Automatisierte Logik und Programmierung

Teil III

Entwurf von Beweisassistenzsystemen

1. Entwurfsprinzipien
2. Bau von Inferenzmaschinen
3. Beweisautomatisierung
4. Verwaltung formalen Wissens
5. Interaktion mit Benutzern

Beweissysteme für die Typentheorie

- **Existierende Systeme sind Beweisassistenzsysteme**
  - Nuprl: Konstruktive Typentheorie (CTT) *
  - MetaPRL: Infrastruktursystem, Hauptanwendung CTT und CZF
  - Agda: Martin-Löf Typentheorie
  - Coq: Calculus of Constructions
  - PVS: Klassische Variante der Typentheorie
  - HOL: Klassische Typentheorie
  - Isabelle: Infrastruktursystem, Hauptanwendung HOL *
  - SpecWare: Algebraische Softwareentwicklung, eigene Typentheorie

  ⋮
- **Vollautomatische Systeme sind nicht praktikabel**
  - Hohe Ausdruckskraft bedingt vielfältige Unentscheidbarkeiten
  - Basismechanismus ist immer interaktive Beweiskonstruktion
  - Automatisierung der Beweisführung ist in Grenzen möglich
  - Starke Unterschiede in Benutzersteuerung und Extras

Allgemeine Entwurfsprinzipien

Es geht um mehr als nur mechanische Beweisführung

- **Computergestützte Regelanwendung**
  - Sichert korrekte Anwendung von Beweisregeln
  - Erspart Schreibarbeit bei Erzeugung von Teilzielen
- **Beweisautomatisierung**
  - Lösung trivialer, aber zeitaufwendiger Beweisteile
  - Strukturierung von Beweisen durch Makro-Beweisschritte
- **Mechanismen zur Verwaltung von Wissen**
  - Ermöglichen Wiederverwendung formal verifizierter Theoreme
  - Hilfe bei Einführung von Konzepten, Notationen und Methoden
  - Strukturierung von Wissen als formale mathematische Theorien
- **Gestaltung der Benutzerinteraktion**
  - Eingabe von Definitionen, Notation, Theoremen und Methoden
  - Visuelle oder kommandobasierte Steuerung von Beweisen
  - Viele Entwurfsentscheidungen erforderlich

Implementierung von Inferenzmaschinen

- **Konzepte der Theorie sind definiert in Meta-Sprache**
  - Term, Sequenz, Regel, Beweis, Theorem, ...
  - Definitionen liegen in relativ präziser Formulierung vor
- **Formalisiere Metasprache als Programmiersprache**
  - Formales Englisch mit eindeutig definierter Semantik $\mapsto$ ML
- **Implementiere Konzepte als abstrakte Datentypen**
  - Umsetzung der textuellen Definition in formale Metasprache ermöglicht nahezu direkte Implementierung der Theorie
  - Beinhaltet Operatoren zur Konstruktion / Analyse konkreter Objekte
  - Korrektheit der Implementierung läßt sich leicht überprüfen
- **Implementiere konkrete Terme & Regeln als Objekte**
  - Tabellen mit vordefinierten Elementen der abstrakten Typen
  - Beweisführung ist Anwendung der allgemeinen Funktion, die konkrete Beweisobjekte (aus Sequenzen & Regeln) konstruiert

(Details in §12)

Automatisierung von Beweisen

- **Interaktive Beweisführung und -prüfung** (Frühe Systeme, PCC)
  - Proof Checking: Computer überprüft vorgegebene formaler Beweise
  - Proof Editing: interaktive Beweiskonstruktion. Benutzer gibt Regeln an Computer führt Regeln aus und zeigt offene Teilprobleme
  - Unterschiede sind Art der Anwendung und Benutzerinteraktion
  - Ähnliche Implementierungsmethodik, leicht zu programmieren, klein
- **Taktisches Theorembeweisen**
  - Taktiken: programmierte Anwendung von Inferenzregeln
  - Entwurf anwendungsspezifischer Inferenzregeln durch Benutzer
  - Flexibel und sicher, gut für mittelgroße Anwendungen
- **Beweisprozeduren für spezielle Probleme**
  - Entscheidungsprozeduren, vollständige Beweissuche,
  - Rewriting, Beweisplaner, Model Checking, Computer Algebra, ...
  - Effiziente Techniken, aber eingeschränkte Anwendungsbereiche

(Details in §13–15)

Wissensgestützte Beweisführung

- **"Echte Beweisführung" ist niemals isoliert**
  - Beweise werden immer im Kontext einer Theorie geführt
  - Kontext bestimmt Begriffswelt, Notation, Erkenntnisse, Methoden,
- **Bibliothek muß formales Wissen verwalten**
  - Definitionen, Präsentationsformen, Theoreme, Beweistechniken,...
  - Mechanismen für Browsen, Suchen, Versionskontrolle, ..
  - Unterstützung einer hierarchischen Theoriestruktur mit Querbezügen
  - Abhängigkeitskontrolle und Konsistenzsicherung (zirkuläre Beweise?)
- **Verschiedene Gestaltungssmöglichkeiten**
  - Datei-textorientiert (Isabelle, Coq, MetaPRL)
    Konventionell editierbar, leicht austauschbar, muß compiliert werden
  - Abstrakte Datenbank (Nuprl)
    Multiuser-fähig, Undo/Redo/Backup, selektive Sichten, ...

(Details in §16)

Interaktion mit Benutzern

- **Benutzer muß Theorien interaktiv entwickeln können**
  - Erzeugung von Definitionen, Statements, Führen von Beweisen, ...
  - System bietet 'visuelle' Unterstützung für Bearbeitung von Objekten
  - System muß (Zwischen-)Ergebnisse anzeigen
- **Layoutfragen sind wichtig**
  - Präsentation muß verständliche mathematische Notation nutzen können
- **Verschiedene Gestaltungsmöglichkeiten**
  - Skript-/kommandorientiert (Isabelle, Coq, MetaPRL)

    Geringer Aufwand (alles findet mit Editor statt), leicht zu erlernen,

    Lineare Arbeitsweise - nur aktuelles Ziel sichtbar / bearbeitbar
  - Visuelle Interaktion (Nuprl)

    Benutzer navigiert durch Bibliothek, Beweisbaum, ...

    Parallele Bearbeitung mehrerer Ziele, mehr sichtbare Information

(Details in §16)

Kernbestandteile eines Beweisassistenzsystems

- **Inferenzmaschine** (Refiner)
  - Anwendung von Inferenzregeln (und Taktiken) auf Beweisziele
- **Bibliothek** (Library)
  - Verwaltung von des gesamten formalen Wissens
- **Benutzerinterface** (Editor)
  - Visuelles Interface zur Kommunikation mit Bibliothek
  - Unterstützung für Bearbeitung von Terme, Beweisen, Definitionen, ...
- **Optionale Komponenten**
  - Extraktion von Programmen aus Beweisen
  - Evaluator: Ausführung von Programmen
  - Exportmechanismen: Ascii Repräsentation, LaTeX, HTML, ...

Mechanismen sind unabhängig implementierbar

Automatisierte Logik und Programmierung

Einheit 12

Implementierung von Inferenzmaschinen

1. Formalisierung der Metasprache
2. Implementierung allgemeiner Konzepte
3. Einbettung der konkreten Theorie
4. Verarbeitung von Inferenzregeln

ML: Formalisierung der Metasprache der CTT

- **Entstanden im Edinburgh LCF Projekt** (frühe 70er Jahre)
  - Formales Englisch zur Unterstützung von logischer Symbolverarbeitung
  - Standardisiert Ende der 80er Jahre als SML und Caml
  - Nuprl benutzt die Originalversion "Classic ML" (Appendix B des Manuals)
- **Funktionale Programmiersprache höherer Stufe**
  - Programmieren = Definition + Anwendung von Funktionen (wie λ-Kalkül)
  - Pattern Matching unterstützt Verständlichkeit komplexe Definitionen
- **Erweiterbare polymorphe Typdisziplin**
  - Grundkonstrukte: `int`, `bool`, `tok`, `string`, `unit`, A`->`B, A`#`B, A`+`B, A `list`
  - Anwenderdefinierbare abstrakte und rekursive Datentypen
  - Typprüfung mit erweitertem Hindley/Milner Typechecking Algorithmus
- **Kontrollierte Behandlung von Ausnahmen**
  - Anwenderdefinierbare Verarbeitung von Laufzeitfehlern

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion:

```
let divides = \x.\y.((x/y)*y = x);;
```

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion: `let divides = \x.\y.((x/y)*y = x);;`
  - Definitorische Gleichung: `let divides x y = ((x/y)*y = x);;`

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion: `let divides = \x.\y.((x/y)*y = x);;`
  - Definitorische Gleichung: `let divides x y = ((x/y)*y = x);;`
  - Anwendung: `divides 4 5;;`

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion: `let divides = \x.\y.((x/y)*y = x);;`
  - Definitorische Gleichung: `let divides x y = ((x/y)*y = x);;`
  - Anwendung: `divides 4 5;;`
- **Rekursive Funktionsdefinition**

```
letrec MIN f init = if f(init)=0 then init
                                 else MIN f (init+1)
```

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion: `let divides = \x.\y.((x/y)*y = x);;`
  - Definitorische Gleichung: `let divides x y = ((x/y)*y = x);;`
  - Anwendung: `divides 4 5;;`
- **Rekursive Funktionsdefinition**

```
letrec MIN f init = if f(init)=0 then init
                                 else MIN f (init+1)
```

- **Lokale Abstraktion**

```
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 [] ;;
```

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion:
    ```
    let divides = \x.\y.((x/y)*y = x);;
    ```
  - Definitorische Gleichung:
    ```
    let divides x y = ((x/y)*y = x);;
    ```
  - Anwendung:
    ```
    divides 4 5;;
    ```
- **Rekursive Funktionsdefinition**
  ```
  letrec MIN f init = if f(init)=0 then init
                                   else MIN f (init+1)
  ```
- **Lokale Abstraktion**
  ```
  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 [] ;;
  ```
- **Pattern Matching**
  ```
  let x.y.rest = upto 1 5 in x,y
  ```

Grundkonzepte von ML

- **Definition einfacher Funktionen**
  - Abstraktion: `let divides = \x.\y.((x/y)*y = x);;`
  - Definitorische Gleichung: `let divides x y = ((x/y)*y = x);;`
  - Anwendung: `divides 4 5;;`
- **Rekursive Funktionsdefinition**

```
letrec MIN f init = if f(init)=0 then init
                                 else MIN f (init+1)
```

- **Lokale Abstraktion**

```
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 [] ;;
```

- **Pattern Matching**

```
let x.y.rest = upto 1 5 in x,y
```

- **Ausnahmen**

```
let divides x y = ((x/y)*y = x) ? false;;
```

(für `divides x 0`)

Formulierung abstrakter Datentypen in ML

```
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)
;;
```

```
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)
;;
```

---

abs_T, rep_T: Konversionen: explizite $\longleftrightarrow$ abstrakte Repräsentation

Implementierung der Typentheorie in ML

- **Repräsentiere Grundkonzepte als ML-Datentypen**
  - Präzisiere Definitionen der Theorie aus §8/9 in formaler Metasprache
    Syntax (Terme), Semantik (Evaluation,...), Inferenzsystem (Sequenzen, Beweise,...)
  - Korrektheit der Implementierung leicht zu überprüfen
- **Kapselung konkreter Objekte durch abstrakte Datentypen**
  - Erzeugung von Objekten und Komponentenzugriff nur mit Konstruktoren und Destruktoren, die durch die Typdeklaration definiert sind
  - z.B. Änderung von Beweisen nur durch Anwendung von "Taktiken"
    Taktiken können im Endeffekt nur aus Regeln erzeugt werden
  - Verhindert unbefugte Manipulation von Theorien & Beweisen
- **Implementiere konkrete Theorie als Objekte dieser Typen**
  - Tabellen für vordefinierte Terme und zugehörige Inferenzregeln
  - Unterstützt durch Algorithmen für Substituion und Evaluation
- **Unterstützung für konservative Erweiterungen**
  - Einfache explizite Mechanismen zur Erweiterung der Syntaxtabellen
  - Beweisautomatisierung als ML-Programme, welche Regelaufrufe steuern

Repräsentation der Syntax der Typentheorie

- **Maschinenlesbarkeit ist notwendig**
  - +: Leichte Erzeugung des abstrakten Syntaxbaums aus formalem Text
    - · Syntaxbaum essentiell für Verarbeitung von Ausdrücken
  - −: Starre, maschinenlesbare Syntax ist unnatürlich
- **Flexibilität ist wichtig für Lesbarkeit**
  - +: Verständlichkeit steigt durch Verwendung von Lehrbuchnotationen

    Es gibt viele gleich gute Notationen für dasselbe Konzept
  - −: Zu viel Flexibilität macht Syntax sehr schwer zu verarbeiten
- **Versuche beides zu vereinigen**
  - – Vielfalt der Konzepte verlangt standardisierte Mechanismen für
    - · Syntaktische Analyse vieler verschiedenartiger Ausdrücke
    - · Substitution und Evaluation von Termen
  - – Mechanismen sollten Erweiterungen (auch konservative) unterstützen

Syntaxdefinition ohne Systematisierung

- x Term, falls x Variable
- (t) Term, falls t Term
- $x:S\to T$ und $S\to T$ Terme, falls x Variable, S und T Terme
- $\lambda x\,.\,e$ Term, falls x Variable, e Term
- $e_1\ e_2$ Term, falls e_1 und e_2 Terme
- $x:S\times T$ und $S\times T$ Terme, falls x Variable, S und T Terme
- (e_1, e_2) Term, falls e_1 und e_2 Terme
- $\mathsf{match}\ e\ \mathsf{with}\ \langle x\,,y\rangle \mapsto u$ Term, falls x, y Variablen, e und u Terme
- $S+T$ Term, falls S und T Terme
- $\mathsf{inl}(e)$ und $\mathsf{inr}(e)$ Terme, falls e Term
- $\mathsf{case}\ \ e\ \ \mathsf{of}\ \ \mathsf{inl}(x_1)\mapsto e_1\ \ |\ \ \mathsf{inr}(x_2)\mapsto e_2$ Term, falls x_1, x_2 Variablen, e, e_1 und e_2 Terme
- $s{=}t\ \mathtt{in}\ T$ Term, falls s, t und T Terme
- Ax Term
- $\mathbb{U}_j$ Term, falls j natürliche Zahl oder Variable

Syntaxanalyse wird schnell unüberschaubar

Ein einheitliches Prinzip der Syntaxdarstellung

Trenne "Abstraktionsform" und Darstellung von Termen

- **Strikte maschinenlesbare interne Struktur**
  - Abstrakte Einheitssyntax für Terme – leicht zu verarbeiten
  - Einheitliche Mechanismen für Variablenvorkommen und Substitution
  - Individuelle Konzepte sind Terme dieser Syntax $\mapsto$ Tabelleneinträge
- **Flexible externe Präsentation**
  - Notation (Präsentation) wird getrennt von Konzept (formaler Term)
  - Display Formen unterstützen vielfältige Notationen für denselben Term
    - Ermöglicht (nahezu) natürliche externe Syntax
    - Erlaubt Notationswechsel ohne Veränderung des eigentlichen Terms
- **Prinzip unterstützt Erweiterungen der Theorie**
  - Ergänze Tabelle der Terme und definiere Display Formen
  - Gleiche Methodik für echte und konservative Erweiterungen

Uniforme interne Syntax – was ist nötig?

- **Name**
  - Terme müssen eindeutig identifizierbar sein
  - Ax ist Präsentationsform eines Terms mit Namen **Axiom**
- **Parameter in Namen**
  - Namen können parametrisiert werden
  - x ist eine Variable mit Namen x – Name des Terms ist $\mathbf{var}\{x : \mathtt{v}\}$
- **Teilterme**
  - Die meisten Terme sind aus anderen Termen zusammengesetzt
  - $f\ t$ ist Abkürzung für einen Ausdruck der Form $\mathbf{apply}(f\,;t)$
- **Gebundene Variablen in Teiltermen**
  - Teilterme können Variablen enthalten, die gebunden werden
  - In $\lambda x\,.\,t$ wird x in t gebunden – Ausdruck ist $\mathbf{lam}(x\,.\,t)$
  - In $x : S \to T$ wird x in T (nicht in S) gebunden
    Ausdruck ist $\mathbf{function}(S\,;\,x\,.\,T)$

Uniforme interne Syntax – was ist nötig?

- **Name**
  - Terme müssen eindeutig identifizierbar sein
  - Ax ist Präsentationsform eines Terms mit Namen **Axiom**
- **Parameter in Namen**
  - Namen können parametrisiert werden
  - x ist eine Variable mit Namen x – Name des Terms ist $\mathbf{var}\{x:\mathrm{v}\}$
- **Teilterme**
  - Die meisten Terme sind aus anderen Termen zusammengesetzt
  - $f\ t$ ist Abkürzung für einen Ausdruck der Form $\mathbf{apply}(f;t)$
- **Gebundene Variablen in Teiltermen**
  - Teilterme können Variablen enthalten, die gebunden werden
  - In $\lambda x\,.\,t$ wird x in t gebunden – Ausdruck ist $\mathbf{lam}(x\,.\,t)$
  - In $x:S\to T$ wird x in T (nicht in S) gebunden
    Ausdruck ist $\mathbf{function}(S;\,x\,.\,T)$

Allgemeine Form: $\boldsymbol{opid}\{\boldsymbol{p}_1..\boldsymbol{p}_k\}(\boldsymbol{x}_1^1..\boldsymbol{x}_{m_1}^1\,.\,\boldsymbol{t}_1;\ldots\boldsymbol{x}_1^n..\boldsymbol{x}_{m_n}^n\,.\,\boldsymbol{t}_n)$

Präzise Definition typetheoretischer Terme

Die Abstraktionsform eines Terms hat die Gestalt

$$opid\{p_1{:}F_1, ..p_k{:}F_k\}(x_1^1, ..x_{m_1}^1.t_1; \ldots x_1^n, ..x_{m_n}^n.t_n)$$

wobei

- $opid\{p_1{:}F_1, ..p_k{:}F_k\}$ **Operator**:
  - *opid*: **Operatorname** → *Operatorentabelle*

    Operatorentabelle muß Operator var enthalten
  - $p_j{:}F_j$: **Parameter**
    - F_j: **Parametertyp** `(var, nat, tok, str, level)`
    - p_j: **Parameterwert**
- $x_1^i, \ldots, x_{m_i}^i.t_i$ **gebundener Term**, mit Term t_i und Variablen x_k^i

  $(m_1,..,m_n)$ ist die **Arität** (Anzahlen der Variablen aller Teilterme)

Repräsentation von Termen als ML-Datentyp

Direkte Umsetzung der Definition von Termen

$$opid\{p_1 : F_1, .. p_k : F_k\}(x_1^1, .. x_{m_1}^1 . t_1; \ldots x_1^n, .. x_{m_n}^n . t_n)$$

opid	Operatorname
$p_j : F_j$	Parameter, bestehend aus Parameterwert und Parametertyp
$x_1^i, \ldots, x_{m_i}^i . t_i$	gebundener Term, wobei t_i Term, x_k^j Variable

---

```
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
;;
abstype  var  = tok
  with tok_to_var t = abs_var t
  and  var_to_tok  v = rep_var v
;;
abstype  level_exp  = tok + int   with   ...
abstype  parm =  ...
```

Auszug der Operatorentabelle

Operator und Termstruktur	*Standard-Darstellungsform*
var$\{x{:}\mathrm{v}\}()$	x
function$\{\}(S\,;\,x\,.\,T)$	$x : S \to T$
lambda$\{\}(x\,.\,t)$	$\lambda x\,.\,t$
apply$\{\}(f\,;t)$	$f\ t$
product$\{\}(S\,;\,x\,.\,T)$	$x : S \times T$
pair$\{\}(s\,,t)$	$\langle s\,,t\rangle$
spread$\{\}(e\,;\,x\,,y\,.\,u)$	match e with $\langle x\,,y\rangle \mapsto u$
union$\{\}(S\,;T)$	$S+T$
inl$\{\}(e)$	$\mathsf{inl}(e)$
inr$\{\}(e)$	$\mathsf{inr}(e)$
decide$\{\}(e\,;\,x_1\,.\,e_1\,;\,x_2\,.\,e_2)$	case e of $\mathsf{inl}(x_1) \mapsto e_1$ | $\mathsf{inr}(x_2) \mapsto e_2$
Axiom$\{\}()$	Ax
equal$\{\}(s\,;t\,;T)$	$s{=}t$ `in` T
universe$\{j{:}\mathtt{l}\}()$	$\mathbb{U}_j$

Erzeugung einer lesbaren Termdarstellung

- **Ordne abstrakten Termen eine Display Form zu**
  - Display Form beschreibt mögliche textliche Darstellung des Terms

    `EdAlias lam: λ<x:var>.<t:term:E> == lambda{}(<x>.<t>)`
  - Linke Seite beschreibt, wie rechte Seite darzustellen ist
  - Beliebige Anordnung von Parametern (spitze Klammern) in Freitext
- **Sonderformen erhöhen Flexibilität**
  - Iteration ermöglicht verkürzte Schreibweisen: $\lambda x, y.t$ statt $\lambda x.\lambda y.t$

```
EdAlias lam: λ<x:var>.<t:term:E> == lambda{}(<x>.<t>)
     #Hd A:: λ<x:var>,<#:term:E> == lambda{}(<x>.<#>)
     #Tl A:: <x:var>.<t:term:E>  == lambda{}(<x>.<t>)
     #Tl A:: <x:var>,<#:term:E>  == lambda{}(<x>.<#>)
```

  - Weglassen von Parametern links möglich (implizite Information)
  - Unterstützung für Layout großer Terme, automatische Klammerung ...

```
EdAlias primrec: h(<t:int>) {->}where h(0) = <f:base>
                                {\\}and h(n+1) = <g:step>(<n:var>,h(n)/<x:var>)
                 == p_rec{}(<t>;<f>;<n>,<x>.<g>)
```

Abstraktion und Präsentation eines Terms werden simultan betrachtet

Einbettung der konkreten Objektsprache

- **Basisterme der CTT**

Operator und Termstruktur	*Darstellungsform*
$\mathbf{function}\{\}(S\,;\,x\,.\,T)$	$x:S\rightarrow T$
$\mathbf{lambda}\{\}(x\,.\,t)$	$\lambda x\,.\,t$
$\vdots$	$\vdots$

  – Konkrete Operatoren werden in Operatorentabelle verwaltet

Einbettung der konkreten Objektsprache

- **Basisterme der CTT**

Operator und Termstruktur	*Darstellungsform*
function$\{\}(S\,;\,x\,.\,T)$	$x:S\to T$
lambda$\{\}(x\,.\,t)$	$\lambda x\,.\,t$
$\vdots$	$\vdots$

  – Konkrete Operatoren werden in Operatorentabelle verwaltet

- **Operatorentabelle wird durch Bibliotheksobjekte erzeugt**
  – Bibliothek enthält Deklaration (Abstraktion) und Präsentationsform

```
- ABS: function def
function{}(.A;x,.B[x];) == !primitive
```

```
- DISP: function
<x:var>:<A:term> → <B:term> == function{}(.<A>;<x>,.<B>;)
```

  – Objekte werden beim Start und bei Änderungen in Tabelle übertragen
  – Vorteil: Operatoren können lokal in Theorien deklariert werden

Einbettung der konkreten Objektsprache

- **Basisterme der CTT**

Operator und Termstruktur	*Darstellungsform*
$\mathsf{function}\{\}(S\,;x\,.\,T)$	$x:S\to T$
$\mathsf{lambda}\{\}(x\,.\,t)$	$\lambda x\,.\,t$
$\vdots$	$\vdots$

  – Konkrete Operatoren werden in Operatorentabelle verwaltet

- **Operatorentabelle wird durch Bibliotheksobjekte erzeugt**
  – Bibliothek enthält Deklaration (Abstraktion) und Präsentationsform

```
- ABS: function def
function{}(.A;x,.B[x];) == !primitive
```

```
- DISP: function
<x:var>:<A:term> → <B:term> == function{}(.<A>;<x>,.<B>;)
```

  – Objekte werden beim Start und bei Änderungen in Tabelle übertragen
  – Vorteil: Operatoren können lokal in Theorien deklariert werden

- **Konstruktoren & Destruktoren für konkrete Terme**
  – Spezialisierte Form der Konstruktoren & Destruktoren des Datentyps

```
let mk_function_term x S T  =  mk_term ('function',[]) [[],S; [x],T]
let dest_function t  =  let op,[(),a; [x],b] = dest_term t in x,a,b
```

  – In der Bibliothek gespeichert als Code-Objekte der Core-Theorie

Erweiterung der Termsprache

- **Abstraktionen für konservative Erweiterungen**
  - $\texttt{add\{\}(.i;.j)} \equiv \texttt{PR[i, n,add-in} \mapsto \texttt{s(add-in)](j)}$
  - $\texttt{mul\{\}(.i;.j)} \equiv \texttt{PR[0, n,mul-in} \mapsto \texttt{add\{\}(.mul-in;j)](j)}$
  - Linke Seite beschreibt Erweiterung der Operatorentabelle
  - Rechte Seite beschreibt Semantik durch existierende Terme

Erweiterung der Termsprache

- **Abstraktionen für konservative Erweiterungen**
  - `add{}(.i;.j)` $\equiv$ `PR[i, n,add-in`$\mapsto$`s(add-in)](j)`
  - `mul{}(.i;.j)` $\equiv$ `PR[0, n,mul-in`$\mapsto$`add{}(.mul-in;j)](j)`
  - – Linke Seite beschreibt Erweiterung der Operatorentabelle
  - – Rechte Seite beschreibt Semantik durch existierende Terme
- **Unechte Abstraktionen für echte Theorieerweiterungen**
  - – Rechte Seite der Abstraktion bleibt undefiniert (`!primitive`)
  - – Semantik und Inferenzregeln sind explizit zu beschreiben

Erweiterung der Termsprache

- **Abstraktionen für konservative Erweiterungen**
  - `add{}(.i;.j)` $\equiv$ `PR[i, n,add-in`$\mapsto$`s(add-in)](j)`
  - `mul{}(.i;.j)` $\equiv$ `PR[0, n,mul-in`$\mapsto$`add{}(.mul-in;j)](j)`
  - – Linke Seite beschreibt Erweiterung der Operatorentabelle
  - – Rechte Seite beschreibt Semantik durch existierende Terme
- **Unechte Abstraktionen für echte Theorieerweiterungen**
  - – Rechte Seite der Abstraktion bleibt undefiniert (`!primitive`)
  - – Semantik und Inferenzregeln sind explizit zu beschreiben
- **Displayformen sind in beiden Fällen anzugeben**
  - `EdAlias add: <i:int>+<j:int> == add{}(.<i>;.<j>)`
  - `EdAlias mul: <i:int>*<j:int> == mul{}(.<i>;.<j>)`

Erweiterung der Termsprache

- **Abstraktionen für konservative Erweiterungen**
  - `add{}(.i;.j)` $\equiv$ `PR[i, n,add-in`$\mapsto$`s(add-in)](j)`
  - `mul{}(.i;.j)` $\equiv$ `PR[0, n,mul-in`$\mapsto$`add{}(.mul-in;j)](j)`
  - – Linke Seite beschreibt Erweiterung der Operatorentabelle
  - – Rechte Seite beschreibt Semantik durch existierende Terme
- **Unechte Abstraktionen für echte Theorieerweiterungen**
  - – Rechte Seite der Abstraktion bleibt undefiniert (`!primitive`)
  - – Semantik und Inferenzregeln sind explizit zu beschreiben
- **Displayformen sind in beiden Fällen anzugeben**
  - `EdAlias add: <i:int>+<j:int> == add{}(.<i>;.<j>)`
  - `EdAlias mul: <i:int>*<j:int> == mul{}(.<i>;.<j>)`
- **Beispiel für Präsentation von Termen**
  - – Interne Form
```
lambda{}(x.lambda{}(y.lambda{}(z.
                add{}(.var{x:v}();
                      .mul{}(.var{y:v}();.var{z:v}())))))
```
  - – Externe Präsentation `λx,y,z. x+y*z`

Repräsentation von Semantik

- **Semantik definiert durch Urteile für kanonische Terme**
  - Kanonische und nichtkanonische Terme müssen getrennt werden
  - Nichtkanonische Terme werden zu kanonischen Termen reduziert
  - Konkrete Urteile werden durch Inferenzregeln repräsentiert ↦ Folie 25
- **Semantik benötigt Evaluationsalgorithmus**
  - Lazy Evaluation basierend auf Reduktionstabelle für konkreter Terme
  - Evaluation verwendet Algorithmus für Substitution von Termen
  - Substitution muß Vorkommen von Variablen in Termen identifizieren
- **Implementierung nutzt standardisierte Mechanismen**
  - Mechanismen basieren auf einheitlicher Termdarstellung (und Tabellen)
  - Unterstützt Vielfalt und Erweiterbarkeit der formalen Sprache

Einheitliche Definitionen sind erforderlich

Vorkommen von Variablen würde sonst viele Definitionen benötigen

- x: die Variable x kommt frei vor; $y \neq x$ kommt nicht vor.
- (t): freie Vorkommen von x in t bleiben frei, gebundene Vorkommen bleiben gebunden.
- $x : S \to T$: freie Vorkommen von x in T werden gebunden;
  freie Vorkommen von $y \neq x$ in T bleiben frei, gebundene Vorkommen bleiben gebunden.
- $\lambda x.t$: freie Vorkommen von x in t werden gebunden;
  freie Vorkommen von $y \neq x$ in t bleiben frei, gebundene Vorkommen bleiben gebunden.
- $f\ t$: freie Vorkommen von x in f und t bleiben frei, gebundene bleiben gebunden.
- $x : S \times T$: freie Vorkommen von x in T werden gebunden;
  freie Vorkommen von $y \neq x$ in T bleiben frei, gebundene bleiben gebunden.
- $\langle a\,, b\rangle$: freie Vorkommen von x in a und b bleiben frei, gebundene bleiben gebunden.
- $\mathsf{match}\ p\ \mathsf{with}\ \langle x\,, y\rangle \mapsto t$: freie Vorkommen von x/y in t werden gebunden; freie Vorkommen von $z \neq x/y$ in t und jeder Variablen z' in p bleiben frei, gebundene bleiben gebunden.
- $S + T$: freie Vorkommen von x in S und T bleiben frei, gebundene bleiben gebunden.
- $\mathsf{inl}(e)/\mathsf{inr}(e)$: freie Vorkommen von x in e bleiben frei, gebundene bleiben gebunden.
- $\mathsf{case}\ \ e\ \ \mathsf{of}\ \ \mathsf{inl}(x_1) \mapsto e_1\ \ |\ \ \mathsf{inr}(x_2) \mapsto e_2$ freie Vorkommen von x_1 in e_1 bzw. x_2 in e_2 werden gebunden; freie Vorkommen von $y \neq x_1$ in e_1 bzw. $y \neq x_2$ in e_2 bleiben frei; freie Vorkommen von y in e bleiben frei; gebundene bleiben gebunden.
- $s = t \in T$: freie Vorkommen von x bleiben frei, gebundene bleiben gebunden.
- $\mathbb{U}_i$: x kommt nicht vor.

Variablenvorkommen auf Basis der Einheitssyntax

Einfache, universell verwendbare Definition ohne Spezialfälle

- $\mathsf{var}\{x:\mathrm{v}\}$: die Variable $x \in \mathcal{V}$ kommt frei vor;
  $y \neq x$ kommt nicht vor.
- $op(b_1; ..; b_n)$: freie Vorkommen von x in b_i bleiben frei,
  gebundene Vorkommen bleiben gebunden.
- $x_1, .., x_m \,.\, t$: freie und gebundene Vorkommen der x_i in t werden gebunden;
  freie Vorkommen von $y \neq x_i$ bleiben frei,
  gebundene Vorkommen bleiben gebunden

$t[x_1, .., x_n]$: $x_1, .., x_n$ kommen frei in t vor

t geschlossen: t enthält keine freien Variablen

Substitution auf Basis der Einheitssyntax

- **Mathematisch: endliche Abbildung von Variablen in Terme**
  - $\boldsymbol{\sigma} = [\boldsymbol{t_1}, .., \boldsymbol{t_n}/\boldsymbol{x_1}, .., \boldsymbol{x_n}] \hat{=} \sigma(x_1){=}t_1, \ldots, \sigma(x_n){=}t_n$
  - $\boldsymbol{e\sigma}$: Anwendung von σ auf den Ausdruck e

    $e[t/x]$ ersetzt jedes freie Vorkommen von x in e durch t
- **Algorithmisch: induktiv definiertes Verfahren**
  - Regeln beschreiben Zerlegung der Termstruktur bei Anwendung von σ

$\mathsf{var}\{x:\mathrm{v}\}()[t/x]$	$= t$	
$\mathsf{var}\{x:\mathrm{v}\}()[t/y]$	$= \mathsf{var}\{x:\mathrm{v}\}()$	$(y{\neq}x)$
$(op(b_1;\ldots;b_n))[t/x]$	$= op(b_1[t/x];\ldots;b_n[t/x])$	
$(x_1,\ldots,x_n.u)[t/x_i]$	$= x_1,\ldots,x_n.u$	
$(x_1,\ldots,x_n.u)[t/y]$	$= x_1,\ldots,x_n.u[t/y]$	*
$(x_1,..x_j,..x_n.u)[t/y]$	$= (x_1,..z,..x_n.u[z/x_j])[t/y]$	**

*: *y verschieden von allen x_i, y nicht frei in u oder keines der x_i frei in t*

**: *y verschieden von allen x_i, y frei in u, x_j frei in t, z neue Variable*

Auswertung von Ausdrücken

Die Semantik von Typentheorie basiert auf Lazy Evaluation

- **Typisierbare Terme sind nicht normalisierbar**
  - Extensionale Gleichheit erlaubt Typsierung nichtterminierender Terme
    Extensionale Gleichheit ist für viele Anwendungen sehr wichtig
  - Starke und schwache Normalisierbarkeit sind nicht wirklich erforderlich
  - Es reicht, Ausdrücke nur "bei Bedarf" auszuwerten
    Ein Funktionskörper wird bei Anwendung der Funktion ausgerechnet
    Komponenten eines Paars werden erst bei einem Zugriff bestimmt
- **Der Begriff des Wertes wird abgeschwächt**
  - Die äußere Form entscheidet, ob ein Term als Wert angesehen wird
  - Innerhalb des Terms können durchaus noch Redizes vorkommen
- **Auswertung folgt einer "lässigen" Strategie**
  - Die Auswertung endet, wenn die äußere Form einem Wert entspricht
  - Reduzierbare innere Teilterme werden erst bei Bedarf ausgewertet

Voraussetzungen zur Auswertung von Ausdrücken

- **Deklariere kanonische Terme in Operatorentabelle**
  - Kennzeichne *opid* des abstrakten Terms als kanonisch (var ist kanonisch)
- **Deklariere Hauptargumente nichtkanonischer Terme**
  - Markiere Teilterme der Terme t als "principal arguments"
  - Dort werden kanonische Elemente des entsprechenden Typs erwartet
- **Definiere Reduzierbarkeit spezieller Terme**
  - **Redex**: nichtkanonischer Term t, dessen Hauptargumente kanonische Elemente des entsprechenden Typs sind
  - **Kontraktum von t**: zugehörige reduzierte Form
  - **t reduzierbar auf u** ($t \xrightarrow{\beta} u$), falls u Kontraktum von t

  Konkrete Reduzierbarkeiten sind in *Redex–Kontrakta Tabelle* eingetragen
- **Definiere geschlossene kanonische Terme als Werte**
  - **u ist Wert eines geschlossenen Terms t** ($t \xrightarrow{l} u$), falls u Ergebnis der lässigen Auswertung auf t ist

  (Algorithmus `EVAL`, Folie 23)

Operatorentabelle mit Kennzeichnung (Auszug)

Operator und Termstruktur	*Standard-Darstellungsform*	*kanonisch?*
var$\{x{:}\mathrm{v}\}()$	x	ja
function$\{\}(S\,;\,x\,.\,T)$	$x:S\to T$	ja
lambda$\{\}(x\,.\,t)$	$\lambda x\,.\,t$	ja
apply$\{\}(\boxed{f}\,;t)$	$\boxed{f}\ t$	nein
product$\{\}(S\,;\,x\,.\,T)$	$x:S\times T$	ja
pair$\{\}(s\,,t)$	$\langle s\,,t\rangle$	ja
spread$\{\}(\boxed{e}\,;\,x\,,y\,.\,u)$	match $\boxed{e}$ with $\langle x\,,y\rangle\mapsto u$	nein
union$\{\}(S\,;T)$	$S+T$	ja
inl$\{\}(e)$	$\mathsf{inl}(e)$	ja
inr$\{\}(e)$	$\mathsf{inr}(e)$	ja
decide$\{\}(e\,;\,x\,.\,s\,;\,y\,.\,t)$	case e of $\mathsf{inl}(x)\mapsto s$ | $\mathsf{inr}(y)\mapsto t$	nein
equal$\{\}(s\,;t\,;T)$	$s{=}t$ `in` T	ja
Axiom$\{\}()$	Ax	ja
universe$\{j:\mathtt{l}\}()$	$\mathbb{U}_j$	ja

Hauptargumente nichtkanonischer Terme sind umrandet

Redex–Kontrakta Tabelle (Auszug)

Redex		Kontraktum
$\mathsf{apply}\{\}(\mathsf{lambda}\{\}(x\,.\,u)\,;t)$	$\xrightarrow{\beta}$	$u[t/x]$
$\mathsf{spread}\{\}(\mathsf{pair}\{\}(s\,,t)\,;\,x\,,y\,.\,u)$	$\xrightarrow{\beta}$	$u[s,t/x,y]$
$\mathsf{decide}\{\}(\mathsf{inl}\{\}(t)\,;\,x\,.\,u\,;\,y\,.\,v)$	$\xrightarrow{\beta}$	$u[t/x]$
$\mathsf{decide}\{\}(\mathsf{inr}\{\}(t)\,;\,x\,.\,u\,;\,y\,.\,v)$	$\xrightarrow{\beta}$	$v[t/y]$
$\vdots$	$\vdots$	$\vdots$
$(\lambda x\,.\,u)\ t$	$\xrightarrow{\beta}$	$u[t/x]$
$\mathsf{match}\ \langle s\,,t\rangle\ \mathsf{with}\ \langle x\,,y\rangle \mapsto u$	$\xrightarrow{\beta}$	$u[s,t/x,y]$
$\mathsf{case}\ \ \mathsf{inl}\{\}(t)\ \ \mathsf{of}\ \ \mathsf{inl}(x) \mapsto u\ \ \mid\ \ \mathsf{inr}(y) \mapsto v$	$\xrightarrow{\beta}$	$u[t/x]$
$\mathsf{case}\ \ \mathsf{inr}\{\}(t)\ \ \mathsf{of}\ \ \mathsf{inl}(x) \mapsto u\ \ \mid\ \ \mathsf{inr}(y) \mapsto v$	$\xrightarrow{\beta}$	$v[t/y]$
$\vdots$	$\vdots$	$\vdots$

Algorithmus EVAL(t) für lässige Auswertung

- **Falls t kanonisch** → **Ausgabe t**
- **Falls t nichtkanonisch:**
  - **Bestimme Hauptargumente $t_1, ..t_n$ von t**
  - **Berechne s_1:=EVAL(t_1), ..., s_n:=EVAL(t_n)**
  - **Bestimme $s := t[s_1, ..s_n/t_1, ..t_n]$** *
  - **Falls s ein Redex ist:**
    - **Bestimme das zugehörige Kontraktum u**
    - **Berechne v := EVAL(u)** → **Ausgabe v**
  - **Falls s kein Redex ist:**
    - **Stoppe ohne Ergebnis: t besitzt keinen Wert**

*: *$t[s_1, ..t_n/s_1, ..t_n]$ ist hier Ersetzung von Termen (Syntaxbäumen) durch Terme*

Algorithmus EVAL(t) für lässige Auswertung

- **Falls t kanonisch** → **Ausgabe t**
- **Falls t nichtkanonisch:**
  - **Bestimme Hauptargumente $t_1, ..t_n$ von t**
  - **Berechne s_1:=EVAL(t_1), ..., s_n:=EVAL(t_n)**
  - **Bestimme $s := t[s_1, ..s_n/t_1, ..t_n]$** *
  - **Falls s ein Redex ist:**
    - **Bestimme das zugehörige Kontraktum u**
    - **Berechne v := EVAL(u)** → **Ausgabe v**
  - **Falls s kein Redex ist:**
    - **Stoppe ohne Ergebnis: t besitzt keinen Wert**

Auswertungsbeispiel

– EVAL((λx.⟨(λy.s(y)) x, 0⟩) 0)

*: $t[s_1, ..t_n/s_1, ..t_n]$ *ist hier Ersetzung von Termen (Syntaxbäumen) durch Terme*

Algorithmus EVAL(t) für lässige Auswertung

- **Falls t kanonisch** → **Ausgabe t**
- **Falls t nichtkanonisch:**
  - **Bestimme Hauptargumente $t_1, ..t_n$ von t**
  - **Berechne s_1:=EVAL(t_1), ..., s_n:=EVAL(t_n)**
  - **Bestimme $s := t[s_1, ..s_n/t_1, ..t_n]$** *
  - **Falls s ein Redex ist:**
    - **Bestimme das zugehörige Kontraktum u**
    - **Berechne v := EVAL(u)** → **Ausgabe v**
  - **Falls s kein Redex ist:**
    - **Stoppe ohne Ergebnis: t besitzt keinen Wert**

Auswertungsbeispiel

– EVAL((λx.⟨(λy.s(y)) x, 0⟩) 0)

= EVAL(EVAL(λx.⟨(λy.s(y)) x,0⟩) 0) Hauptargument kanonisch

*: *$t[s_1, ..t_n/s_1, ..t_n]$ ist hier Ersetzung von Termen (Syntaxbäumen) durch Terme*

Algorithmus EVAL(t) für lässige Auswertung

- **Falls t kanonisch** → **Ausgabe t**
- **Falls t nichtkanonisch:**
  - **Bestimme Hauptargumente $t_1, ..t_n$ von t**
  - **Berechne s_1:=EVAL(t_1), ..., s_n:=EVAL(t_n)**
  - **Bestimme $s := t[s_1, ..s_n/t_1, ..t_n]$** *
  - **Falls s ein Redex ist:**
    - **Bestimme das zugehörige Kontraktum u**
    - **Berechne v := EVAL(u)** → **Ausgabe v**
  - **Falls s kein Redex ist:**
    - **Stoppe ohne Ergebnis: t besitzt keinen Wert**

Auswertungsbeispiel

– EVAL((λx.⟨(λy.s(y)) x, 0⟩) 0)

= EVAL() Gesamtterm ist ein Redex

*: *$t[s_1, ..t_n/s_1, ..t_n]$ ist hier Ersetzung von Termen (Syntaxbäumen) durch Terme*

Algorithmus EVAL(t) für lässige Auswertung

- **Falls t kanonisch** → **Ausgabe t**
- **Falls t nichtkanonisch:**
  - **Bestimme Hauptargumente $t_1, ..t_n$ von t**
  - **Berechne s_1:=EVAL(t_1), ..., s_n:=EVAL(t_n)**
  - **Bestimme $s := t[s_1, ..s_n/t_1, ..t_n]$** *
  - **Falls s ein Redex ist:**
    - **Bestimme das zugehörige Kontraktum u**
    - **Berechne v :=** EVAL(u) → **Ausgabe v**
  - **Falls s kein Redex ist:**
    - **Stoppe ohne Ergebnis: t besitzt keinen Wert**

Auswertungsbeispiel

– EVAL((λx.⟨(λy.**s**(y)) x, 0⟩) 0)

= EVAL(⟨(λy.**s**(y)) 0, 0⟩) Kontraktum reduziert λ-Term

*: *$t[s_1, ..t_n/s_1, ..t_n]$ ist hier Ersetzung von Termen (Syntaxbäumen) durch Terme*

Algorithmus EVAL(t) für lässige Auswertung

- **Falls t kanonisch** → **Ausgabe t**
- **Falls t nichtkanonisch:**
  - **Bestimme Hauptargumente $t_1, ..t_n$ von t**
  - **Berechne s_1:=EVAL(t_1), ..., s_n:=EVAL(t_n)**
  - **Bestimme $s := t[s_1, ..s_n/t_1, ..t_n]$** *
  - **Falls s ein Redex ist:**
    - **Bestimme das zugehörige Kontraktum u**
    - **Berechne v := EVAL(u)** → **Ausgabe v**
  - **Falls s kein Redex ist:**
    - **Stoppe ohne Ergebnis: t besitzt keinen Wert**

Auswertungsbeispiel

– EVAL((λx.⟨(λy.s(y)) x, 0⟩) 0)

= ⟨(λy.s(y)) 0, 0⟩ Endergebnis: Term ist kanonisch, hat inneren Redex

*: *$t[s_1, ..t_n/s_1, ..t_n]$ ist hier Ersetzung von Termen (Syntaxbäumen) durch Terme*

Implementierung von Substitution und Auswertung

- **Algorithmus zur Bestimmung freier Variablen**
  - `let free_vars t = ... : term -> var list`
  - Codierung der Definition von Variablenvorkommen aus Folie 17
- **Substitutionsalgorithmus zur Berechnung von $t\,\sigma$**
  - `let subst sigma t = ... : (var#term) list -> term -> term`
  - Direkte Codierung der formalen Regeln aus Folie 18 als rekursives ML Programm
- **Auswertungsalgorithmus `compute`: `term -> term`**
  - Direkte Codierung des Algorithmus `EVAL` aus Folie 23
  - Effizienzsteigerung durch Memory-sharing und Caching
- **Einbettung der konkreten Semantik**
  - Redex–Kontrakta Tabelle wird durch Bibliotheksobjekte erzeugt
  - Regelobjekte (Folie 28) für Reduktionsregeln generieren Einträge

Repräsentation des Inferenzsystems

- **Direkte Umsetzung der Konzepte aus Einheit 9**
  - ML-Datentypen für Sequenzen, Regeln, Beweise
  - Bibliotheksobjekte zur Darstellung konkreter Regelschemata

  Korrektheit der Umsetzung ist unmittelbar ersichtlich (Textvergleich)
- **Algorithmen zur Konstruktion konkreter Beweise**
  - Konversion patternbasierter Regelschemata in konkrete Regeln (Dekompositions- und Validierungprogramme)
  - Erzeugung von Beweisobjekten aus Sequenzen und Regeln
  - Extraktion von Evidenztermen aus Beweisen
- **Interface für benutzergesteuerte Beweiskonstruktion**

  ↦ Einheit 16
  - Erzeugung von Beweis-/Inferenzobjekten nach Eingabe von Beweiszielen und Regeln
  - Darstellung des aktuellen Beweisbaums

Implementierung von Sequenzen

Struktur: $x_1 : T_1, \ldots, x_n : T_n \vdash C$

x_i	Variable,
T_i, C	Term
$x_i : T_i$	Deklaration
$x_1 : T_1, \ldots, x_n : T_n$	Hypothesenliste
C	Konklusion

```
abstype  declaration = var # term # bool
  with mk_declaration v t b = abs_declaration(v,t,b)
  and  dest_declaration d = rep_declaration d
;;
lettype  sequent = declaration list # term;;
  let mk_sequent vtb_list term = map mk_declaration term
  and dest_sequent seq = map dest_declaration (fst seq), snd seq
;;
```

Sequenzen können als rudimentäre Beweisbäume angesehen werden

Datenstrukturen für Regeln und Beweise

Inferenzregel: r = **(dec,val)**

- dec Dekomposition: Abbildung von Sequenzen in Listen von Sequenzen
- val Validierung: Abbildung von Listen von Termen und Sequenzen in Terme

Beweis mit Wurzel S: Sequenz S oder Struktur $\pi = (S, r, [\pi_1,..,\pi_n])$

- S Sequenz
- r Inferenzregel
- $\pi_1,\ldots,\pi_n$ Beweise, deren Wurzeln die Teilziele von dec(S) sind

---

```
abstype  rule     = .....
absrectype  proof = sequent # rule # proof list
  with make_proof_node 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);;
```

Repräsentation konkreter Inferenzregeln

- **Bibliothek enthält Regel-Objekte mit konkreten Schemata**

$\Gamma \vdash S \times T$ ⌊**ext** $\langle s, t \rangle$⌋
by `independent_pairFormation`
$\Gamma \vdash S$ ⌊**ext** s⌋
$\Gamma \vdash T$ ⌊**ext** t⌋

```
- RULE: independent_pairFormation
H  ⊢ A × B ext <a, b>
  BY independent_pairFormation ()

  H  ⊢ A ext a
  H  ⊢ B ext b
```

Konstruktor für Beweise (`refine`) wandelt Regelschemata in Taktiken um

Redex-Kontrakta Tabelle wird aus Reduktionsregeln generiert

- **Substitutionen und Parameter explizit dargestellt**

$\Gamma \vdash x_1 : S_1 \to T_1 = x_2 : S_2 \to T_2 \in \mathbb{U}_j$ ⌊Ax⌋
by `functionEquality` $\boxed{x}$
$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j$ ⌊Ax⌋
$\Gamma, x : S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j$ ⌊Ax⌋

```
- RULE: functionEquality
H  ⊢ (x1:a1 → b1) = (x2:a2 → b2)
  BY functionEquality y

  H  ⊢ a1 = a2
  H y:a1 ⊢ !subst(b1; x1.y) = !subst(b2; x2.y)
```

- **Aufruf von Spezialprozeduren durch Verweis auf System**

```
- RULE: arith
H  ⊢ C ext t
  BY arith 𝕌

     Let SubGoals t = CallLisp(ARITH)
  SubGoals
```

Verarbeitung von Inferenzregeln (Refiner)

- **Basisinferenzmaschine ohne eigene "Intelligenz"**
  - Implementierung von `refine` als "rule-compiler"

    Wandelt Inhalte der Regel-Objekte in Taktiken um
- **Schutz gegen unbefugte Manipulation von Beweisen**
  - Bearbeitung von Beweisobjekten muß Refiner benutzen
- **Inferenzmechanismen**
  - Pattern Matching + Term Rewriting für die meisten Regelschemata
  - Entscheidungsprozeduren für `arith` und `equality`
  - β-Reduktion für `compute`
  - Matching zweiter Stufe für Auf- und Rückfalten von Abstraktionen
- **Korrektheit des Systems wird leicht verifizierbar**
  - Überprüfe korrekte Repräsentation der Regeln in Bibliotheksobjekten
  - Verifiziere Implementierung von `refine`

Implementierung der Basisinferenzmaschine

- **Verwende Taktiken als Basismechanismus**
  - Dekomposition erzeugt Teilziele und Validierung
  - Validierung baut Beweisbaum, wenn Blätter bewiesen
  - Ermöglicht einfache Komposition von Regeln
- **Basistaktiken werden aus Regelschemata erzeugt**
  - Einsatz von Pattern Matching und Term Rewriting
  - `deduce_children`: matche mit Hauptziel und instantiiere Teilziele
    Aufruf interner Prozeduren für `arith`, `equality` und `compute`
  - `deduce_validation`: erzeuge Beweisbaum und Extraktterme
- **Unabhängig vom restlichen Beweissystem**
  - Implementierung als separater Prozess möglich
  - Abfrage der Regeln durch Kommunikation mit Bibliothek realisierbar
  - Erlaubt simultane und asynchrone Verwendung mehrerer Refiner

Benutzerdefinierbare Erweiterung des Systems

- **Konservative Erweiterung des Inferenzsystems**
  - Implementierung von Taktiken als programmierte Anwendung von Inferenzregeln
    ↦ Einheit 13
  - Häufig: Spezialisierung des Schließens für neue Konzepte
  - Beweisautomatisierung ohne Risiko inkorrekte Beweise zu generieren
- **Konservative Erweiterung der Sprache**
  - Erzeugung von Abstraktionen und Displayformen für neue Konzepte
  - Semantik der neuen Begriffe ergibt sich direkt aus Definition
  - Inferenzregeln programmierbar durch Auffalten/Falten der Abstraktion und taktische Verarbeitung der definierenden Terme
- **Echte Erweiterung der Sprache**

  Nur für Experten, Mechanismus wurde absichtlich etwas kompliziert gestaltet
  - Erzeugung von primitiven Abstraktionen und Displayformen
  - Direkte Eingabe neuer Regelobjekte
  - Semantik und Inferenzregeln müssen theoretisch abgesichert sein