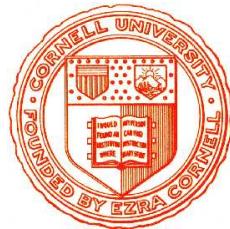


# Automatisierte Logik und Programmierung II

## Teil III

### Aufbau von Beweissystemen



- **Proof Checking:**

- Überprüfung gegebener formaler Beweise durch Computer

## ● Proof Checking:

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● Proof Checking:

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● Proof Editing:

- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln

## ● Proof Checking:

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● Proof Editing:

- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln
- Leicht zu programmieren, Anwendbarkeit abhängig von Benutzerinterface

## ● Proof Checking:

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● Proof Editing:

- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln
- Leicht zu programmieren, Anwendbarkeit abhängig von Benutzerinterface

## ● Taktisches Theorembeweisen:

- Beweiskonstruktion durch programmierte Anwendung von Inferenzregeln
- Entwurf anwendungsspezifischer Inferenzregeln durch Benutzer möglich

## ● **Proof Checking:**

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● **Proof Editing:**

- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln
- Leicht zu programmieren, Anwendbarkeit abhängig von Benutzerinterface

## ● **Taktisches Theorembeweisen:**

- Beweiskonstruktion durch programmierte Anwendung von Inferenzregeln
- Entwurf anwendungsspezifischer Inferenzregeln durch Benutzer möglich
- Flexibel und sicher, gut für mittelgroße Anwendungen

## ● Proof Checking:

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● Proof Editing:

- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln
- Leicht zu programmieren, Anwendbarkeit abhängig von Benutzerinterface

## ● Taktisches Theorembeweisen:

- Beweiskonstruktion durch programmierte Anwendung von Inferenzregeln
- Entwurf anwendungsspezifischer Inferenzregeln durch Benutzer möglich
- Flexibel und sicher, gut für mittelgroße Anwendungen

## ● Beweisprozeduren

- Entscheidungsprozeduren: automatische Tests for entscheidbare Probleme
- Theorembeweiser: vollständige Beweissuche in Prädikatenlogik
- Beweisplaner, Rewriting, Model Checking, Computer Algebra, ...

## ● Proof Checking:

Frühe Systeme, PCC

- Überprüfung gegebener formaler Beweise durch Computer
- Leicht zu programmieren aber extrem mühsam in Anwendung

## ● Proof Editing:

- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln
- Leicht zu programmieren, Anwendbarkeit abhängig von Benutzerinterface

## ● Taktisches Theorembeweisen:

- Beweiskonstruktion durch programmierte Anwendung von Inferenzregeln
- Entwurf anwendungsspezifischer Inferenzregeln durch Benutzer möglich
- Flexibel und sicher, gut für mittelgroße Anwendungen

## ● Beweisprozeduren

fest eingeschränkte Anwendungsbereiche

- Entscheidungsprozeduren: automatische Tests for entscheidbare Probleme
- Theorembeweiser: vollständige Beweissuche in Prädikatenlogik
- Beweisplaner, Rewriting, Model Checking, Computer Algebra, ...
- Effizient aber unflexibel durch Verwendung maschinennaher Techniken

- **Ausdruckstarke Theorien sind unentscheidbar**

- Vollautomatische Beweissysteme nicht praktikabel
- Interaktive Beweiskonstruktion als Basismechanismus

- **Ausdruckstarke Theorien sind unentscheidbar**

- Vollautomatische Beweissysteme nicht praktikabel
- Interaktive Beweiskonstruktion als Basismechanismus

- **Begrenzte Automatisierung möglich**

- Strategische Beweissuche durch **Taktiken** (einfach)
- Entscheidungsprozeduren für Teiltheorien (theoretisch aufwendig)
- Einbindung externer Beweisprozeduren (theoretisch & technisch schwierig)

- **Ausdruckstarke Theorien sind unentscheidbar**

- Vollautomatische Beweissysteme nicht praktikabel
- Interaktive Beweiskonstruktion als Basismechanismus

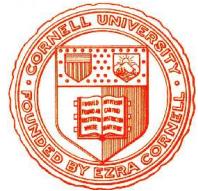
- **Begrenzte Automatisierung möglich**

- Strategische Beweissuche durch **Taktiken** (einfach)
- Entscheidungsprozeduren für Teiltheorien (theoretisch aufwendig)
- Einbindung externer Beweisprozeduren (theoretisch & technisch schwierig)

- **Existierende Systeme**

- **Nuprl**: Konstruktive Typentheorie (ITT)
- **Coq**: Calculus of Constructions
- **Alf**: Martin-Löf Typentheorie (Name ändert sich ständig)
- **PVS**: Klassische Variante der Typentheorie
- **HOL**: Klassische Typentheorie
- **Isabelle**: Infrastruktursystem, Hauptanwendung HOL
- **MetaPRL**: Infrastruktursystem, Hauptanwendung ITT und CZF

# Automatisierte Logik und Programmierung



## Lektion 11

### Interaktive Beweisassistenten



1. Ziele einer Implementierung
2. ML als formale Beschreibungssprache
3. Implementierung der Objektsprache
4. Systemkomponenten
5. Zur Korrektheit der Implementierung

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte
- Benötigt Repräsentation der Metasprache als Programmiersprache

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte
- Benötigt Repräsentation der Metasprache als Programmiersprache

- **Basisterme und -regeln der Theorie implementieren**

- In Systemtabellen oder als explizite Objekte der Bibliothek

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte
- Benötigt Repräsentation der Metasprache als Programmiersprache

- **Basisterme und -regeln der Theorie implementieren**

- In Systemtabellen oder als explizite Objekte der Bibliothek

- **Mechanismen zur Verarbeitung formalen Wissens**

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte
- Benötigt Repräsentation der Metasprache als Programmiersprache

- **Basisterme und -regeln der Theorie implementieren**

- In Systemtabellen oder als explizite Objekte der Bibliothek

- **Mechanismen zur Verarbeitung formalen Wissens**

- Refiner: Anwendung von Inferenzregeln (und Taktiken) auf Beweisziele
  - Basisinferenzmaschine ohne eigene “Intelligenz”

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte
- Benötigt Repräsentation der Metasprache als Programmiersprache

- **Basisterme und -regeln der Theorie implementieren**

- In Systemtabellen oder als explizite Objekte der Bibliothek

- **Mechanismen zur Verarbeitung formalen Wissens**

- **Refiner**: Anwendung von Inferenzregeln (und Taktiken) auf Beweisziele
  - Basisinferenzmaschine ohne eigene “Intelligenz”
- **Library**: Verwaltung des gesamten formalen Wissens

- **Datenstrukturen für Kernbegriffe der Theorie**

- Formalisierung der Metatheorie: Beweis, Regeln, Term, Abstraktion, ...
- Operatoren zur Konstruktion und Analyse konkreter Objekte
- Benötigt Repräsentation der Metasprache als Programmiersprache

- **Basisterme und -regeln der Theorie implementieren**

- In Systemtabellen oder als explizite Objekte der Bibliothek

- **Mechanismen zur Verarbeitung formalen Wissens**

- **Refiner**: Anwendung von Inferenzregeln (und Taktiken) auf Beweisziele
  - Basisinferenzmaschine ohne eigene “Intelligenz”
- **Library**: Verwaltung des gesamten formalen Wissens
- **Editor**: visuelles Benutzerinterface
  - Bearbeitung von Termen, Beweisen, Definitionen, ...

# ML: FORMALE METASPRACHE ALS PROGRAMMIERSPRACHE

- 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)

- 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  $\lambda$ -Kalkül)
- Pattern Matching unterstützt Verständlichkeit komplexe Definitionen

- 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  $\lambda$ -Kalkül)
- Pattern Matching unterstützt Verständlichkeit komplexe Definitionen

- **Erweiterbare polymorphe Typdisziplin**

- Grundkonstrukte: **int**, **bool**, **tok**, **string**, **unit**,  
 **$A \rightarrow B$** ,  **$A \# B$** ,  **$A + B$** ,  **$A \text{ list}$**
- Anwenderdefinierbare abstrakte und rekursive Datentypen
- Typprüfung durch erweiterten Hindley/Milner Typechecking Algorithmus

## ● 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  $\lambda$ -Kalkül)
- Pattern Matching unterstützt Verständlichkeit komplexe Definitionen

## ● Erweiterbare polymorphe Typdisziplin

- Grundkonstrukte: **int**, **bool**, **tok**, **string**, **unit**,  
 $A \rightarrow B$ ,  $A \# B$ ,  $A + B$ ,  $A \text{ list}$
- Anwenderdefinierbare abstrakte und rekursive Datentypen
- Typprüfung durch erweiterten Hindley/Milner Typechecking Algorithmus

## ● Kontrollierte Behandlung von Ausnahmen

- Anwenderdefinierbare Verarbeitung von Laufzeitfehlern

# ABSTRAKTE 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

- **Präzisierung der informalen Definitionen**

- Terme, Regeln, Beweise, Abstraktion, Bibliothek, ...

- **Präzisierung der informalen Definitionen**

- Terme, Regeln, Beweise, Abstraktion, Bibliothek, ...

- **Abstrakte Datentypen kapseln Objekte**

- Kontrollierter Zugriff nur durch Konstruktoren und Destruktoren

- **Präzisierung der informalen Definitionen**

- Terme, Regeln, Beweise, Abstraktion, Bibliothek, ...

- **Abstrakte Datentypen kapseln Objekte**

- Kontrollierter Zugriff nur durch Konstruktoren und Destruktoren

- **Besonderer Schutz für Beweise**

- Änderung nur durch Anwendung von Regeln möglich
  - Verhindert unbefugte Manipulationen und Beweisen

- **Präzisierung der informalen Definitionen**

- Terme, Regeln, Beweise, Abstraktion, Bibliothek, ...

- **Abstrakte Datentypen kapseln Objekte**

- Kontrollierter Zugriff nur durch Konstruktoren und Destruktoren

- **Besonderer Schutz für Beweise**

- Änderung nur durch Anwendung von Regeln möglich
  - Verhindert unbefugte Manipulationen und Beweisen

- **Unterstützung für Beweistaktiken**

- Beweise können nur mit Taktiken verändert werden
  - Taktiken können (im Endeffekt) nur aus Regeln erzeugt werden

# TERME

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

$opid$

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 mkvar t = abs_var t
  and  dvar   v = rep_var v
;;
abstype level_exp = tok + int   with ...
abstype parm   = int + tok + string + var + level_exp + bool with ...
```

# 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
  with mk_assumption v t = abs_declaration(v,t)
  and dest_assumption d = rep_declaration d
;;
lettype sequent = declaration list # term;;
```

**Zugriff auf Sequenzkomponenten durch Beweisdestruktoren**

# REGELN UND BEWEISE

Inferenzregel:  $r = (\text{dec}, \text{val})$

$\text{dec}$  Dekomposition: Abbildung von Sequenzen in Listen von Sequenzen

$\text{val}$  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 mk_proof_goal decs t = abs_proof((decs,t), ◇, [])
  and  refine r p   = let children  = deduce_children r p
                      and validation= deduce_validation r p
                      in children, validation
  and  hypotheses p = fst (fst (rep_proof p))
  and  conclusion p = snd (fst (rep_proof p))
  and  refinement p = fst (snd (rep_proof p))
  and  children   p = snd (snd (rep_proof p))
;;
lettype validation = proof list -> proof;;
lettype tactic      = proof -> (proof list # validation);;
```

- **Regeln repräsentiert als Regelschemata**

- Beweisbaum speichert angewandte Regel in jedem Knoten
- `refine` wandelt Regeln in Taktiken um
- Taktik verwendet Pattern Matching und Term Rewriting
- Erleichtert Komposition von Regeln

- **Regeln repräsentiert als Regelschemata**

- Beweisbaum speichert angewandte Regel in jedem Knoten
- `refine` wandelt Regeln in Taktiken um
- Taktik verwendet Pattern Matching und Term Rewriting
- Erleichtert Komposition von Regeln

- **Taktiken verfeinern Regelbegriff**

- Taktiken sind Dekompositionen
- Anwendung der Dekomposition erzeugt Teilziele und Validierung
- Anwendung der Validierung baut Beweisbaum, wenn Blätter bewiesen

- **Regeln repräsentiert als Regelschemata**

- Beweisbaum speichert angewandte Regel in jedem Knoten
- `refine` wandelt Regeln in Taktiken um
- Taktik verwendet Pattern Matching und Term Rewriting
- Erleichtert Komposition von Regeln

- **Taktiken verfeinern Regelbegriff**

- Taktiken sind Dekompositionen
- Anwendung der Dekomposition erzeugt Teilziele und Validierung
- Anwendung der Validierung baut Beweisbaum, wenn Blätter bewiesen

- **Korrektheit des Systems leicht verifizierbar**

- Überprüfe korrekte Repräsentation der Regeln (Bibliotheksobjekte)
- Verifizierte Implementierung von `refine`

- **Regeln repräsentiert als Regelschemata**

- Beweisbaum speichert angewandte Regel in jedem Knoten
- `refine` wandelt Regeln in Taktiken um
- Taktik verwendet Pattern Matching und Term Rewriting
- Erleichtert Komposition von Regeln

- **Taktiken verfeinern Regelbegriff**

- Taktiken sind Dekompositionen
- Anwendung der Dekomposition erzeugt Teilziele und Validierung
- Anwendung der Validierung baut Beweisbaum, wenn Blätter bewiesen

- **Korrektheit des Systems leicht verifizierbar**

- Überprüfe korrekte Repräsentation der Regeln (Bibliotheksobjekte)
- Verifizierte Implementierung von `refine`

- **Refiner kann ausgelagert werden**

- Prozedur muß `deduce_children` und `deduce_validation` bereitstellen

- **Struktur einer Abstraktion:**  $lhs \equiv rhs$

$lhs$  (Abstraktions-)Term, dessen Unterterme Variablen sind

$rhs$  Term, dessen freie Variablen auch in  $lhs$  frei sind

Neuer Term auf linker Seite wird durch Term der rechten Seite definiert

- **Struktur einer Abstraktion:**  $lhs \equiv rhs$

$lhs$  (Abstraktions-)Term, dessen Unterterme Variablen sind

$rhs$  Term, dessen freie Variablen auch in  $lhs$  frei sind

Neuer Term auf linker Seite wird durch Term der rechten Seite definiert

- **Einfache Repräsentation als Datenstruktur**

- Datentyp: `abstype abstraction = term # term`
- Konstruktor `mk_abstraction` testet Zusatzbedingungen

- **Struktur einer Abstraktion:**  $lhs \equiv rhs$

$lhs$  (Abstraktions-)Term, dessen Unterterme Variablen sind

$rhs$  Term, dessen freie Variablen auch in  $lhs$  frei sind

Neuer Term auf linker Seite wird durch Term der rechten Seite definiert

- **Einfache Repräsentation als Datenstruktur**

- Datentyp: `abstype abstraction = term # term`
- Konstruktor `mk_abstraction` testet Zusatzbedingungen

- **Abstraktionsanwendung ist aufwendiger** (Folie 13)

- Pattern Matching und Instantiierung von Variablen
- Variablen zweiter Stufe beschreiben Terme mit gebundenen Variablen

- **Struktur einer Abstraktion:**  $lhs \equiv rhs$

$lhs$  (Abstraktions-)Term, dessen Unterterme Variablen sind

$rhs$  Term, dessen freie Variablen auch in  $lhs$  frei sind

Neuer Term auf linker Seite wird durch Term der rechten Seite definiert

- **Einfache Repräsentation als Datenstruktur**

- Datentyp: `abstype abstraction = term # term`
- Konstruktor `mk_abstraction` testet Zusatzbedingungen

- **Abstraktionsanwendung ist aufwendiger** (Folie 13)

- Pattern Matching und Instantiierung von Variablen
- Variablen zweiter Stufe beschreiben Terme mit gebundenen Variablen

- **Unabhängige Behandlung der Darstellungsform**

- **Display-Formen** beschreiben textliche Darstellung, Formatierung, Klammerung, Abkürzungen, ...
- Unterstützt vertraute, einfache und verständliche Notationen

- **Bibliothek:** formales mathematisches Lehrbuch

- Definitionen, Sätze, Beweise, Methoden, Anmerkungen, Regeln, ...
- Ermöglicht zusätzliche Inferenzregeln: `lemma`, `extract`, ...

- **Bibliothek:** formales mathematisches Lehrbuch

- Definitionen, Sätze, Beweise, Methoden, Anmerkungen, Regeln, ...
- Ermöglicht zusätzliche Inferenzregeln: `lemma`, `extract`, ...

- **Bibliotheksstruktur**

- Ungeordnete `Kollektion` von `Objekten`
- `Strukturen` (Theorien, Directories, Links,...) können aufgesetzt werden

- **Bibliothek:** formales mathematisches Lehrbuch

- Definitionen, Sätze, Beweise, Methoden, Anmerkungen, Regeln, ...
- Ermöglicht zusätzliche Inferenzregeln: `lemma`, `extract`, ...

- **Bibliotheksstruktur**

- Ungeordnete `Kollektion` von `Objekten`
- `Strukturen` (Theorien, Directories, Links,...) können aufgesetzt werden

- **Bibliotheksobjekte**

Tupel bestehend aus Inhalt und Verwaltungsinformation

`Inhalt:` Abstraktion, Display Form, (Teil-)Beweis, `ML` code, Text, ...

`Art:` `ABS`, `DISP`, `STM`, `CODE`, `COM`, `RULE`, `DIR`, ...

`Eigenschaften:` Status, Name, Aktiv?, Referenzumgebung, ...

`Extra:` Abhängige Objekten, interne Id, sichtbare Position, ...

- **Bibliothek:** formales mathematisches Lehrbuch

- Definitionen, Sätze, Beweise, Methoden, Anmerkungen, Regeln, ...
- Ermöglicht zusätzliche Inferenzregeln: `lemma`, `extract`, ...

- **Bibliotheksstruktur**

- Ungeordnete `Kollektion` von `Objekten`
- `Strukturen` (Theorien, Directories, Links,...) können aufgesetzt werden

- **Bibliotheksobjekte**

Tupel bestehend aus Inhalt und Verwaltungsinformation

`Inhalt:` Abstraktion, Display Form, (Teil-)Beweis, `ML code`, Text, ...

`Art:` `ABS`, `DISP`, `STM`, `CODE`, `COM`, `RULE`, `DIR`, ...

`Eigenschaften:` Status, Name, Aktiv?, Referenzumgebung, ...

`Extra:` Abhängige Objekten, interne Id, sichtbare Position, ...

**In Nuprl wird jedes Objekt als abstrakter Term definiert**

## ● Basisterme

<i>Operator und Termstruktur</i>	<i>Darstellungsform</i>
<b>function</b> { $\lambda$ }( $S$ ; $x.T$ )	$x:S \rightarrow T$
<b>lambda</b> { $\lambda$ }( $x.t$ )	$\lambda x.t$
<b>apply</b> { $\lambda$ }( $f;t$ )	$f t$
⋮	⋮

- Auflistung der Abstraktionsterme in **ML-Operatorentabelle**
- Erstellung von **Display Formen** für jeden Basisterm

## ● Basisterme

<i>Operator und Termstruktur</i>	<i>Darstellungsform</i>
<b>function</b> { $\lambda$ }( $S; x.T$ )	$x:S \rightarrow T$
<b>lambda</b> { $\lambda$ }( $x.t$ )	$\lambda x.t$
<b>apply</b> { $\lambda$ }( $f;t$ )	$f t$
⋮	⋮

- Auflistung der Abstraktionsterme in **ML-Operatorentabelle**
- Erstellung von **Display Formen** für jeden Basisterm

## ● Konstruktoren & Destruktoren

```

let mk_function_term x S T = make_term ('function', []) [] ,S; [x],T
and mk_lambda_term x t = make_term ('lambda', []) [[x],t]
and mk_apply_term f a = make_term ('apply', []) [] ,f; [],a]
    :
let dest_function t = let op,[(),a; [x],b] = dest_term t in x,a,b
and dest_lambda t = let op,[[x],b] = dest_term t in x,b
and dest_apply t = let op,[(),f; [],a] = dest_term t in f,a
    :

```

## ● Basisterme

<i>Operator und Termstruktur</i>	<i>Darstellungsform</i>
<b>function</b> { $\lambda$ }( $S; x.T$ )	$x:S \rightarrow T$
<b>lambda</b> { $\lambda$ }( $x.t$ )	$\lambda x.t$
<b>apply</b> { $\lambda$ }( $f;t$ )	$f t$
⋮	⋮

- Auflistung der Abstraktionsterme in **ML-Operatorentabelle**
- Erstellung von **Display Formen** für jeden Basisterm

## ● Konstruktoren & Destruktoren

```

let mk_function_term x S T = make_term ('function', []) [] ,S; [x],T
and mk_lambda_term x t = make_term ('lambda', []) [[x],t]
and mk_apply_term f a = make_term ('apply', []) [] ,f; [],a]
    :
let dest_function t = let op, [(),a; [x],b] = dest_term t in x,a,b
and dest_lambda t = let op, [[x],b] = dest_term t in x,b
and dest_apply t = let op, [(),f; [],a] = dest_term t in f,a
    :

```

## ● Aufbau durch **Verwendung von Bibliotheksobjekten**

- Operatorentabelle, Konstruktoren, Destruktoren in **Code-Objekten**
- Display Formen und Inferenzregeln sind **explizite Bibliotheksobjekte**

## ● Basisterme

<i>Operator und Termstruktur</i>	<i>Darstellungsform</i>
<b>function</b> { $\lambda$ }( $S; x.T$ )	$x:S \rightarrow T$
<b>lambda</b> { $\lambda$ }( $x.t$ )	$\lambda x.t$
<b>apply</b> { $\lambda$ }( $f;t$ )	$f t$
⋮	⋮

- Auflistung der Abstraktionsterme in **ML-Operatorentabelle**
- Erstellung von **Display Formen** für jeden Basisterm

## ● Konstruktoren & Destruktoren

```

let mk_function_term x S T = make_term ('function', []) [] ,S; [x],T
and mk_lambda_term x t = make_term ('lambda', []) [[x],t]
and mk_apply_term f a = make_term ('apply', []) [] ,f; [],a]
    :
let dest_function t = let op, [(),a; [x],b] = dest_term t in x,a,b
and dest_lambda t = let op, [[x],b] = dest_term t in x,b
and dest_apply t = let op, [(),f; [],a] = dest_term t in f,a
    :

```

## ● Aufbau durch **Verwendung von Bibliotheksobjekten**

- Operatorentabelle, Konstruktoren, Destruktoren in **Code-Objekten**
- Display Formen und Inferenzregeln sind **explizite Bibliotheksobjekte**

→ **schnelle, flexible Implementierung “beliebiger” Theorien**

- Inferenzregeln dargestellt als **Regel-Objekte**

$\Gamma \vdash S \times T \quad \text{[ext } \langle s, t \rangle]$

**by** `independent_pairFormation`

$\Gamma \vdash S \quad \text{[ext } s]$

$\Gamma \vdash T \quad \text{[ext } t]$

- Inferenzregeln dargestellt als **Regel-Objekte**

$\Gamma \vdash S \times T \quad \text{[ext } \langle s, t \rangle]$

by **independent\_pairFormation**

$\Gamma \vdash S \quad \text{[ext } s]$

$\Gamma \vdash T \quad \text{[ext } t]$

- RULE: **independent\_pairFormation**

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

H  ⊢ A ext a
H  ⊢ B ext b
```

- Inferenzregeln dargestellt als **Regel-Objekte**

$\Gamma \vdash S \times T \text{ [ext } \langle s, t \rangle]$

by **independent\_pairFormation**

$\Gamma \vdash S \text{ [ext } s]$

$\Gamma \vdash T \text{ [ext } t]$

- RULE: **independent\_pairFormation**

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

H ⊢ A ext a
H ⊢ B ext b
```

- Substitutionen und Parameter explizit dargestellt

$\Gamma \vdash x_1:S_1 \rightarrow T_1 = x_2:S_2 \rightarrow T_2 \in \mathbb{U}_j \text{ [Ax]}$

by **functionEquality**  $\boxed{x}$

$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \text{ [Ax]}$

$\Gamma, x:S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j \text{ [Ax]}$

## • Inferenzregeln dargestellt als **Regel-Objekte**

$\Gamma \vdash S \times T \text{ [ext } \langle s, t \rangle]$

by **independent\_pairFormation**

$\Gamma \vdash S \text{ [ext } s]$

$\Gamma \vdash T \text{ [ext } t]$

- RULE: **independent\_pairFormation**

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

H ⊢ A ext a
H ⊢ B ext b
```

## • Substitutionen und Parameter explizit dargestellt

$\Gamma \vdash x_1:S_1 \rightarrow T_1 = x_2:S_2 \rightarrow T_2 \in \mathbb{U}_j \text{ [Ax]}$

by **functionEquality**  $\boxed{x}$

$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \text{ [Ax]}$

$\Gamma, \textcolor{teal}{x}:S_1 \vdash T_1[\textcolor{teal}{x}/x_1] = T_2[\textcolor{teal}{x}/x_2] \in \mathbb{U}_j \text{ [Ax]}$

- 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)
```

# IMPLEMENTIERUNG DES KONKREten INFERENCE SYSTEMS

## • Inferenzregeln dargestellt als **Regel-Objekte**

$\Gamma \vdash S \times T \text{ [ext } \langle s, t \rangle]$

by **independent\_pairFormation**

$\Gamma \vdash S \text{ [ext } s]$

$\Gamma \vdash T \text{ [ext } t]$

- RULE: **independent\_pairFormation**

$H \vdash A \times B \text{ ext } \langle a, b \rangle$   
BY **independent\_pairFormation ()**

$H \vdash A \text{ ext } a$

$H \vdash B \text{ ext } b$

## • Substitutionen und Parameter explizit dargestellt

$\Gamma \vdash x_1:S_1 \rightarrow T_1 = x_2:S_2 \rightarrow T_2 \in \mathbb{U}_j \text{ [Ax]}$

by **functionEquality**  $\boxed{x}$

$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \text{ [Ax]}$

$\Gamma, \boxed{x}:S_1 \vdash T_1[\boxed{x}/x_1] = T_2[\boxed{x}/x_2] \in \mathbb{U}_j \text{ [Ax]}$

- RULE: **functionEquality**

$H \vdash (x_1:a_1 \rightarrow b_1) = (x_2:a_2 \rightarrow b_2)$   
BY **functionEquality y**

$H \vdash a_1 = a_2$

$H y:a_1 \vdash !\text{subst}(b_1; x_1.y) = !\text{subst}(b_2; x_2.y)$

## • Aufruf von **Spezialprozeduren** möglich

- RULE: **arith**

$H \vdash C \text{ ext } t$   
BY **arith**  $\mathbb{U}$

Let SubGoals t = CallLisp(ARITH)  
SubGoals

- **Inferenzmaschine** (Refiner)

- Anwendung von Inferenzregeln auf Beweisziele
- Erzeugung noch zu bearbeitender Teilprobleme

# KOMPONENTEN VON BEWEISSYSTEMEN

- **Inferenzmaschine** (Refiner)

- Anwendung von Inferenzregeln auf Beweisziele
- Erzeugung noch zu bearbeitender Teilprobleme

- **Bibliothek** (Library)

- Logische Datenbank zur Verwaltung von formalem Wissen

# KOMPONENTEN VON BEWEISSYSTEMEN

- **Inferenzmaschine** (Refiner)

- Anwendung von Inferenzregeln auf Beweisziele
- Erzeugung noch zu bearbeitender Teilprobleme

- **Bibliothek** (Library)

- Logische Datenbank zur Verwaltung von formalem Wissen

- **Benutzerinterface** (Editor)

- Interface zur Kommunikation mit der Bibliothek
- Visuelle Bearbeitung von Terme, Beweise, Definitionen, ...

- **Inferenzmaschine** (Refiner)

- Anwendung von Inferenzregeln auf Beweisziele
- Erzeugung noch zu bearbeitender Teilprobleme

- **Bibliothek** (Library)

- Logische Datenbank zur Verwaltung von formalem Wissen

- **Benutzerinterface** (Editor)

- Interface zur Kommunikation mit der Bibliothek
- Visuelle Bearbeitung von Terme, Beweise, Definitionen, ...

- **Optionale Komponenten**

- **Extraktion** von Programmen aus Beweisen
- **Evaluator**: Ausführung von Programmen
- **Exportmechanismen**: Ascii Repräsentation, LaTeX, HTML, ...

- **Inferenzmaschine** (Refiner)
  - Anwendung von Inferenzregeln auf Beweisziele
  - Erzeugung noch zu bearbeitender Teilprobleme
- **Bibliothek** (Library)
  - Logische Datenbank zur Verwaltung von formalem Wissen
- **Benutzerinterface** (Editor)
  - Interface zur Kommunikation mit der Bibliothek
  - Visuelle Bearbeitung von Terme, Beweise, Definitionen, ...
- **Optionale Komponenten**
  - **Extraktion** von Programmen aus Beweisen
  - **Evaluator**: Ausführung von Programmen
  - **Exportmechanismen**: Ascii Repräsentation, LaTeX, HTML, ...

**Mechanismen sind unabhängig  
als separate Prozesse implementieren?**

- Basisinferenzmaschine ohne eigene “Intelligenz”
  - Implementierung von `refine`
    - Wandelt Inhalte der Regel-Objekte in Taktiken um

- **Basisinferenzmaschine ohne eigene “Intelligenz”**
  - Implementierung von `refine`
    - Wandelt Inhalte der Regel-Objekte in Taktiken um
- **Schutz gegen unbefugte Manipulation von Beweisen**
  - Bearbeitung von Beweisobjekten muß Refiner benutzen

- **Basisinferenzmaschine ohne eigene “Intelligenz”**
  - Implementierung von `refine`
    - 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`
  - $\beta$ -Reduktion für `compute`
  - Matching zweiter Stufe für Auf- und Rückfalten von Abstraktionen

- **Basisinferenzmaschine ohne eigene “Intelligenz”**

- Implementierung von `refine`
    - 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`
  - $\beta$ -Reduktion für `compute`
  - Matching zweiter Stufe für Auf- und Rückfalten von Abstraktionen

- **Unabhängig vom restlichen Beweissystem**

- Implementierung als separater Prozess möglich
  - Abfrage der Regeln durch Kommunikation mit Bibliothek realisierbar
  - Erlaubt simultane und asynchrone Verwendung mehrerer Refiner

- **Grundoperationen zur Verwaltung von Objekten**

- Erzeugung, Löschen, Umbenennen, Verschieben, (De)Aktivieren, Drucken,
- Strukturierung in Theorien und Directories, Browsen, Suchen, ...

- **Grundoperationen zur Verwaltung von Objekten**

- Erzeugung, Löschen, Umbenennen, Verschieben, (De)Aktivieren, Drucken,
- Strukturierung in Theorien und Directories, Browsen, Suchen, ...

- **Wissensarchivierung**

- **Zertifikate**: Rechtfertigung für gespeicherte Inferenzen
- Explizite Links und logische Abhängigkeiten zwischen Objekten

## ● Grundoperationen zur Verwaltung von Objekten

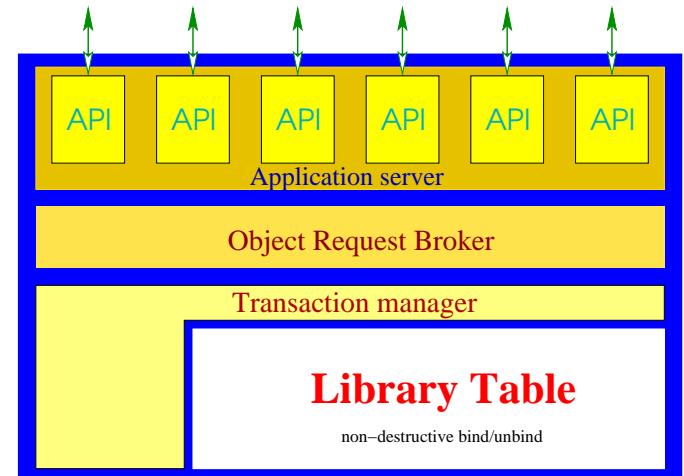
- Erzeugung, Löschen, Umbenennen, Verschieben, (De)Aktivieren, Drucken,
- Strukturierung in Theorien und Directories, Browsen, Suchen, ...

## ● Wissensarchivierung

- Zertifikate: Rechtfertigung für gespeicherte Inferenzen
- Explizite Links und logische Abhängigkeiten zwischen Objekten

## ● Anbindung anderer Komponenten

- Refiner, Editor, externe Systeme als Klienten
- Mehrfache Instanzen möglich



## ● Grundoperationen zur Verwaltung von Objekten

- Erzeugung, Löschen, Umbenennen, Verschieben, (De)Aktivieren, Drucken,
- Strukturierung in Theorien und Directories, Browsen, Suchen, ...

## ● Wissensarchivierung

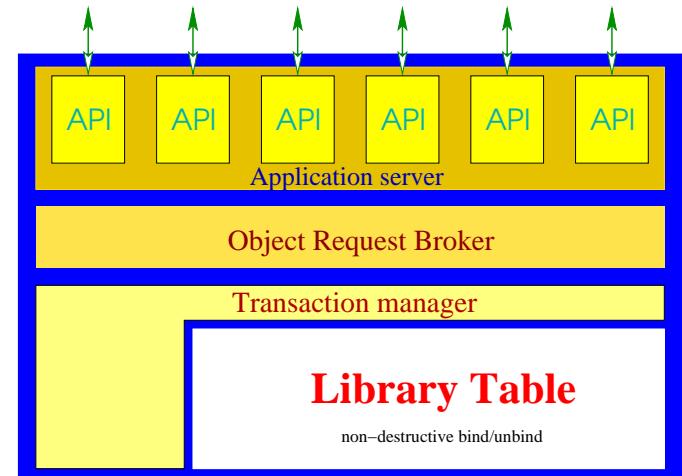
- Zertifikate: Rechtfertigung für gespeicherte Inferenzen
- Explizite Links und logische Abhängigkeiten zwischen Objekten

## ● Anbindung anderer Komponenten

- Refiner, Editor, externe Systeme als Klienten
- Mehrfache Instanzen möglich

## ● Datenbankoperationen

- Dauerhafter Objektspeicher, Konsistenzsicherung
- Backup alter Zustände, Undo, Versionskontrolle
- Transaktionsgesteuerter simultaner Zugriff mehrerer Klienten
- Selektive Sichten auf Teile der Bibliothek



## Visuelle Unterstützung zur Bearbeitung von Wissen

- **Navigator**

- Navigation durch Bibliothek und Aufruf bereitgestellter Operationen

## Visuelle Unterstützung zur Bearbeitung von Wissen

- **Navigator**

- Navigation durch Bibliothek und Aufruf bereitgestellter Operationen

- **Kommandointerface**

- Interpretation von **ML**-Programmen und metasprachlichen Befehlen

## Visuelle Unterstützung zur Bearbeitung von Wissen

- **Navigator**

- Navigation durch Bibliothek und Aufruf bereitgestellter Operationen

- **Kommandointerface**

- Interpretation von ML-Programmen und metasprachlichen Befehlen

- **Beweiseditor**

- Beweisführung und Navigation durch Beweisbäume

## Visuelle Unterstützung zur Bearbeitung von Wissen

- **Navigator**

- Navigation durch Bibliothek und Aufruf bereitgestellter Operationen

- **Kommandointerface**

- Interpretation von ML-Programmen und metasprachlichen Befehlen

- **Beweiseditor**

- Beweisführung und Navigation durch Beweisbäume

- **Termeditor**

- Strukturelles Editieren von Termen in Präsentationsform

## Visuelle Unterstützung zur Bearbeitung von Wissen

- **Navigator**

- Navigation durch Bibliothek und Aufruf bereitgestellter Operationen

- **Kommandointerface**

- Interpretation von ML-Programmen und metasprachlichen Befehlen

- **Beweiseditor**

- Beweisführung und Navigation durch Beweisbäume

- **Termeditor**

- Strukturelles Editieren von Termen in Präsentationsform

- **Objekteditoren**

- Erstellung und Modifikation spezifischer Objekte

## Visuelle Unterstützung zur Bearbeitung von Wissen

- **Navigator**

- Navigation durch Bibliothek und Aufruf bereitgestellter Operationen

- **Kommandointerface**

- Interpretation von ML-Programmen und metasprachlichen Befehlen

- **Beweiseditor**

- Beweisführung und Navigation durch Beweisbäume

- **Termeditor**

- Strukturelles Editieren von Termen in Präsentationsform

- **Objekteditoren**

- Erstellung und Modifikation spezifischer Objekte

- **Unabhängig**

- Mehrere Editoren können gleichzeitig auf dieselbe Library zugreifen

## ● Visuelle Navigation durch Bibliothek

- Keyboard- oder Maus-gesteuertes Durchlaufen
- Patterngesteuerte Namenssuche
- Springen zu gespeicherten Positionen

## ● Visuelle Navigation durch Bibliothek

- Keyboard- oder Maus-gesteuertes Durchlaufen
- Patterngesteuerte Namenssuche
- Springen zu gespeicherten Positionen

## ● Ausführung von Bibliothekskommandos

- Vorbereitete “Buttons” für die wichtigsten Operationen
  - Erzeugung von Objekten, Theorien, Definitionen, Modulen
  - Löschen, Kopieren, Verschieben, Umbenennen, Drucken, ...
  - Import, Export, Drucken und Dokumentation von Theorien
- Aufruf der Operationen öffnet Kommandomenü
- Graphische Interaktion verbesslungsfähig (i.w. Textterminal)

## ● **Visuelle Navigation durch Bibliothek**

- Keyboard- oder Maus-gesteuertes Durchlaufen
- Patterngesteuerte Namenssuche
- Springen zu gespeicherten Positionen

## ● **Ausführung von Bibliothekskommandos**

- Vorbereitete “Buttons” für die wichtigsten Operationen
  - Erzeugung von Objekten, Theorien, Definitionen, Modulen
  - Löschen, Kopieren, Verschieben, Umbenennen, Drucken, ...
  - Import, Export, Drucken und Dokumentation von Theorien
- Aufruf der Operationen öffnet Kommandomenü
- Graphische Interaktion verbesslungsfähig (i.w. Textterminal)

## ● **Undo und Redo für jede Operation**

## ● **Visuelle Navigation durch Bibliothek**

- Keyboard- oder Maus-gesteuertes Durchlaufen
- Patterngesteuerte Namenssuche
- Springen zu gespeicherten Positionen

## ● **Ausführung von Bibliothekskommandos**

- Vorbereitete “Buttons” für die wichtigsten Operationen
  - Erzeugung von Objekten, Theorien, Definitionen, Modulen
  - Löschen, Kopieren, Verschieben, Umbenennen, Drucken, ...
  - Import, Export, Drucken und Dokumentation von Theorien
- Aufruf der Operationen öffnet Kommandomenü
- Graphische Interaktion verbesslungsfähig (i.w. Textterminal)

## ● **Undo und Redo für jede Operation**

## ● **Anpassbar**

- Buttons und Erscheinungsbild durch Bibliotheksobjekte definiert

# BROWSEN DER BIBLIOTHEK MIT NUPRLS NAVIGATOR

```
- TERM: Navigator
MkTHY* OpenThy* CloseThy* ExportThy* ChkThy* ChkAllThys* ChkOpenThy*
CheckMinTHY* MinTHY* EphTHY* ExTHY*

Mill* ObidCollector* NameSearch* PathStack* RaiseTopLoops*
PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FixRefEnvs*
CpObj* reNameObj* EditProperty* SaveObj* RmLink* MkLink* RmGroup*

ShowRefenv* SetRefenvSibling* SetRefenvUsing* SetRefenv* ProveRR* SetInOBJ*
MkTHM* MkML* AddDef* AddRecDef* AddRecMod* AddDefDisp* AbReduce* NavAtAp*
Act* DeAct* MkThyDir* RmThyObj* MvThyObj*

↑↑↑↑ ↑↑↑ ↑↑ ↑ ← <>
↓↓↓↓ ↓↓↓ ↓↓ ↓ → >>

Navigator: [num_thy_1; standard; theories]

Scroll position : 5

List Scroll : Total 159, Point 5, Visible : 10
-----
CODE  TTF  RE_init_num_thy_1
COM   TTF  num_thy_1_begin
COM   TTF  num_thy_1_summary
COM   TTF  num_thy_1_intro
DISP  TTF  divides_df
-> ABS  TTF  divides
STM   TTF  divides_wf
STM   TTF  comb_for_divides_wf
STM   TTF  zero_divs_only_zero
STM   TTF  one_divs_any
-----
```

- Bewegung des **Nav Points** durch Keyboard, Maus, oder Arrow-buttons
- Öffnen von Objekten durch “rechtsgehen” (oder Mittel-Click)
- Sichtbarkeitsbereich kann vergrößert oder verkleinert werden

- **Mathematische Notation erlaubt keine Parser**

- Zu reichhaltig (nicht kontextfrei) und nicht einheitlich geregelt
- Notation ist keine gute Repräsentationform für logische Konzepte

- **Mathematische Notation erlaubt keine Parser**

- Zu reichhaltig (nicht kontextfrei) und nicht einheitlich geregelt
- Notation ist keine gute Repräsentationform für logische Konzepte

- **Typentheorie trennt Notation von Struktur**

- Logische Struktur leichter zu verarbeiten
- Separate Darstellungsform sorgt für verständliche Notation

- **Mathematische Notation erlaubt keine Parser**

- Zu reichhaltig (nicht kontextfrei) und nicht einheitlich geregelt
- Notation ist keine gute Repräsentationform für logische Konzepte

- **Typentheorie trennt Notation von Struktur**

- Logische Struktur leichter zu verarbeiten
- Separate Darstellungsform sorgt für verständliche Notation

- **Editiere logische Struktur von Termen**

- bei gleichzeitiger Präsentation der Darstellungsform auf dem Bildschirm

- **Mathematische Notation erlaubt keine Parser**

- Zu reichhaltig (nicht kontextfrei) und nicht einheitlich geregelt
- Notation ist keine gute Repräsentationform für logische Konzepte

- **Typentheorie trennt Notation von Struktur**

- Logische Struktur leichter zu verarbeiten
- Separate Darstellungsform sorgt für verständliche Notation

- **Editiere logische Struktur von Termen**

- bei gleichzeitiger Präsentation der Darstellungsform auf dem Bildschirm

- **Struktureditor**

- Erzeugung des Termbaums durch **Ausfüllen von Slots** in Darstellungsform
- Kenntnis der genauen Syntax nicht erforderlich
- **Umdenken** erforderlich: keine lineare Eingabe von Text

- **Mathematische Notation erlaubt keine Parser**

- Zu reichhaltig (nicht kontextfrei) und nicht einheitlich geregelt
- Notation ist keine gute Repräsentationform für logische Konzepte

- **Typentheorie trennt Notation von Struktur**

- Logische Struktur leichter zu verarbeiten
- Separate Darstellungsform sorgt für verständliche Notation

- **Editiere logische Struktur von Termen**

- bei gleichzeitiger Präsentation der Darstellungsform auf dem Bildschirm

- **Struktureditor**

- Erzeugung des Termbaums durch **Ausfüllen von Slots** in Darstellungsform
- Kenntnis der genauen Syntax nicht erforderlich
- **Umdenken** erforderlich: keine lineare Eingabe von Text

**Benutzer kann mit verständlicher Notation arbeiten**

## ● **Sichtbare Entwicklung von Beweisen**

- Navigation durch Beweisbaum mit Maus und Keyboard
- Arbeiten im einzelnen Beweisknoten
- Kontrolliertes Interface zum Refiner (via Library)
- Graphische Interaktion verbesserungsfähig (i.w. Textterminal)

## ● **Sichtbare Entwicklung von Beweisen**

- Navigation durch Beweisbaum mit Maus und Keyboard
- Arbeiten im einzelnen Beweisknoten
- Kontrolliertes Interface zum Refiner (via Library)
- Graphische Interaktion verbesserungsfähig (i.w. Textterminal)

## ● **Operationen auf Beweisen**

- Erzeugung von Beweiszielen mit Term-Editor
- Synchrone oder asynchrone Ausführung von Taktiken
- Komprimierung und Expansion bis zu elementaren Schritten
- Verarbeitung von Backup-Beweisen und ‘Schmierblatt’-Beweisen
- Erzeugung von Extrakt-Termen

# TYPISCHER BEWEISKNOTEN

- PRF: intsqrt

```

① # top 1
②
③ 1. x:N
  ⊢ ∃y:N. y² ≤ x ∧ x < (y+1)²
④ BY NatInd 1
⑤ # 1 1
  ....basecase.....
  ⊢ ∃y:N. y² ≤ 0 ∧ 0 < (y+1)²
⑥ BY exR 「0」
  There is 1 hidden subgoal
⑤ # 1 2
  ....upcase.....
  1. x:Z
  2. 0 < x
  3. ∃y:N. y² ≤ x-1 ∧ x-1 < (y+1)²
  ⊢ ∃y:N. y² ≤ x ∧ x < (y+1)²
⑤ BY

```

- PRF: intsqrt

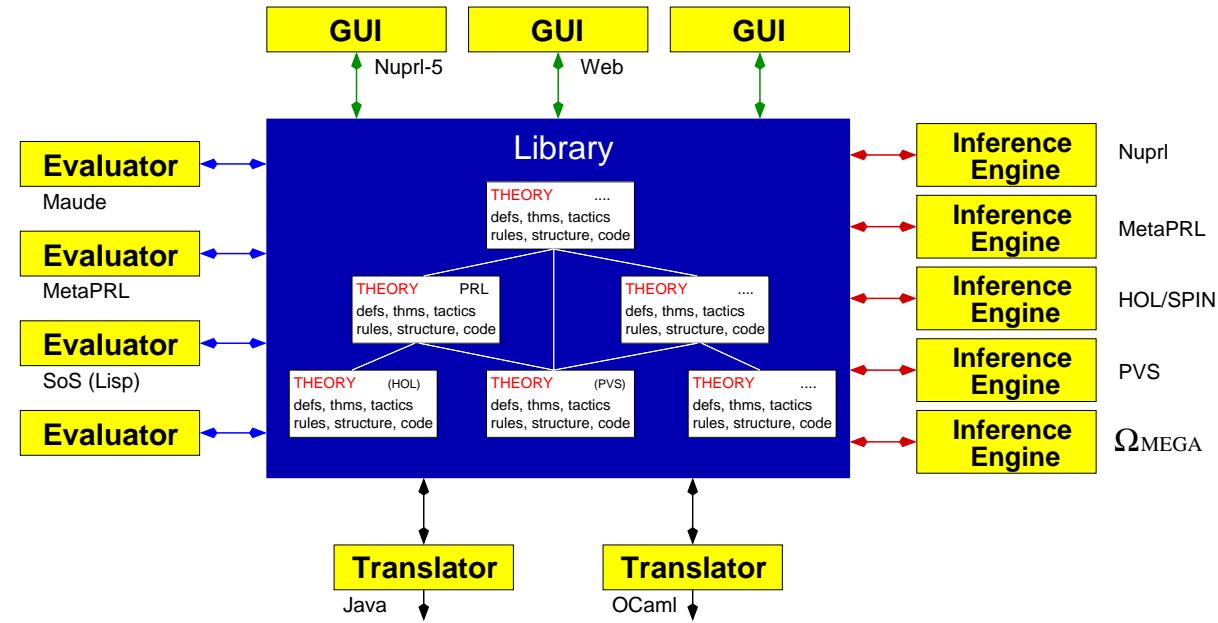
```

① # top 1 2
② .....upcase.....
③ 1. x:Z
  2. 0 < x
  3. ∃y:N. y² ≤ x-1 ∧ x-1 < (y+1)²
  ⊢ ∃y:N. y² ≤ x ∧ x < (y+1)²
④ BY |

```

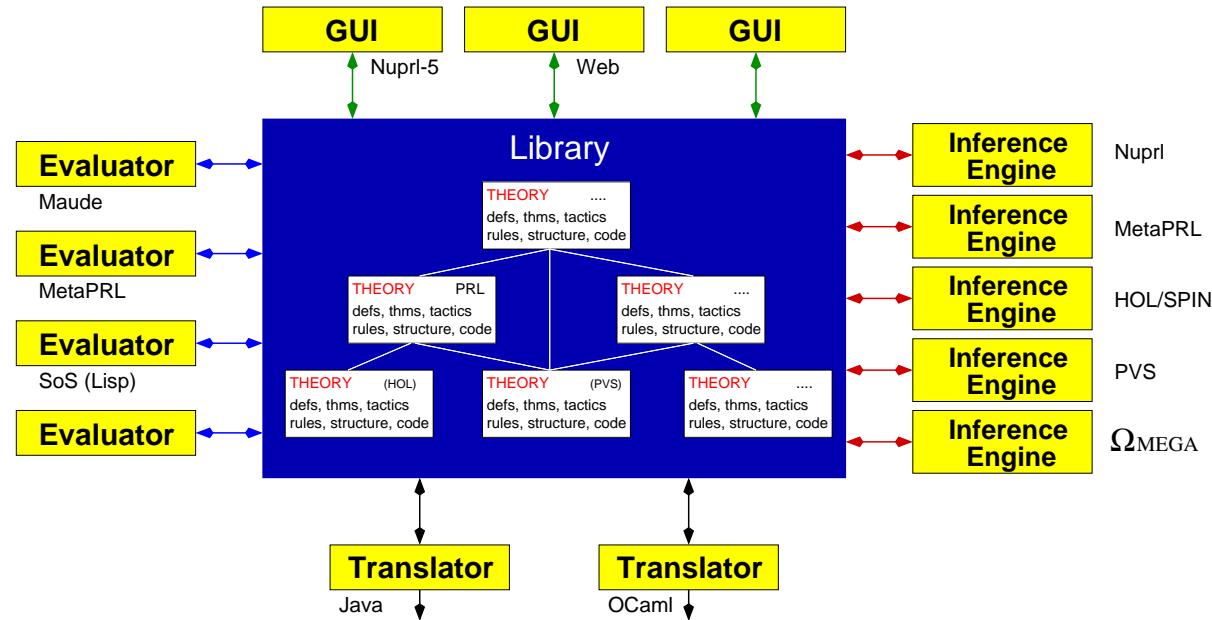
- ① Status und Adresse im Beweisbaum
- ② Annotation des Beweisknotens
- ③ Beweisziel (Sequenz)
- ④ Angewandte Beweistaktik
- ⑤ Teilziele mit Status, Adresse, Sequenz (neue Hypothesen)
- ⑥ Beweise der Teilziele, sofern vorhanden

# NUPRL: GESAMTARCHITEKTUR



## ● Kooperierende Prozesse

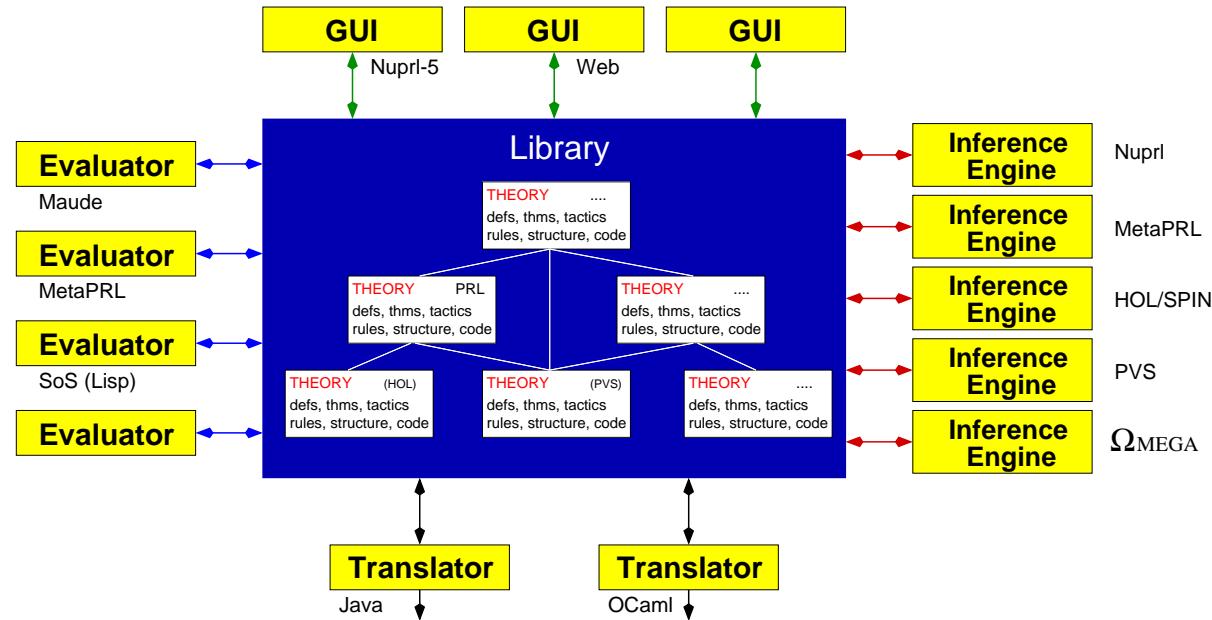
# NUPRL: GESAMTARCHITEKTUR



## ● Kooperierende Prozesse

- Library im Zentrum
- “Beliebig viele” Refiner, Editoren und externe Systeme als Klienten

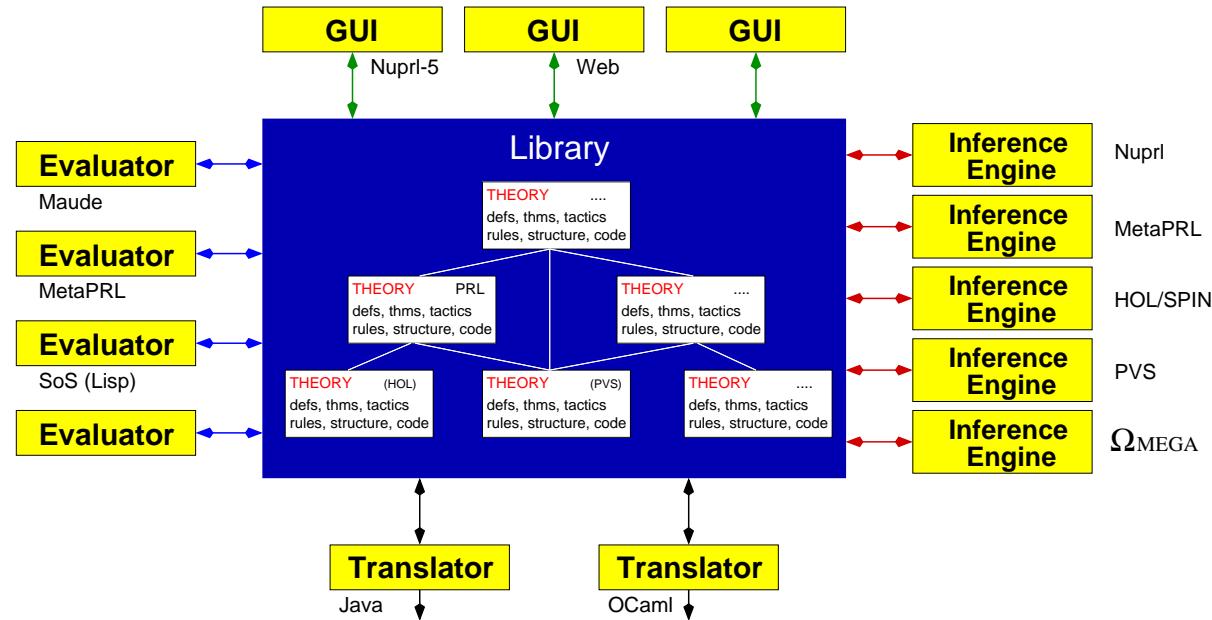
# NUPRL: GESAMTARCHITEKTUR



## ● Kooperierende Prozesse

- Library im Zentrum
- “Beliebig viele” Refiner, Editoren und externe Systeme als Klienten
- Angebundene externe Klienten: MetaPRL, JProver

# NUPRL: GESAMTARCHITEKTUR



## ● Kooperierende Prozesse

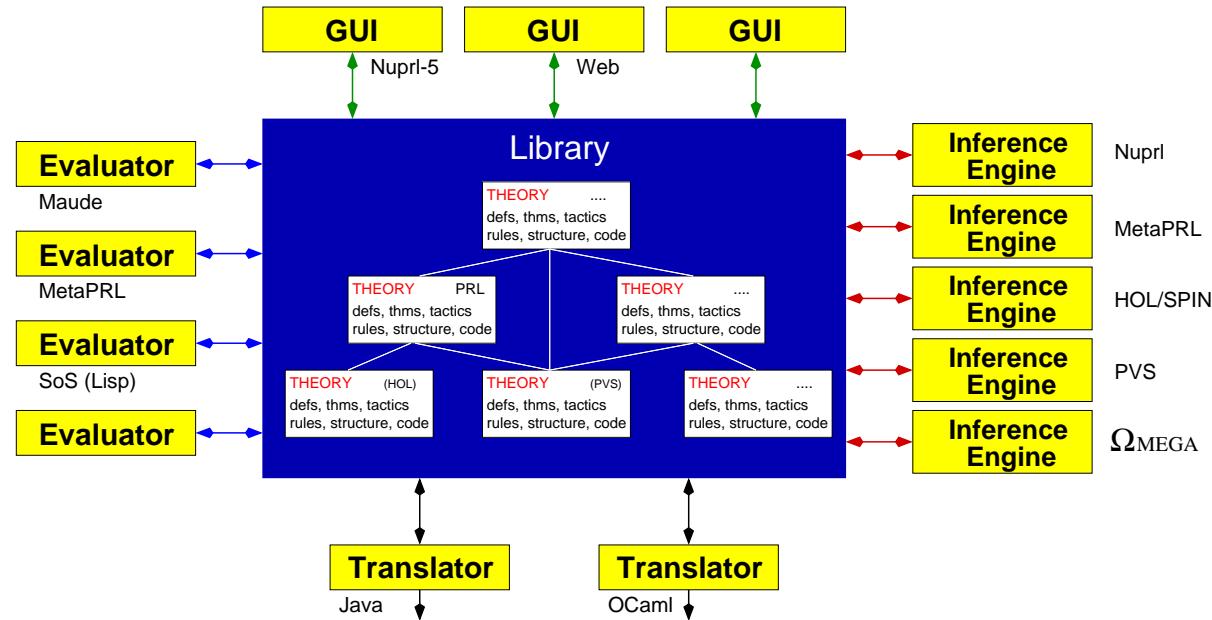
- Library im Zentrum
- “Beliebig viele” Refiner, Editoren und externe Systeme als Klienten
- Angebundene externe Klienten: MetaPRL, JProver

## ● Kooperierende Inferenzmaschinen

- Asynchrones und verteiltes Theorembeweisen

(In Erprobung)

# NUPRL: GESAMTARCHITEKTUR



## ● Kooperierende Prozesse

- Library im Zentrum
- “Beliebig viele” Refiner, Editoren und externe Systeme als Klienten
- Angebundene externe Klienten: MetaPRL, JProver

## ● Kooperierende Inferenzmaschinen

- Asynchrones und verteiltes Theorembeweisen (In Erprobung)

## ● Reflexive Systemstruktur

- Systemdesign in Library enthalten (und veränderbar)