{B}$.

1. null? $\in$ Seq(α)$\to\mathbb{B}$
2. $\in$ $\in \alpha \times$Seq(α)$\to\mathbb{B}$
3. $\sqsubseteq$ $\in$ Seq(α)$^2\to\mathbb{B}$
4. λi,j. [i..j] $\in \mathbb{Z}^2\to$Seq($\mathbb{Z}$)
5. λL. [f(x)\|x$\in$L$\wedge$p(x)] $\in$ Seq(α)$\to$Seq(β)
6. λL. \|L\| $\in$ Seq(α)$\to\mathbb{N}$
7. λL,i.L[i] $\in$ Seq(α)$\times\mathbb{Z}\to\alpha$
8. last $\in$ Seq(α)$\to\alpha$
9. $\cdot$ $\in$ Seq(α)$\times\alpha\to$Seq(α)
10. ins $\in$ Seq(α)$\times\mathbb{N}\times\alpha\to$Seq($\alpha$)
11. del $\in$ Seq(α)$\times\mathbb{N}\to$Seq(α)
12. $\circ$ $\in$ Seq(α)$^2\to$Seq(α)
13. rev $\in$ Seq(α)$\to$Seq(α)
14. domain $\in$ Seq(α)$\to$Set(α)
15. range $\in$ Seq(α)$\to$Set(α)
16. map $\in (\alpha\to\beta)\times$Seq($\alpha$)$\to$Seq($\beta$)
17. reduce $\in (\alpha^2\to\alpha)\times$Seq($\alpha$) $\not\to \alpha$
18. λL,i,j. $L_{[i..j]}$ $\in$ Seq(α)$\times\mathbb{Z}^2\to$Seq(α)
19. nodups $\in$ Seq(α)$\to\mathbb{B}$
20. perm $\in$ Seq(α)$\times$Set(α)$\to\mathbb{B}$
21. rearranges $\in$ PROP(Seq(α)^2)

Lemma B.2.7 Prepend

$\forall\alpha$:TYPES.$\forall$L:Seq(α).$\forall$a:α.

1. a.L$\neq$[]
2. a.L$\neq$L

Lemma B.2.8 null?

$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{N}$.

1. null?(L) $\Leftrightarrow$ L=[]
2. L$\sqsubseteq$L' $\Rightarrow$ null?(L')$\Rightarrow$null?(L)
3. $\neg$null?(a.L)
4. $\neg$null?([a])
5. null?([f(x)|x$\in$L$\wedge$p(x)]) $\Leftrightarrow$ null(L) $\vee\forall$x$\in$L.$\neg$p(x))
6. null?(L) $\Rightarrow$ |L|=0
7. $\neg$null?(L$\cdot$a)
8. i$\leq$|L|+1 $\Rightarrow$ $\neg$null?(ins(L,i,a))
9. null?(del(L,i)) $\Leftrightarrow$ null?(L) $\vee$ L=[L[i]]
10. null?(L$\circ$L') $\Leftrightarrow$ null?(L)$\wedge$null?(L')
11. null?(rev(L)) $\Leftrightarrow$ null?(L)
12. null?(L) $\Leftrightarrow$ empty?(domain(L))
13. null?(L) $\Leftrightarrow$ empty?(range(L))

Lemma B.2.9 Membership

$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$L,L':Seq(α).$\forall$a,a':α.$\forall$b:β.$\forall$i,j,k:$\mathbb{Z}$.

1. a$\notin$[]
2. a'$\in$a.L $\Leftrightarrow$ a'=a $\vee$ a'$\in$L
3. a$\in$L $\Leftrightarrow$ a=first(L) $\vee$ a$\in$rest(L)
4. L$\sqsubseteq$L' $\Rightarrow$ a$\in$L $\Rightarrow$ a$\in$L'
5. a'$\in$[a] $\Leftrightarrow$ a=a'
6. k$\in$[i..j] $\Leftrightarrow$ i$\leq$k $\wedge$ k$\leq$j
7. b$\in$[f(x)|x$\in$L$\wedge$p(x)] $\Leftrightarrow$ $\exists$x$\in$L. p(x) $\wedge$ b=f(x)
8. a'$\in$L$\cdot$a $\Leftrightarrow$ a'=a $\vee$ a'$\in$L
9. i$\leq$|L|+1 $\Rightarrow$ a'$\in$ins(L,i,a) $\Leftrightarrow$ a'=a $\vee$ a'$\in$L
10. 0<i $\wedge$ i$\leq$|L| $\wedge$ a$\neq$L[i] $\Rightarrow$ a$\in$del(L,i)$\Leftrightarrow$ a'$\in$L
11. a$\in$L$\circ$L' $\Leftrightarrow$ a$\in$L $\vee$ a$\in$L'
12. a$\in$rev(L) $\Leftrightarrow$ a$\in$L
13. i$\in$domain(L) $\Leftrightarrow$ 1$\leq$i $\wedge$ i$\leq$|L|
14. a$\in$L $\Leftrightarrow$ a$\in$range(L)

Lemma B.2.10 Prefix

$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$L,L':Seq(α).$\forall$a,a':α.$\forall$b:β.$\forall$i,j,k,l:$\mathbb{Z}$.

1. []$\sqsubseteq$L
2. L$\sqsubseteq$[] $\Leftrightarrow$ null?(L)
3. a.L$\sqsubseteq$L' $\Leftrightarrow$ a=first(L') $\wedge$ L$\sqsubseteq$rest(L')
4. L$\sqsubseteq$L' $\Rightarrow$ first(L)=first(L')
5. L$\sqsubseteq$L' $\Rightarrow$ rest(L)$\sqsubseteq$rest(L')
6. [a]$\sqsubseteq$L $\Leftrightarrow$ a=first(L)
7. [i..j]$\sqsubseteq$[k..l] $\Leftrightarrow$ i=j $\wedge$ j$\leq$l
8. L$\sqsubseteq$L' $\Rightarrow$ [f(x)|x$\in$L$\wedge$p(x)]$\sqsubseteq$[f(x)|x$\in$L'$\wedge$p(x)]
9. L$\sqsubseteq$L$\cdot$a
10. L$\sqsubseteq$L$\circ$L'
11. L$\sqsubseteq$L' $\Leftrightarrow$ $\exists$L'':Seq(α). L$\circ$L''=L'
12. L$\sqsubseteq$L' $\Rightarrow$ domain(L)$\subseteq$domain(L')
13. L$\sqsubseteq$L' $\Rightarrow$ range(L)$\subseteq$range(L')
14. L$\sqsubseteq$L
15. L$\sqsubseteq$L' $\wedge$ L'$\sqsubseteq$L'' $\Rightarrow$ L$\sqsubseteq$L''
16. L$\sqsubseteq$L' $\wedge$ L'$\sqsubseteq$L $\Leftrightarrow$ L=L'
17. i$\leq$|L| $\wedge$ L$\sqsubseteq$L' $\Rightarrow$ L[i]=L'[i]

Lemma B.2.11 Integer sequence

$\forall$i,j:$\mathbb{Z}$.

1. i>j $\Rightarrow$ null?([i..j])
2. [i..i] = [i]
3. i$\leq$j $\Rightarrow$ first([i..j]) = i
4. [i+1..j] = rest([i..j])
5. i-1$\leq$j $\Rightarrow$ [i-1..j] = (i-1).[i..j]
6. i$\leq$j+1 $\Rightarrow$ [i..j+1] = [i..j]$\cdot$(j+1)
7. [i+1..j+1] = [k+1 | k$\in$[i..j]]
8. i$\leq$j $\Rightarrow$ last([i..j]) = j

Lemma B.2.12 General sequence former

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$L,L':Seq($\alpha$).$\forall$a:$\alpha$.$\forall$f,f':$\alpha\to\beta$.$\forall$g:$\beta\to\gamma$.$\forall$p,p':$\alpha\to\mathbb{B}$.$\forall$q:$\beta\to\mathbb{B}$.$\forall$i:$\mathbb{N}$.

1. [f(x)|x$\in$[]$\wedge$p(x)] = []
2. $\neg$p(a) $\Rightarrow$ [f(x)|x$\in$a.L$\wedge$p(x)] = [f(x)|x$\in$L$\wedge$p(x)]
3. p(a) $\Rightarrow$ [f(x)|x$\in$a.L$\wedge$p(x)] = f(a).[f(x)|x$\in$L$\wedge$p(x)]
4. p((first(L)) $\Rightarrow$ first([f(x)|x$\in$L$\wedge$p(x)]) = f(first(L))
5. p((first(L)) $\Rightarrow$ rest([f(x)|x$\in$L$\wedge$p(x)]) = [f(x)|x$\in$rest(L)$\wedge$p(x)]
6. $\neg$p(a) $\Rightarrow$ [f(x)|x$\in$[a]$\wedge$p(x)] = []
7. p(a) $\Rightarrow$ [f(x)|x$\in$[a]$\wedge$p(x)] = [f(a)]
8. [g(y) | y$\in$[f(x)|x$\in$L$\wedge$p(x)] $\wedge$ q(y)] = [g(f(x))|x$\in$L$\wedge$p(x)$\wedge$q(f(x))]
9. $\forall$a:α. $\neg$p(a) $\Rightarrow$ [f(x)|x$\in$L$\cdot$a$\wedge$p(x)] = [f(x)|x$\in$L$\wedge$p(x)]
10. p(a) $\Rightarrow$ [f(x)|x$\in$L$\cdot$a$\wedge$p(x)] = [f(x)|x$\in$L$\wedge$p(x)]$\cdot$f(a)
11. i$\leq$|L|+1 $\Rightarrow$ $\neg$p(a) $\Rightarrow$ [f(x)|x$\in$ins(L,i,a)$\wedge$p(x)] = [f(x)|x$\in$L$\wedge$p(x)]
12. $\neg$p(L[i]) $\Rightarrow$ [f(x)|x$\in$del(L,i)$\wedge$p(x)] = [f(x)|x$\in$L$\wedge$p(x)]
13. [f(x)|x$\in$L$\circ$L'$\wedge$p(x)] = [f(x)|x$\in$L$\wedge$p(x)]$\circ$[f(x)|x$\in$L'$\wedge$p(x)]
14. [f(x)|x$\in$rev(L)$\wedge$p(x)] = rev([f(x)|x$\in$L$\wedge$p(x)])
15. $\forall$x$\in$L.p(x)$\Rightarrow$f(x)=f'(x) $\Rightarrow$ [f(x)|x$\in$L$\wedge$p(x)]=[f'(x)|x$\in$L$\wedge$p(x)]
16. $\forall$x$\in$L. p(x)$\Leftrightarrow$p'(x) $\Rightarrow$ [f(x)|x$\in$L$\wedge$p(x)]=[f(x)|x$\in$L$\wedge$p'(x)]

Lemma B.2.13 Cardinality

$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i,j:$\mathbb{Z}$.

1. |[]| = 0
2. |a.L| = |L|+1
3. $\neg$null?(L) $\Rightarrow$ |rest(L)| = |L|-1
4. |[a]| = 1
5. L$\sqsubseteq$L' $\Rightarrow$ |L|$\leq$|L'|
6. |[f(x)|x$\in$L$\wedge$p(x)]|=|[x|x$\in$L$\wedge$p(x)]|
7. $\forall$i,j:$\mathbb{Z}$. i$\leq$j $\Rightarrow$ |[i..j]| = j-i+1
8. |L$\cdot$a| = |L|+1
9. i$\leq$|L|+1 $\Rightarrow$ |ins(L,i,a)| = |L|+1
10. 0<i $\wedge$ i$\leq$|L| $\Rightarrow$ |del(L,i)| = |L|-1
11. |L$\circ$L'| = |L|+|L'|
12. |rev(L)| = |L|
13. |L| = |domain(L)|

Lemma B.2.14 Selecting the i-th element

$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i,j:$\mathbb{N}$.

1. L[1]=first(L)
2. L[i+1]=rest(L)[i]
3. ins(L,i,a)[i]=a
4. [f(x)|x$\in$L][i]=f(L[i])
5. i$\leq$|L| $\Rightarrow$ L·a[i]=L[i]
6. L·a[|L|+1]=a
7. i<j $\Rightarrow$ ins(L,j,a)[i]=L[i]
8. i>j $\Rightarrow$ ins(L,j,a)[i]=L[i-1]
9. i<j $\Rightarrow$ del(L,j)[i]=L[i]
10. i$\geq$j $\Rightarrow$ del(L,j)[i]=L[i+1]
11. i$\leq$|L| $\Rightarrow$ L$\circ$L'[i]=L[i]
12. i>|L| $\Rightarrow$ L$\circ$L'[i]=L'[i-|L|]
13. i$\leq$|L| $\Rightarrow$ rev(L)[i]=L[|L|-i+1]

Lemma B.2.15 last

$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{N}$.

1. last[a]=a
2. last(L·a)=a
3. last([f(x)|x$\in$L])=f(last(L))
4. i$\leq$|L| $\Rightarrow$ last(ins(L,i,a))=last(L)
5. last(ins(L,|L|+1,a)=a
6. i<L $\Rightarrow$ last(del(L,i))=last(L)
7. last(del(L,|L|))=L[|L|-1]
8. $\neg$null?(L') $\Rightarrow$ last(L$\circ$L')=last(L')
9. null?(L') $\Rightarrow$ last(L$\circ$L')=last(L)
10. last(rev(L))=first(L)

Lemma B.2.16 Append

$\forall\alpha$:TYPES.$\forall$L:Seq(α).$\forall$a,a':α.

1. []·a=[a]
2. (a'.L)·a=a'.(L·a)

Lemma B.2.17 Insert

$\forall\alpha$:TYPES.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{N}$.

1. ins(L,1,a)=a.L
2. ins(L,i+1,a)=first(L).ins(rest(L),i,a)
3. ins(L,|L|+1,a)=L·a
4. i$\leq$|L| $\Rightarrow$ ins(del(L,i),i,L[i])=L
5. i$\leq$|L| $\Rightarrow$ ins(L$\circ$L',i,a)=ins(L,i,a)$\circ$L'
6. i>|L| $\Rightarrow$ ins(L$\circ$L',i,a)=L$\circ$ins(L',i-|L|,a)

Lemma B.2.18 Delete

$\forall\alpha$:TYPES.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{N}$.

1. del(L,1)=rest(L)
2. del(L,i+1)=first(L).del(rest(L),i)
3. del(L·a,|L|+1)=L
4. i$\leq$|L|+1 $\Rightarrow$ del(ins(L,i,a),i)=L
5. i$\leq$|L| $\Rightarrow$ del(L$\circ$L',i)=del(L,i)$\circ$L'
6. i>|L| $\Rightarrow$ del(L$\circ$L',i)=L$\circ$del(L',i-|L|)
7. i$\leq$|L| $\Rightarrow$ del(rev(L),i)=rev(del(L,|L|-i+1))

Lemma B.2.19 Concat
$\forall\alpha$:TYPES.$\forall$L,L',L'':Seq(α).$\forall$a:α.$\forall$i,j,k:$\mathbb{Z}$.
1. L$\circ$[]=L
2. []$\circ$L=L
3. (a.L)$\circ$L'=a.(L$\circ$L')
4. [a]$\circ$L=a.L
5. L$\circ$[a]=L$\cdot$a
6. i$\leq$j$\wedge$j<k $\Rightarrow$ [i..j]$\circ$[j+1..k]=[i..k]
7. L$\circ$(L'$\circ$L'')=(L$\circ$L')$\circ$L''

Lemma B.2.20 Reverse
$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\rightarrow\beta$.$\forall$p:$\alpha\rightarrow\mathbb{B}$.$\forall$L,L':Seq($\alpha$).$\forall$a:$\alpha$.
1. rev([])=[]
2. rev(a.L)=rev(L)$\cdot$a
3. rev([a]=[a]
4. rev([f(x)|x$\in$L$\wedge$p(x)])=[f(x)|x$\in$rev(L)$\wedge$p(x)]
5. rev(L$\cdot$a)=a.rev(L)
6. rev(L$\circ$L')=rev(L')$\circ$rev(L)
7. rev(rev(L))=L

Lemma B.2.21 Domain
$\forall\alpha$:TYPES.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{N}$.
1. domain([])=$\emptyset$
2. domain(a.L)=domain(L)+(|L|+1)
3. domain([a])=$\{1\}$
4. L$\sqsubseteq$L' $\Rightarrow$ domain(L)$\subseteq$domain(L')
5. domain(L$\cdot$a)=domain(L)+(|L|+1)
6. i$\leq$|L| $\Rightarrow$ domain(ins(L,i,a))=domain(L)+(|L|+1)
7. i$\leq$|L| $\Rightarrow$ domain(del(L,i))=domain(L)-|L|
8. domain(L$\circ$L') = domain(L)+domain(L')
9. domain(rev(L))=domain(L)

Lemma B.2.22 Range
$\forall\alpha,\beta$:TYPES.$\forall$f:$\alpha\rightarrow\beta$.$\forall$p:$\alpha\rightarrow\mathbb{B}$.$\forall$L,L':Seq($\alpha$).$\forall$a:$\alpha$.$\forall$i,j:$\mathbb{Z}$.
1. range([])=$\emptyset$
2. range(a.L)=range(L)+a
3. range([i..j])=$\{$i..j$\}$
4. L$\sqsubseteq$L' $\Rightarrow$ range(L)$\subseteq$range(L')
4. range([f(x)|x$\in$L$\wedge$p(x)])=$\{$f(x)|x$\in$range(L)$\wedge$p(x)$\}$
5. range(L$\cdot$a)=range(L)+a
6. i$\leq$|L|+1 $\Rightarrow$ range(ins(L,i,a))=range(L)+a
7. range(L$\circ$L') = range(L)$\cup$range(L')
8. range(rev(L)) = range(L)

Lemma B.2.23 Reduce
$\forall\alpha$:TYPES.$\forall$bop:$\alpha^2\rightarrow\alpha$.$\forall$L:Seq($\alpha$).$\forall$a:$\alpha$.
1. reduce(bop,[a]) = a
2. $\neg$null?(L) $\Rightarrow$ reduce(bop,a.L) = bop(reduce(bop,L),a)

Lemma B.2.24 Nodups

$\forall\alpha$:TYPES.$\forall$L,L':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{Z}$.

1. nodups([])
2. nodups(a.L) $\Leftrightarrow$ nodups(L) $\wedge$ a$\notin$L
3. L$\sqsubseteq$L' $\Rightarrow$ nodups(L') $\Rightarrow$ nodups(L)
4. nodups[i..j]
5. nodups(L) $\Leftrightarrow$ |L| = |range(L)|
6. nodups(L·a) $\Leftrightarrow$ nodups(L) $\wedge$ a$\notin$L
7. i$\leq$|L| $\Rightarrow$ nodups(ins(L,i,a)) $\Leftrightarrow$ nodups(L) $\wedge$ a$\notin$L
8. nodups(L$\circ$L') $\Leftrightarrow$ nodups(L) $\wedge$ nodups(L') $\wedge$ $\forall$x$\in$L.x$\notin$(L')
9. nodups(rev(L)) $\Leftrightarrow$ nodups(L)

Lemma B.2.25 Difference

$\forall\alpha$:TYPES.$\forall$L:Seq(α).$\forall$a:α.

1. []-a=[]
2. a.L-a=L
3. a$\in$L $\Rightarrow$ $\exists$k$\in$domain(L).L=$L_{[1..k-1]}\circ$a.$L_{[k+1..|L|]}$

Lemma B.2.26 Rearranges

$\forall\alpha,\beta$:TYPES.$\forall$L,L',S,S':Seq(α).$\forall$a:α.$\forall$i:$\mathbb{N}$.$\forall$f:$\alpha\to\beta$.$\forall$p:$\alpha\to\mathbb{B}$.$\forall$g:$\mathbb{Z}$

1. rearranges([],S) $\Leftrightarrow$ S=[]
2. rearranges(a.L,S) $\Leftrightarrow$ a$\in$S $\wedge$ rearranges(L,S-a)
3. rearranges(L,S) $\Rightarrow$ $\forall$x$\in$L.x$\in$S
4. rearranges(L,S) $\Leftrightarrow$ |L| = |S|
5. rearranges(L·a,S) $\Leftrightarrow$ a$\in$S $\wedge$ rearranges(L,S-a)
6. i$\leq$|L| $\Rightarrow$ rearranges(ins(L,i,a),S) $\Leftrightarrow$ a$\in$S $\wedge$ rearranges(L,S-a)
7. rearranges(L,S) $\wedge$ rearranges(L',S')) $\Rightarrow$ rearranges(L$\circ$L',S$\circ$S')
8. rearranges(L,S) $\Rightarrow$ domain(L)=domain(S)
9. rearranges(L,S) $\Rightarrow$ range(L)=range(S)
10. rearranges(L,rev(L))
11. rearranges(L,L)
12. rearranges(L,S) $\Leftrightarrow$ rearranges(S,L)
13. rearranges(L,L') $\wedge$rearranges(L',S)) $\Rightarrow$ rearranges(L,S)
14. rearranges($L_{<g}\circ L_{\geq g}$,L)
15. rearranges(L,S) $\Leftrightarrow$ $\forall$a$\in$L. a$\in$S $\wedge$ rearranges(L-a,S-a)
16. rearranges(L,S) $\Rightarrow$ rearranges([f(x)|x$\in$L$\wedge$p(x)],[f(x)|x$\in$S$\wedge$p(x)])

B.3 Endliche Abbildungen

Der Typ der endlichen Abbildungen ist ein generischer Datentyp, der auf der Basis der Konzepte `Map, =, {| |}, extend, apply` und `domain` eingeführt werden kann. Die einfachste Form einer Implementierung ist die Verwendung von Listen von Paaren (Tabellen).

BASISKONZEPTE

Definition B.3.1 (Theory Implementation of Finite Maps)

{| |}	$\equiv$ []
$\mapsto ab$	$\equiv \langle a,b\rangle$
extend	$\equiv \lambda$M,a,b. ($\mapsto$ab.M)
mapind(M; t_b; a,b,M',$FM'.t_{ind}$)	$\equiv$ list_ind(M; t_b; ab,M',FM'. let ab=$\langle a,b\rangle$ in t_{ind})
apply	$\equiv \lambda$M,y.mapind(M; ∞; a,b,M',appM'. λx.if x=a then b else appM'(x)) (y)
$M(a)$	$\equiv$ apply(M,a)
domain	$\equiv \lambda$M. {x.1|x$\in$range(M)}
=	$\equiv \lambda$M,M'. domain(M)=domain(M') $\wedge$ $\forall$x$\in$domain(M).M(x)=M'(x)
Map(α,β)	$\equiv$ M,M':Seq($\alpha\times\beta$) // M=M'

Lemma B.3.2 Axioms of Finite Maps

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$M,M':Map($\alpha$,$\beta$).$\forall$a,a':$\alpha$.$\forall$b:$\beta$.$\forall$P:PROP(Map($\alpha$,$\beta$)).

1. Map(α,β) $\in$ TYPES
2. {\| \|} $\in$ Map(α,β)
3. apply $\in$ Map(α,β)$\times\alpha\not\to\beta$
4. extend $\in$ Map(α,β)$\times\alpha\times\beta\to$Map($\alpha$,$\beta$)
5. domain $\in$ Map(α,β)$\to$Set(α)
6. extend(M,a,b)(a)=b
7. a'$\neq$a $\Rightarrow$ extend(M,a,b)(a')=M(a')
8. domain({\| \|})=$\emptyset$
9. domain(extend(M,a,b))=domain(M)+a
10. M=M' $\Leftrightarrow$ domain(M)=domain(M') $\wedge$ $\forall$x$\in$domain(M).M(x)=M'(x)
11. $\forall$g:γ.$\forall$h:($\gamma\times$Map(α,β)$\times\alpha$)$\to\beta$.$\exists$f:Map(α,β)$\to\beta$.
    f({\| \|}) = g $\wedge$ $\forall$M:Map(α,β).$\forall$a:α.$\forall$b:β. a$\notin$domain(M) $\Rightarrow$ f(extend(M,a,b)) = h(f(M),M,a,b)
12. (P({\| \|}) $\wedge$ $\forall$M:Map(α,β).$\forall$a:α.$\forall$b:β.P(M)$\Rightarrow$P(extend(M,a,b))) $\Rightarrow$ $\forall$M:Map(α,β).P(M)

Lemma B.3.3 Finite Constructability

$\forall\alpha,\beta$:TYPES.$\forall$M:Map(α,β).

1. M={\| \|} $\vee$ $\exists$a:α.$\exists$b:β.$\exists$M':Map(α,β). a$\notin$domain(M) $\wedge$ M=extend(M,a,b)

ABGELEITETE KONZEPTE

Definition B.3.4 (Map Vocabulary)

{| *map-list-exp* |}	$\equiv$ *map-list-exp*.nil
let M=extend(M',a,b) in e	$\equiv$ mapind(M; ∞; a,b,M',_,e)
$\sqsubseteq$	$\equiv \lambda$M,M'. $\forall$x$\in$domain(M).M(x)=M'(x)
range	$\equiv \lambda$M. {M(x)|x$\in$domain(M)}
{|$\mapsto f_x g_x$|$x\in S\wedge p_x$|}	$\equiv$ setind(S; {| |}; a,_,GSF. if p_x[a/x] then extend(GSF,f_x[a/x],g_x[a/x]) else GSF)
{|$\mapsto f_x g_x$|$x\in S$|}	$\equiv$ {|$\mapsto f_x g_x$|$x\in S\wedge$true|}
$\circ$	$\equiv \lambda$M,M'. {|$\mapsto$xM'(M(x))|x$\in$domain(M) $\wedge$ M(x)$\in$domain(M')|}
|M|	$\equiv$ |domain(M)|

Lemma B.3.5 Operator Signatures

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$g:$\alpha\to\gamma$.$\forall$p:$\alpha\to$Bool.

1. $\sqsubseteq$ $\in$ Map(α,β)$^2\to$Bool
2. range $\in$ Map(α,β)$\to\beta$
3. λS. $\{|\mapsto$f(x)g(x)$|$x$\in$S $\wedge$ p(x)$|\}$ $\in$ Set(α)$\to$Map(β,γ)
4. $\circ$ $\in$ Map(α,β)$\times$Map(β,γ)$\to$Map(α,γ)
5. λM. |M| $\in$ Map(α,β)$\to\mathbb{N}$

Lemma B.3.6 extend

$\forall\alpha,\beta$:TYPES.$\forall$M:Map(α,β).$\forall$a:α.$\forall$b:β.

1. extend(M,a,b)$\neq\{|\,|\}$
2. M(a)=b $\Rightarrow$ extend(M,a,b)=M
3. M(a)$\neq$b $\Rightarrow$ extend(M,a,b)$\neq$M

Lemma B.3.7 Domain

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$g:$\alpha\to\gamma$.$\forall$p:$\alpha\to$Bool.$\forall$S:Set($\alpha$).$\forall$M:Map($\alpha$,$\beta$).$\forall$M':Map($\beta$,$\gamma$).

1. domain($\{|\mapsto$f(x)g(x)$|$x$\in$S$\wedge$p(x)$|\}$) = $\{$f(x)$|$x$\in$S$\wedge$p(x)$\}$
2. domain(M$\circ$M')=$\{$x$|$x$\in$domain(M)$\wedge$M(x)$\in$domain(M')$\}$
3. domain(M$\circ$M')$\subseteq$domain(M)

Lemma B.3.8 Range

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$g:$\alpha\to\gamma$.$\forall$p:$\alpha\to$Bool.$\forall$M:Map($\alpha$,$\beta$).$\forall$M':Map($\beta$,$\gamma$).$\forall$a:$\alpha$.$\forall$b:$\beta$.$\forall$S:Set($\alpha$).

1. range($\{|\,|\}$) = $\emptyset$
2. a$\notin$domain(M) $\Rightarrow$ range(extend(M,a,b))=range(M)+b
3. range($\{|\mapsto$f(x)g(x)$|$x$\in$S$\wedge$p(x)$|\}$)=$\{$g(x)$|$x$\in$S$\wedge$p(x)$\}$
4. range(M$\circ$M')=$\{$M'(M(x))$|$x$\in$domain(M)$\wedge$M(x)$\in$domain(M')$\}$
5. range(M$\circ$M')$\subseteq$range(M')

Lemma B.3.9 Submap

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$g:$\alpha\to\gamma$.$\forall$p:$\alpha\to$Bool.$\forall$M,M',M'':Map($\alpha$,$\beta$).$\forall$a:$\alpha$.$\forall$b:$\beta$.$\forall$S,S':Set($\alpha$).

1. $\{|\,|\}\sqsubseteq$M
2. M$\sqsubseteq\{|\,|\}$ $\Leftrightarrow$ M=$\{|\,|\}$
3. extends(M,a,b)$\sqsubseteq$M' $\Leftrightarrow$ M$\sqsubseteq$M' $\wedge$ a$\in$domain(M') $\wedge$ M'(a)=
4. a$\notin$domain(M') $\wedge$ M$\sqsubseteq$M' $\Rightarrow$ M$\sqsubseteq$extends(M',a,b)
5. S$\subseteq$S' $\Rightarrow$ $\{|\mapsto$f(x)g(x)$|$x$\in$S$\wedge$p(x)$|\}\sqsubseteq\{|\mapsto$f(x)g(x)$|$x$\in$S'$\wedge$p(x)$|\}$
6. M$\sqsubseteq$M' $\Rightarrow$ domain(M)$\subseteq$main(M')
7. M$\sqsubseteq$M' $\Rightarrow$ range(M)$\subseteq$range(M)
8. M$\sqsubseteq$M
9. M$\sqsubseteq$M' $\wedge$ M'$\sqsubseteq$M $\Leftrightarrow$ M=M'
10. M$\sqsubseteq$M' $\wedge$ M'$\sqsubseteq$M'' $\Rightarrow$ M$\sqsubseteq$M''

Lemma B.3.10 General Map Former

$\forall\alpha,\beta,\gamma$:TYPES.$\forall$f:$\alpha\to\beta$.$\forall$g:$\alpha\to\gamma$.$\forall$p:$\alpha\to$Bool.$\forall$S:Set($\alpha$).$\forall$a:$\alpha$.

1. Map$\mapsto$f(x)g(x)$|$x$\in\emptyset$ $\wedge$ p(x)=$\{|\,|\}$
2. p(a) $\Rightarrow$ $\{|\mapsto$f(x)g(x)$|$x$\in$(S+a)$\wedge$p(x)$|\}$= extend($\{|\mapsto$f(x)g(x)$|$x$\in$S$\wedge$p(x)$|\}$,f(a),g(a))
3. $\neg$p(a) $\Rightarrow$ $\{|\mapsto$f(x)g(x)$|$x$\in$(S+a)$\wedge$p(x)$|\}=\{|\mapsto$f(x)g(x)$|$x$\in$S$\wedge$p(x)$|\}$
4. $\forall$a$\in$S. p(a) $\Rightarrow$ $\{|\mapsto$f(x)g(x)$|$x$\in$S $\wedge$ p(x)$|\}$(f(a)) = g(a)

Lemma B.3.11 Map Composition
$\forall\alpha,\beta,\gamma$:TYPES.$\forall$M:Map($\alpha,\beta$).$\forall$M':Map($\beta,\gamma$).$\forall$a:$\alpha$.$\forall$b:$\beta$.$\forall$c:$\gamma$.

1. {| |}$\circ$M= {| |}
2. b$\in$domain(M') $\Rightarrow$ extend(M,a,b)$\circ$M' = extend(M$\circ$M',a,M'(b))
3. b$\notin$domain(M') $\Rightarrow$ extend(M,a,b)$\circ$M' = M$\circ$M'
4. M$\circ${| |}= {| |}
5. b$\notin$rangeαM $\Rightarrow$ M$\circ$extend(M',b,c) = M$\circ$M'

Lemma B.3.12 Map size
$\forall\alpha,\beta,\gamma$:TYPES.$\forall$M:Map($\alpha,\beta$).$\forall$M':Map($\beta,\gamma$).$\forall$a:$\alpha$.$\forall$b:$\beta$.

1. |{| |}| = 0
2. a$\in$domain(M) $\Rightarrow$ |extends(M,a,b)| = |M|
3. a$\notin$domain(M) $\Rightarrow$ |extends(M,a,b)| = |M|+1
4. M$\sqsubseteq$M' $\Rightarrow$ |M|$\leq$ |M'|
5. |M$\circ$M'|$\leq$|M|
6. |M$\circ$M'|$\leq$|M'|

B.4 Costas Arrays

Die folgende Erweiterung der Theorie der endlichen Folgen ist nötig, um das Costas-Arrays Problem lösen zu können. Wie immer beschreiben die Lemmata Distributivgesetze, die zur Vereinfachung verwendet werden.

Definition B.4.1 (dtrow: Reihe in der Differenzentafel)
$\texttt{dtrow}(L,j) \equiv \texttt{[}L\texttt{[i]-}L\texttt{[i+}j\texttt{] | i} \in \texttt{[1..|}L\texttt{|-}j\texttt{]]}$

Lemma B.4.2 dtrow
$\forall \texttt{L,L':Seq}(\mathbb{Z}).\forall \texttt{i:}\mathbb{Z}.\forall \texttt{j:}\mathbb{N}.$

1. $\texttt{dtrow([],j)} = \texttt{[]}$
2. $\texttt{j} \leq \texttt{|L|} \Rightarrow \texttt{dtrow(i.L,j)} = \texttt{(i-L[j]).dtrow(L,j)}$
3. $\texttt{j} \neq \texttt{0} \Rightarrow \texttt{dtrow([i],j)} = \texttt{[]}$
4. $\texttt{L} \sqsubseteq \texttt{L'} \Rightarrow \texttt{dtrow(L,j)} \sqsubseteq \texttt{dtrow(L',j)}$
5. $\texttt{j} \geq \texttt{|L|} \Rightarrow \texttt{dtrow(L,j)} = \texttt{[]}$
6. $\texttt{j} \leq \texttt{|L|} \Rightarrow \texttt{dtrow(L·i,j)} = \texttt{dtrow(L,j)·(L[|L|+1-j]-i)}$

B.5 Integer Segmente

Die folgende Erweiterung der Theorie der endlichen Folgen ist nötig, um das Problem der Maximalen Segmentsumme lösen zu können.

Definition B.5.1 (Segmentsummen und Maxima)

$\sum_{i=p}^{q} L\texttt{[i]}$	$\equiv \texttt{reduce(+,[}L\texttt{[i]|i} \in \texttt{[}p..q\texttt{]]})$
$\{f_{pq} \mid q \in S \wedge p \in S_q\}$	$\equiv \bigcup\{\ \{f_{pq} \mid p \in S_q\}\ \mid\ q \in S\}$
$m = \texttt{MAX}(S)$	$\equiv m \in S\ \wedge\ \forall \texttt{x} \in S.\ \texttt{x} \leq m$

Man beachte, daß die Segmentsumme $\sum_{i=p}^{q}\texttt{L[i]}$ nur für $L \neq []$ und $\texttt{1} \leq \texttt{p} \leq \texttt{q} \leq \texttt{|L|}$ definiert ist.

Lemma B.5.2 Segmentsumme
$\forall \texttt{L:Seq}(\mathbb{Z}).\forall \texttt{a:}\mathbb{Z}.$

1. $\sum_{i=1}^{1}\texttt{a.L[i]} = \texttt{a}$
2. $\forall \texttt{q} \in \texttt{domain(L).}\quad \sum_{i=1}^{q+1}\texttt{a.L[i]} = \sum_{i=1}^{q}\texttt{L[i]+a}$
3. $\forall \texttt{q} \in \texttt{domain(L).} \forall \texttt{p} \in \{\texttt{1..q}\}.\ \sum_{i=p+1}^{q+1}\texttt{a.L[i]} = \sum_{i=p}^{q}\texttt{L[i]}$
4. $\forall \texttt{q} \in \texttt{domain(L).} \forall \texttt{p} \in \{\texttt{1..q}\}.\ \sum_{i=p}^{q}\texttt{L·a[i]} = \sum_{i=p}^{q}\texttt{L[i]}$
5. $\forall \texttt{p} \in \texttt{domain(L).}\quad \sum_{i=p}^{|L|+1}\texttt{L·a[i]} = \sum_{i=p}^{|L|}\texttt{L[i]+a}$

Lemma B.5.3 Set Formers and Integer Ranges
$\forall \texttt{f:}\mathbb{Z}\to\mathbb{Z}.\forall \texttt{g:}\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}.\forall \texttt{j,k:}\mathbb{Z}.$

1. $\{\texttt{f(x)|x} \in \{\texttt{j+1..k+1}\}\} = \{\texttt{f(x+1)|x} \in \{\texttt{j..k}\}\}$
2. $\{\texttt{g(x,y)|x} \in \{\texttt{j+1..k+1}\} \wedge \texttt{y} \in \{\texttt{j+1..x}\}\} = \{\texttt{g(x+1,y)|x} \in \{\texttt{j..k}\} \wedge \texttt{y} \in \{\texttt{j+1..x+1}\}\}$
3. $\{\texttt{g(x+1,y)|x} \in \{\texttt{j..k}\} \wedge \texttt{y} \in \{\texttt{j+1..x+1}\}\} = \{\texttt{g(x+1,y+1)|x} \in \{\texttt{j..k}\} \wedge \texttt{y} \in \{\texttt{j..x}\}\}$
4. $\{\texttt{f(x)|x} \in \{\texttt{j..k}\}\} = \{\texttt{f(x)|x} \in \{\texttt{j+1..k}\}\} + \texttt{f(j)}$
5. $\{\texttt{g(x,y)|x} \in \{\texttt{j..k}\} \wedge \texttt{y} \in \{\texttt{j..x}\}\} = \{\texttt{g(x,y)|x} \in \{\texttt{j+1..k}\} \wedge \texttt{y} \in \{\texttt{j+1..x}\}\} \cup \{\texttt{g(x,j)|x} \in \{\texttt{j+1..k}\}\} + \texttt{g(j,j)}$
6. $\{\texttt{f(x)|x} \in \{\texttt{j..j}\}\} = \{\texttt{f(j)}\}$
7. $\{\texttt{g(x,y)|x} \in \{\texttt{j..j}\} \wedge \texttt{y} \in \{\texttt{j..x}\}\} = \{\texttt{g(j,j)}\}$

Lemma B.5.4 Maximum
$\forall \texttt{m,m',a:}\mathbb{Z}.\ \forall \texttt{S,S':Set}(\mathbb{Z}).$

1. $\neg(\texttt{m = MAX}(\emptyset))$
2. $\texttt{m=MAX(S)} \wedge \texttt{m'=MAX(S)} \Rightarrow \texttt{m=m'}$
3. $\texttt{m=MAX(S)} \Rightarrow \texttt{max(a,m)=MAX(S+a)}$
4. $\texttt{a=MAX(\{a\})}$
5. $\texttt{m=MAX(S)} \wedge \texttt{m'=MAX(S')} \wedge \texttt{S} \subseteq \texttt{S'} \Rightarrow \texttt{m} \leq \texttt{m'}$
6. $\texttt{m=MAX(S)} \Rightarrow \texttt{m+a=MAX(\{x+a|x} \in \texttt{S\})}$
7. $\texttt{m=MAX(S)} \wedge \texttt{m'=MAX(S')} \Rightarrow \texttt{max(m,m')=MAX(S} \cup \texttt{S')}$