

Automatisierte Logik und Programmierung

Teil III



Entwurf von Beweisassistentensystemen



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

- **Esistierende Systeme sind Beweisassistentensysteme**
 - **Nuprl**: Konstruktive Typentheorie (CTT) *
 - **MetaPRL**: Infrastruktursystem, Hauptanwendung CTT und CZF
 - **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

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 \$11)

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 §12–14)

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 Theoriestructur mit Querbezügen
 - Abhängigkeitskontrolle und Konsistenzsicherung (zirkuläre Beweise?)
 - **Verschiedene Gestaltungsmöglichkeiten**
 - Datei-textorientiert (Isabelle, Coq, MetaPRL)
 - Konventionell editierbar, leicht austauschbar, muß kompiliert werden
 - Abstrakte Datenbank (Nuprl)
 - Multiuser-fähig, Undo/Redo/Backup, selektive Sichten, ...
- (Details in §15)

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 §15)

KERBESTANDTEILE 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 11



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**,
 $A \rightarrow B$, $A \# B$, $A + B$, **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);;`
- **Definitorsche 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

- **Repräsentation der Grundkonzepte der CTT**
 - Term, Sequenz, Regel, Beweis, Abstraktion, Display Form, ... (vgl. §8)
 - Abstrakte Datentypen formalisieren Definitionen der CTT und liefern Operatoren zur Konstruktion und Analyse konkreter Objekte
- **Abstraktion liefert Kapselung formaler Objekte**
 - Zugriff auf Komponenten nur mit Konstruktoren und Destruktoren
 - z.B. Änderung von Beweisen nur durch Anwendung von Regeln
 - Verhindert unbefugte Manipulation von Theorien & Beweisen
- **Unterstützung für taktische Beweisautomatisierung**
 - Beweise können nur Aufruf von Taktiken verändert werden
 - Taktiken können im Endeffekt nur aus Regeln erzeugt werden

REPRÄSENTATION VON CTT-TERMEN IN ML

Direkte Umsetzung der Definition der Struktur von Termen

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

$opid$ Operatorname

$p_j:F_j$ Parameter, bestehend aus Parameterwert und Parametertyp

$x_1^i, \dots, x_{m_i}^i \cdot t_i$ 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 = ...
  
```

IMPLEMENTIERUNG VON SUBSTITUTION UND AUSWERTUNG

- **Algorithmus zur Bestimmung freier Variablen**

- **let free_vars** $t = \dots : \text{term} \rightarrow \text{var list}$
- Codierung der einheitlichen Definition von Variablenvorkommen in §8

- **Substitutionsalgorithmus $t\sigma$**

- **let subst** σ $t = \dots : (\text{var}\#\text{term}) \text{ list} \rightarrow \text{term} \rightarrow \text{term}$
- Direkte Codierung der formalen Regeln aus §8

$\text{var}\{x:v\}() [t/x]$	$= t$
$\text{var}\{x:v\}() [t/y]$	$= \text{var}\{x:v\}() \quad (y \neq x)$
$(op(b_1; \dots; b_n)) [t/x]$	$= op(b_1[t/x]; \dots; b_n[t/x])$
$(x_1, \dots, x_n \cdot u) [t/x_i]$	$= x_1, \dots, x_n \cdot u$
$(x_1, \dots, x_n \cdot u) [t/y]$	$= x_1, \dots, x_n \cdot u [t/y] \quad *$
$(x_1, \dots, x_j, \dots, x_n \cdot u) [t/y]$	$= (x_1, \dots, z, \dots, x_n \cdot u [z/x_j]) [t/y] \quad **$

*: 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

- **Auswertungsalgorithmus **compute****

- Direkte Codierung des Algorithmus EVAL aus §8
- Effizienzsteigerung durch **Memory-sharing und Caching**

REPRÄSENTATION VON SEQUENZEN

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

x_i	Variable,
T_i, C	Term
$x_i : T_i$	Deklaration
$x_1 : T_1, \dots, x_n : T_n$	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: $r = (\text{dec}, \text{val})$

dec Dekomposition: Abbildung von Sequenzen in Listen von Sequenzen

val Validierung: Abbildung von Listen von Termen und Sequenzen in Terme

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

Z Sequenz

r Inferenzregel

$\pi_1, \dots, \pi_n$ Beweise, deren Wurzeln die Teilziele von $\text{dec}(Z)$ sind

abstype rule =

absrectype proof = sequent # rule # proof list

with **make_proof_node** decs t = **abs_proof**((decs, t), $\diamond$, [])

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);;

EINBETTUNG DER KONKRETEN OBJEKTSPRACHE

● Basisterme der CTT	<i>Operator und Termstruktur</i>	<i>Darstellungsform</i>
	function { }(S ; $x . T$)	$x : S \rightarrow T$
	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

● **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

REPRÄSENTATION DES KONKRETEN INFERENZSYSTEMS

- 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

$\Gamma \vdash x_1 : S_1 \rightarrow T_1 = x_2 : S_2 \rightarrow T_2 \in \mathbb{U}_j \quad |Ax|$
by **functionEquality** $[x]$
 $\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \quad |Ax|$
 $\Gamma, x : S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j \quad |Ax|$

- 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
```

IMPLEMENTIERUNG DER BASISINFERENZMASCHINE

- **Verwende Taktiken als Basismechanismus**
 - Dekomposition erzeugt Teilziele und Validierung
 - Validierung baut Beweisbaum, wenn Blätter bewiesen
- **Basistaktiken werden aus Regelschemata erzeugt**
 - Einsatz von Pattern Matching und Term Rewriting
 - `deduce_children` matched mit Hauptziel und instantiiert Teilziele
 - Aufruf interner Prozeduren für `arith` und `equality`
 - Matching zweiter Stufe für Auf- / Rückfalten von Abstraktionen
 - `deduce_validation` erzeugt Beweisbaum und Extraktterme
- **Korrektheit des Systems wird leicht verifizierbar**
 - Überprüfe korrekte Repräsentation der Regeln in Bibliotheksobjekten
 - Verifiziere Implementierung von `refine`
- **Refiner ist unabhängig vom restlichen Beweissystem**
 - Realisierung als separater Prozess (mit Kommunikation) möglich
 - Erlaubt simultane und asynchrone Verwendung mehrerer Refiner