- 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.