A Tutorial on Lambda Prolog and its Applications to Theorem Proving

Amy Felty
Bell Labs, Lucent Technologies

September 1997

Outline

1. The \( \lambda \)Prolog Language
2. Specifying Logics and Inference Systems
3. Implementing Automatic Theorem Provers
4. Implementing Interactive Tactic Theorem Provers
5. An Implementation of Higher-Order Term Rewriting
6. Encoding the Logical Framework in \( \lambda \)Prolog

Tutorial References

- \( \lambda \)Prolog [Miller & Nadathur]: For information on the language in general and on obtaining the Terzo implementation (implemented in Standard ML), see:
  
  http://www.cis.upenn.edu/~dale/1Prolog/

- Theorem Proving Applications:
  
  
  

General References

For an extensive bibliography on higher-order logic programming and logical frameworks, see:

http://www.cs.cmu.edu/afs/cs/user/fp/www/1fs.html
Part I: The λProlog Language

- Types and Terms
- First-Order Horn Clauses
- Implication and Universal Quantification in Goals (First-Order Harrop Formulas)
- λ-terms and Quantification over Functions and Predicates (Higher-Order Horn Clauses)
- λProlog (Higher-Order Hereditary Harrop Formulas)
- The Module System
- The L_\Lambda Sublanguage

Extensions to Prolog

- polymorphic typing
- Due to hohc:
  - higher-order programming
  - λ-terms as data structures
- Due to fohh:
  - modular programming
  - abstract data types
  - hypothetical reasoning

Sublanguages of λProlog

fohh  ___________  L_\Lambda  ___________  hohh

fohc  ___________  hohc

fohc  first-order Horn clauses
fohh  first-order hereditary Harrop formulas
hohc  higher-order Horn clauses
hohh  higher-order hereditary Harrop formulas

Kinds and Kind Declarations

- Primitive Types
  - System Types
    
```plaintext
kind  o  type.  (for propositions)
kind  int  type.
kind  real  type.
kind  string  type.
```
  - User-Defined Types, e.g.,
    
```plaintext
kind  node  type.
```
- Type Constructors
  
```plaintext
kind  list  type -> type.
kind  pair  type -> type -> type.
```
**Types**

- System types: `o`, `int`, `real`, `string`
- User-introduced primitive types: `node`
- Type variables (denoted by capital letters)
- Constructed types
  - `list string`, `pair int (list string)`
- Functional types (includes predicate types)
  - `int -> real -> string`
  - `int -> int -> o`
  - `o -> int -> o`
  - `(int -> int) -> real`
  - `list A -> (A -> B) -> list B -> o`

  Note: `->` associates to the right, e.g., `τ₁ τ₂ τ₃` denotes `τ₁ → (τ₂ → τ₃)`.

**Clauses and Goals**

<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td><code>o.</code></td>
</tr>
<tr>
<td><code>,</code>, <code>=&gt;</code>, <code>;</code></td>
<td><code>o -&gt; o -&gt; o.</code></td>
</tr>
<tr>
<td><code>:-</code>, <code>,;</code></td>
<td><code>o -&gt; o -&gt; o.</code></td>
</tr>
<tr>
<td><code>pi</code>, <code>sigma</code></td>
<td><code>(A -&gt; o) -&gt; o.</code></td>
</tr>
</tbody>
</table>

- `type` `append` `list A -> list A -> list A -> o.`

**Declarations and Terms**

- `type :: A -> list A -> list A.`
- `infixr :: 5.`
- `type nil list A.`
- `kind a, b, c type.`
- `type f a -> b -> c.`
- `type s a.`
- `type t b.`

**First-Order Horn Clauses**

- Atomic Formulas:
  - `A` of type `o` whose top-level symbol is not a logical constant.
- Goal Formulas:
  - `G ::= T | A | G_{1} ∧ G_{2} | G_{1} ∨ G_{2} | ∃x G`
- Definite Clauses:
  - `D ::= A | G ⊑ A | ∀x D`

**Curried Notation**

- `((f s) t)` or `(f s t)` instead of `f(s, t)`.
- `(f s)` also allowed.
Explicit Quantification

\((\exists x \ B_1 \supset B_2) \equiv \forall x \ (B_1 \supset B_2)\)

type adj, path node \(\rightarrow\) node \(\rightarrow\) o.

path X Y :- adj X Z, path Z Y.

path X Y :- sigma Z \((\text{adj} X Z, \text{path} Z Y)\).

pi x \((\text{pi} y \((\text{path} \ x \ y :- \ \sigma \text{ma} \ Z \((\text{adj} X Z, \text{path} Z Y))))\).

?- path a X.

?- sigma x \((\text{path} a \ x)\).

First-Order Restrictions

- Types in type declarations are of order 0 or 1 (no nesting of \(\supset\) to the left). Also, o only occurs as a target type. Note that the types of \(\text{pi}\) and \(\text{sigma}\) are exceptions.

Example:

\(\text{int, int} \rightarrow \text{int, int} \rightarrow \text{o, int} \rightarrow \text{int} \rightarrow \text{int}\)

But not:

\((\text{int} \rightarrow \text{int}) \rightarrow \text{int, o} \rightarrow \text{o}\)

- Clausal order is either 0 or 1.

Example:

\(\text{adj a b.}\)

path X Y :- adj X Z, path Z Y.

Type and Clausal Order

- Order of a type expression:

  \(\text{ord}(\tau) = 0\) (for atomic type or type variable \(\tau\))

  \(\text{ord}(\tau_1 \rightarrow \tau_2) = \max(\text{ord}(\tau_1) + 1, \text{ord}(\tau_2))\)

- Clausal order:

  \(\text{ord}(A) = 0\) (if \(A\) is atomic or \(\top\))

  \(\text{ord}(B_1 \land B_2) = \max(\text{ord}(B_1), \text{ord}(B_2))\)

  \(\text{ord}(B_1 \lor B_2) = \max(\text{ord}(B_1), \text{ord}(B_2))\)

  \(\text{ord}(\exists x B) = \text{ord}(B)\)

  \(\text{ord}(\forall x B) = \text{ord}(B)\)

First-Order Hereditary Harrop Formulas

- Goal Formulas:

  \(G ::= \top | A | G_1 \land G_2 | G_1 \lor G_2 | \exists x \ G | D \supset G | \forall x G\)

- Definite Clauses:

  \(D ::= A | G \supset A | \forall x \ D\)

- First-order restrictions hold.
Goal-Directed Search

Goal-directed search is formalized with respect to uniform proofs. See [Miller et. al., APAL 91]. Nondeterministic search is complete with respect to intuitionistic provability.

Let $\Sigma$ be a set of type declarations and let $\mathcal{P}$ be set of program clauses. Six primitive operations describe goal-directed search.

**AND** To prove $G_1 \land G_2$ from $\langle \Sigma, \mathcal{P} \rangle$, attempt to prove both $G_1$ and $G_2$ from $\langle \Sigma, \mathcal{P} \rangle$.

**OR** To prove $G_1 \lor G_2$ from $\langle \Sigma, \mathcal{P} \rangle$, attempt to prove either $G_1$ or $G_2$ from $\langle \Sigma, \mathcal{P} \rangle$.

**INSTANCE** To prove $\exists_x G$ from $\langle \Sigma, \mathcal{P} \rangle$, pick a term $t$ of type $\tau$ and attempt to prove $[t/x]G$ from $\langle \Sigma, \mathcal{P} \rangle$.

**AUGMENT** To prove $D \supset G$ from $\langle \Sigma, \mathcal{P} \rangle$, attempt to prove $G$ from $\langle \Sigma, \mathcal{P} \cup \{D\} \rangle$. Note that $D$ is removed after the interpreter succeeds or fails to prove $G$. Thus, the program grows and shrinks dynamically in a stack based manner.

**GENERIC** To prove $\forall_x G$ from $\langle \Sigma, \mathcal{P} \rangle$, introduce a new constant $c$ of type $\tau$ and attempt to prove $[c/x]G$ from $\langle \Sigma \cup \{c\}, \mathcal{P} \rangle$.

**BACKCHAIN** To prove an atomic goal $A$ from $\langle \Sigma, \mathcal{P} \rangle$, the current program $\mathcal{P}$ must be considered.

- If there is a universal instance of a program clause which is equal to $A$, then we have a proof.
- If there is a program clause with a universal instance of the form $G \supset A$ then attempt to prove $G$ from $\langle \Sigma, \mathcal{P} \rangle$.
- If neither case holds then there is no proof of $A$ from $\langle \Sigma, \mathcal{P} \rangle$.

**Logic Variables and Unification**

- In **INSTANCE** a logic variable is used instead of “guessing” a term.
- In **BACKCHAIN** logic variables are used to obtain a universal instance of the clause, and unification is used to match the goal with the head of the clause.
- Note that the **AUGMENT** operation may result in program clauses containing logic variables.
- Because the constant $c$ in **GENERIC** must be new, unification must be modified so that it prevents the variables in the goal and program from being instantiated with terms containing $c$. 
Equality and Conversion

- **α-conversion:**
  \( \lambda x.s = \lambda y.s[y/x] \) provided \( y \) does not occur free in \( s \).
- **β-conversion:**
  \( (\lambda x.s)t = s[t/x] \)
- **η-conversion:**
  \( \lambda x.(sx) = s \) provided \( x \) does not occur free in \( s \).

\[ \lambda \text{Prolog implements } \lambda \text{-conversion as its notion of equality. The following terms are equivalent.} \]

\[ x(f x) y(f y) (g x)(g x) f \]

\[ \lambda \text{Prolog programs cannot determine the name of a bound variable.} \]

Substitution and Quantification

- **Substitute** \( (f \ a) \) for \( X \).
  \[ p(f \ a) : - p y(q \ (f \ a) \ y) \]

- **Substitute** \( (f \ y) \) for \( X \).
  \[ p(f \ y) : - p y(q \ (f \ y) \ y) \]

- **Variable capture** must be avoided.
  \[ p(f \ y) : - p z(q \ (f \ y) \ z) \]

Implication and Universal Quantification in Goals

- **Implication**
  \[ \text{kind } \text{bug, jar } \text{type.} \]
  \[ \text{type } j \text{jar.} \]
  \[ \text{type sterile, heated jar } \text{jar } \text{-> o.} \]
  \[ \text{type dead, bug } \text{insect } \text{-> o.} \]
  \[ \text{type in } \text{insect } \text{-> jar } \text{-> o.} \]

- **Universal quantification**
  \[ \text{sterile } J : - p x(bug } x \text{ = } \text{in } x \text{ J } \text{-> dead } x) . \]
  \[ \text{dead } B : - \text{heated } j, \text{ in } B \text{ j, bug } B . \]
  \[ \text{heated } j . \]
  \[ ? - \text{sterile } j . \]

\[ (\Sigma, P) ? - p x(bug } x \text{ = } \text{in } x \text{ j } \text{-> dead } x) . \]

\[ (\Sigma \cup \{g\}, P) ? - \text{bug } g \text{ = } \text{in } g \text{ j } \text{-> dead } g . \]

\[ (\Sigma \cup \{g\}, P \cup \{\text{bug } g, \text{ in } g \}) ? - \text{dead } g . \]

Logic Variables in Programs

- **Logic variables**
  \[ \text{type reverse list A } \text{-> list A } \text{-> o.} \]
  \[ \text{type rev list A } \text{-> list A } \text{-> list A } \text{-> o.} \]
  \[ \text{type rv list A } \text{-> list A } \text{-> o.} \]

- **Reverse**
  \[ \text{reverse } L \text{ K : - } \]
  \[ \text{pi } L \text{(rev nil L L) = } \]
  \[ \text{pi } L \text{(pi } L \text{(pi } K \text{(pi } M \text{(rev (X::L) K M } : - \text{rev L K (X::M))))}) \]
  \[ = \text{rev L K nil.} \]
  \[ ? - \text{reverse } (1::2::nil) \text{ K .} \]

- **Reverse**
  \[ \text{reverse } L \text{ K : - } \]
  \[ \text{rv nil K } = \]
  \[ \text{pi } L \text{(pi } K \text{(pi } K \text{(rv (X::L) K } : - \text{rv L (X::K))))} \]
  \[ = \text{rv L nil.} \]
Abstract Data Types

type empty stack -> o.
type enter, remove int -> stack -> stack -> o.

?- pi emp\(\pi\) stk\(\pi\) (empty emp =⇒
    pi S\(\pi\) (pi X\(\pi\) (enter X S (stk X S))) =⇒
    pi S\(\pi\) (pi X\(\pi\) (remove X (stk X S) S)) =⇒
    sigma S1 (sigma S2 (sigma S3 (sigma S4 (sigma S5\(\sigma\)
        (empty S1, enter 1 S1 S2, enter 2 S2 S3,
        remove A S3 S4, remove B S4 S5))))).
A == 2, B == 1.

?- pi emp\(\pi\) stk\(\pi\) (... =⇒
    sigma U\(\sigma\) (empty U, enter 1 U V))).
no.

The term \((stk 1 emp)\) is formed as an instance of \(V\), but the goal
fails because emp cannot escape its scope.

Higher-Order Horn Clauses

- Atomic Formulas:
  \(A\) is a term of type \(o\) whose top-level symbol is not a logical
  constant, and which does not contain any occurrences of \(\Box\).

- Rigid Atomic Formulas:
  \(A_r\) is an atomic formula whose top-level symbol is also not a
  variable.

- Goal Formulas:
  \(G ::= \top \mid A \mid G_1 \land G_2 \mid G_1 \lor G_2 \mid \exists x \, G\)

- Definite Clauses:
  \(D ::= A_r \mid G \supset A_r \mid \forall x \, D\)

- No restrictions on order of types. Restrictions on clausal or-
der still hold. Terms instantiating \(x\) also cannot contain any
  occurrences of \(\Box\).

Examples of Higher-Order Programs

type mappred \((A \rightarrow B \rightarrow o) \rightarrow list A \rightarrow list B \rightarrow o\).
type forvery \((A \rightarrow o) \rightarrow list A \rightarrow o\).
type forsome \((A \rightarrow o) \rightarrow list A \rightarrow o\).
type sublist \((A \rightarrow o) \rightarrow list A \rightarrow list A \rightarrow o\).
mappred P nil nil.
mappred P X LS Y L = P X Y, mappred P L K.

forevery P nil.
forevery P X LS : - P X, forvery P L.

forsome P X LS : - P X ; forsome P L.
s sublist P X LS X KL : - P X, sublist P L K.
s sublist P X LS K : - sublist P L K.
s sublist P nil nil.

The mappred Program

type mappred \((A \rightarrow B \rightarrow o) \rightarrow list A \rightarrow list B \rightarrow o\).
mappred P nil nil.
mappred P X LS Y L = P X Y, mappred P L K.

type age person \rightarrow int \rightarrow o.
age bob 23.
age sue 24.
age ned 23.

?- mappred age \(\text{ned::bob::sue::nil}\) L.
L = (23::23::24::nil).

?- mappred age L (23::23::24::nil).
L = (ned::bob::sue::nil);
L = (bob::ned::sue::nil).

?- mappred \((x/y):(age y x)) (23::24::nil) K.
K = (bob::sue::nil);
K = (ned::sue::nil).
The sublist Program

```haskell
type sublist (A -> o) -> list A -> list A -> o.
sublist P (X::L) (X::K) :- P X, sublist P L K.
sublist P (X::L) K :- sublist P L K.
sublist P nil nil.

type male, female person -> o.

male bob.

female sue.

male ned.

?- sublist male (ned::bob::sue::nil) L. L == (ned::bob::nil); L == (ned::nil); L == (bob::nil); no
```

The forvery Program

```haskell
type forvery (A -> o) -> list A -> o.

forvery P nil.

forvery P (X :: L) :- P X, forvery P L.

age bob 23.

age sue 24.

age ned 23.

?- forvery (x\(\sigma y\(\text{age } y\))) (ned::bob::sue::nil). yes.

?- forvery (x\(\text{age } x\)) (ned::bob::sue::nil). no.

?- forvery (x\(\text{age } x\)) (ned::bob::nil). A == 23.
```

The forsome Program

```haskell
type forsome (A -> o) -> list A -> o.

forsome P (X :: L) :- P X ; forsome P L.

male bob.

female sue.

male ned.

?- forsome female (ned::bob::sue::nil) L. yes.
```

Computing with \(\lambda\)-terms

```haskell
type mapfun (A -> B) -> list A -> list B -> o.
type reducefun (A -> B -> B) -> list A -> B -> B -> o.

mapfun F nil nil.

mapfun F (X :: L) ((F X) :: K) :- mapfun F L K.

reducefun F nil Z Z.

reducefun F (H::T) Z (F H R) :- reducefun F T Z R.
```
The mapfun Program

type mapfun (A -> B) -> list A -> list B -> o.
mapfun F nil nil.
mapfun F (X :: L) ((F X) :: K) :- mapfun F L K.

type g i -> i -> i.
type a,b i.
?- mapfun (x\(g a x)) (a::b::nil) L.
L == ((g a a)::(g a b)::nil).
The interpreter forms the terms
  ((x\(g a x)) a)  and  ((x\(g a x)) b)
and reduces them.

?- mapfun F (a::b::nil) ((g a a)::(g a b)::nil).
F == x\(g a x);
no.
The interpreter tries the 4 unifiers for  (F a) and  (g a a) in the following order.
  x\(g x x)  x\(g a x)  x\(g x a)  x\(g a a)

Computing with \(\lambda\) terms is not Functional Programming

An alternative definition of mapfun illustrating that it is weaker than mappred.

type mapfun (A -> B) -> list A -> list B -> o.
mapfun F L K :- mappred (x\y\(y = F x)) L K.

Computing with \(\lambda\) terms involves unification and conversion, but not function computation. The following goal is not provable.

?- mapfun F (a::b::nil) (c::d::nil).
no.

The reducefun Program

type reducefun (A -> B -> B) -> list A -> B -> B -> o.
reducefun F nil Z Z.
reducefun F (H::T) Z (F H R) :- reducefun F T Z R.

?- reducefun (x\y\(x + y)) ((3::4::8::nil) 6 R, S is R).
  R == 3 + (4 + (8 + 6)).
  S == 21.

?- reducefun F (4::8::nil) 6 (1 + (4 + (1 + (8 + 6))).
  F == x\y\(1 + (4 + (1 + (8 + 6)))
  F == x\y\(1 + (x + (1 + (8 + 6)));
  F == x\y\(1 + (x + y));
  no.

?- pi z\(reducefun F (4::8::nil) z (1 + (4 + (1 + (8 + z))))).
  F == x\y\(1 + (x + y));
  no.

Higher-Order Hereditary Harrop Formulas

- Goal Formulas:
  
  \[ G ::= \top | A | G_1 \land G_2 | G_1 \lor G_2 | \exists x \, G | D \supset G | \forall x \, G \]

- Definite Clauses:
  
  \[ D ::= A | G \supset A | \forall x \, D \]

- No restrictions on order of types or on clausal order. The restriction that atomic terms and substitution terms cannot contain occurrences of \(\supset\) still holds.
- New restriction: the head of any atomic formula that appears in a \(D\) formula cannot be a variable that is essentially existentially quantified.
Essentially Existential and Universal Occurrences

- If a subformula occurs to the left of an even number of occurrences of ∃ in a goal formula, then it is a positive subformula occurrence. If it occurs to the left of an odd number of occurrences of ∃, it is a negative subformula occurrence. These definitions are reversed for clauses.
- A bound variable occurrence is essentially universal if it is bound by a positive occurrence of a universal quantifier, by a negative occurrence of an existential quantifier, or by a λ-abstraction. Otherwise, it is essentially existential.
- In terms of the λProlog interpreter, variables that get instantiated with logic variables are essentially existential, while variables that get instantiated with new constants are essentially universal.

Logical Foundation of λProlog

- Based on Church’s Simple Theory of Types [Church 40, JSL]
  - The type o for formulas, and the quantifiers pi and sigma adopted directly.
- Differences
  - Different logical connectives are taken as primitive.
  - Intuitionistic instead of classical logic is used.
  - Type variables and constructors are allowed.

Discharging a Constant from a Term

\[ \langle \Sigma, P \rangle \vdash pi \ y (append \ (1::2::nil) \ y \ X) \]
\[ \langle \Sigma \cup \{k\}, P \rangle \vdash append \ (1::2::nil) \ k \ X. \]

The term \((1::2::k)\) is formed as an instance of \(X\), but as seen before, the goal fails because \(k\) cannot escape its scope.

\[ \langle \Sigma, P \rangle \vdash pi \ y (append \ (1::2::nil) \ y \ (H \ y)) \]
\[ \langle \Sigma \cup \{k\}, P \rangle \vdash append \ (1::2::nil) \ k \ (H \ k). \]

The terms \((H \ k)\) and \((1::2::k)\) are unified. Of the two unifiers, \(\omega\ (1::2::k)\) and \(\omega\ (1::2::w)\), only the second is possible. It is the result of discharging \(k\) from the term \((1::2::k)\).
λProlog’s Module System

1. One-line header
   module moduleName.
2. Preamble (4 directives)
   accumulate, import, local, localKind
3. Declarations (which form the signature) and clauses

---

Example Modules using accumulate

module mod1.
kind item type.
type p,q item -> o.
p X := q X.
module mod2.
accumulate mod1.
type a item.
q a.
module mod3.
kind item type.
type p,q item -> o.
type a item.
p X := q X.
q a.

Modules mod2 and mod3 have the same signature and program.

---

The accumulate directive

- Used to incorporate other modules as if they were typed at the beginning of the current module.
- The signature of the module and of all of the modules named by the accumulate directive must be successfully pairwise merged.
- Two signatures can be _merged_ when:
  - If a token has a kind declaration in both signatures, the declarations must be identical.
  - If a token has a type declaration in both signatures, the types must be the same up to renaming of type variables.
  - If a token has a type declaration in both signatures, if it also has an infix declaration in one signature, it must have the same infix declaration in the other.

---

Declaring local scope to constants

- Universal quantification in goals, e.g., ∀x(D ⊃ G), can be used to introduce a new scoped constant. Note that this formula is equivalent to (∃x D) ⊃ G.
- Modules as existentially quantified program clauses provides local scoping:

  \[ E ::= D | \exists x \ E \mid E_1 \land E_2 \]

- No need to change the interpreter. A goal of the form \( E \supset G \) can be expanded to one that doesn’t contain existential quantifiers in clauses by using the equivalence \((∃x D) ⊃ G \equiv ∀x(D ⊃ G)\).
Example Module using local

module stack.

kind stack    type -> type.
type empty    stack A -> o.
type enter, remove A -> stack A -> stack A -> o.

local emp     stack A.
local stk     A -> stack A -> stack A.
empty emp.
enter X S (stk X S).
remove X (stk X S) S.

Example Module using import

module int_stack.
import stack.

type stack_test int -> int -> o.

stack_test A B :-
    signa S1\(signa S2\(signa S3\(signa S4\(signa S5\ (empty S1, enter 1 S1 S2, enter 2 S2 S3, remove A S3 S4, remove B S4 S5))))).

?- stack_test A B. A == 2, B == 1.

The import directive

module mod1.
import mod2 mod3.

- The clauses in mod2 and mod3 are available during the search for proofs of the body of clauses in mod1. Logically...
- Suppose $E_2$ and $E_3$ are the formulas associated with mod2 and mod3 and $G \supset A$ is a clause in mod1.
- Then the clause used by the interpreter is really the one that is equivalent to

$$((E_2 \land E_3) \supset G) \supset A$$

after existential quantifiers in $E_2$ and $E_3$ are changed to universal quantifiers over $G$.

The $\Lambda$ Sublanguage

- Restricts $\lambda$Prolog by placing the following restriction on variables:
  For every subterm in formula $B$ of the form $x y_1 \ldots y_n$ ($n \geq 0$) where $x$ is essentially existentially quantified in $B$, the variables $y_1, \ldots, y_n$ must be distinct variables that are essentially universally quantified within the scope of the binding for $x$.
- Simplifies $\beta$-conversion: all $\beta$-redexes have the form $ty_1 \ldots y_n$ where we can assume that $t$ has the form $\lambda y_1 \ldots \lambda y_n t'$. By $\beta$-reduction $(\lambda y_1 \ldots \lambda y_n t')y_1 \ldots y_n$ simply reduces to $t'$.
- Simplifies unification: it is decidable and most general unifiers exist; it can be implemented with a simple extension to first-order unification.
\textbf{L}_\lambda \text{ Unification Examples}

- An example in \textit{L}_\lambda
  \[
x \land y \land (g \ x \ z) \land (\forall y) = \forall \ w \ (x \ w)
  \]
  \[
  U == x \land y \land (\forall y) \quad I == \forall \ (g \ (\forall w) \ (\forall u))
  \]

- An example that is not in \textit{L}_\lambda
  \[
  (F \ a) = (g \ a \ a)
  \]
  \[
  F == x \land (g \ x \ x) \quad F == x \land (g \ a \ x)
  \]
  \[
  F == x \land (g \ x \ a) \quad F == x \land (g \ a \ a)
  \]

\textbf{Interpreters for \lambda Prolog}

We distinguish between two kinds of interpreters for \lambda Prolog.

- Specifications are with respect to a non-deterministic interpreter (which is complete with respect to intuitionistic provability).

- The deterministic interpreter which provides an ordering on clause and goal selection and uses a depth-first search discipline with backtracking as in Prolog is used for actual implementations.

\textbf{Part II: Specifying Logics and Inference Systems}

- Specifying Syntax
- A Program for Computing Negation Normal Forms
- Example Specifications
  - Natural Deduction
  - A Sequent System
  - A Modal Logic Specification
  - $\beta\eta$-Convertibility for the Untyped \lambda-Calculus
  - Evaluation for a Functional Language
- Correctness of Specifications

\textbf{Why Theorem Proving as an Application?}

- Specification
  - The declarative nature of programs allows natural specifications of a variety of logics as well as of the tasks involved in theorem proving.
  - \lambda-terms are useful for expressing the higher-order abstract syntax of object logics.
  - Universal quantification and implication in goal formulas are useful for specifying various inference systems naturally and directly.

- Implementation
  - Search is fundamental to theorem proving.
  - Unification can be used to solve certain equations between objects (e.g., formulas, proofs).
  - \lambda-conversion can be used to implement substitution directly.
A First-Order Object Logic

kind form type.
kind i type.
type and, or, imp form -> form -> form.
type neg form -> form.
type forall (i -> form) -> form.
type exists (i -> form) -> form.
type false form.
type c i. infixlor 4.
type f i -> i -> i. infixl and 5.
type q form. infixr imp 6.
type p i -> form.

\( \forall x \exists y (p(x) \lor p(y)) \)

(forall x\((exists y((p x) imp (p y))))

Negation Normal Form

\( \neg \neg A \equiv A \)
\( \neg (A \land B) \equiv \neg A \lor \neg B \)
\( \neg (A \lor B) \equiv \neg A \land \neg B \)
\( \neg (A \land B) \equiv A \land \neg B \)
\( \neg (A \lor B) \equiv \exists x (\neg A) \)
\( \neg (\forall x A) \equiv \forall x (\neg A) \)

Negation Normal Form Clauses I

kind nnf form -> form -> o.
nnf (A and B) (C and D) :- nnf A C, nnf B D.
nnf (A or B) (C or D) :- nnf A C, nnf B D.
nnf (A imp B) (C or D) :- nnf (neg A) C, nnf B D.
nnf (forall A) (forall B) :- pi x\((nnf (A x) (B x))\).
nnf (exists A) (exists B) :- pi x\((nnf (A x) (B x))\).

Negation Normal Form Clauses II

nnf (neg (neg A)) B :- nnf A B.
nnf (neg (A and B)) (C or D) :-
  nnf (neg A) C, nnf (neg B) D.
nnf (neg (A or B)) (C and D) :-
  nnf (neg A) C, nnf (neg B) D.
nnf (neg (A imp B)) (C and D) :-
  nnf A C, nnf (neg B) D.
nnf (neg (forall A)) (exists B) :-
  pi x\((nnf (neg (A x)) (B x))\).
nnf (neg (exists A)) (forall B) :-
  pi x\((nnf (neg (A x)) (B x))\).
Natural Deduction I

\[
\begin{align*}
A & \quad B \\ \frac{A \land B}{A} & \quad \frac{B}{A \lor B} & \quad \frac{(A) B}{A \supset B} \\
\hline
\end{align*}
\]

\[\frac{(A) \quad \frac{y/x][A]}{\exists x A} \quad \frac{[y/x][A]}{\forall x A}}{\forall x A} \quad \frac{\exists x A}{[y/x][A] \lor B} \quad \frac{A \lor B \lor x p(x)}{\exists x p(x) \lor E}
\]

Proviso on \(\forall I\): \(y\) cannot appear free in \(\forall x A\), or in any assumption on which the deduction of \([y/x][A]\) depends.

Natural Deduction II

\[
\begin{align*}
A \land B & \quad \frac{A \land B}{A} & \quad \frac{A \lor B}{B} & \quad \frac{A \supset B}{\forall x A} & \quad \frac{[y/x][A]}{\forall x A} \\
\hline
\end{align*}
\]

\[\frac{A \lor B}{C} \quad \frac{C \lor E}{C} \quad \frac{\exists x A}{[y/x][A] \lor B} \quad \frac{\exists x A}{B \lor E}
\]

Proviso on \(\exists E\) rule: \(y\) cannot appear free in \(\exists x A\), in \(B\), or in any assumption on which the deduction of the upper occurrence of \(B\) depends.

A Natural Deduction Proof

\[
\begin{align*}
p(a) \lor p(b) & \quad \frac{p(a)}{\exists x p(x)} \quad \frac{p(b)}{\exists x p(x)} \\
\hline
\exists x p(x) \lor E
\end{align*}
\]

Specifying Natural Deduction Rules

\[
\begin{align*}
A & \quad B \quad \frac{A \land B}{A} & \quad \frac{B}{A \lor B} & \quad \frac{A \lor B}{A \land B} \\
\hline
\end{align*}
\]

type proof npref -> form -> o.

type and_i npref -> npref -> npref.

proof (and_i P1 P2) (A and B) :-
  proof P1 A, proof P2 B.

proof (or_i P) (A or B) :-
  proof P A; proof P B.
Specifying Existential Introduction

\[
\begin{array}{c}
\frac{[t/x]A}{\exists x A}
\end{array}
\quad \vdash
\]

<table>
<thead>
<tr>
<th>type</th>
<th>exists_i nprf -&gt; nprf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>proof</td>
<td>(exists_i P) (exists A) :- proof P (A T).</td>
</tr>
<tr>
<td>type</td>
<td>exists_i i -&gt; nprf -&gt; nprf.</td>
</tr>
<tr>
<td>proof</td>
<td>(exists_i T P) (exists A) :- proof P (A T).</td>
</tr>
</tbody>
</table>

Specifying Universal Introduction

\[
\begin{array}{c}
\frac{[y/x]A}{\forall x A}
\end{array}
\quad \vdash
\]

Proviso on \(\forall\)-I: \(y\) cannot appear free in \(\forall x A\), or in any assumption on which the deduction of \([y/x]A\) depends.

|proof| (forall_i P) (forall A) :- pi y\(\backslash\)proof (P y) (A y). |
|type| forall_i (i -> nprf) -> nprf. |

Specifying the Discharge of Assumptions

\[
\begin{array}{c}
\frac{(A) B}{A \supset B}
\end{array}
\quad \vdash
\]

|proof| (imp_i P) (A imp B) :- pi pA\(\backslash\)((proof pA A) \Rightarrow (proof (P pA) B)). |
|type| imp_i (nprf -> nprf) -> nprf. |

Example Execution

\[
\begin{array}{c}
\frac{q}{q \supset q}
\end{array}
\quad \vdash
\]

|proof| (imp_i P) (A imp B) :- pi pA\(\backslash\)((proof pA A) \Rightarrow (proof (P pA) B)). |
|proof| (\(\Sigma, P\)) \(- pi pA\(\backslash\)((proof pA A) \Rightarrow (proof (R1 pA) q)). |
|proof| (\(\Sigma \cup \{pA\}, P\)) \(- (proof pA q) \Rightarrow (proof (R1 pA) q)). |
|proof| (\(\Sigma \cup \{pA\}, P \cup \{proof pA q\}\)) \(- proof (R1 pA) q). |

Unification Problem: \(R = (imp\_i R1, (R1 pA) = pa\)
Solution: \(R1 := x\backslash x, R := (imp\_i x\backslash x)\)
Not a Solution: \(R1 := x\backslash pA\)
Elimination Rules

\[
\frac{A \land B}{A} \land E \\
\frac{A \land B}{B} \land E \\
\frac{A}{B} \lor E \\
\frac{A \lor B}{\forall x A} \lor E \\
\frac{A \lor B}{C} \lor E \\
\frac{\neg A}{\bot} \neg E \\
\frac{\exists x A}{B} \exists E
\]

\[\begin{align*}
\Gamma & \rightarrow A \\
\Gamma & \rightarrow A \land B \\
\Gamma & \rightarrow B \land \Gamma \\
\Gamma & \rightarrow A \lor B \\
\Gamma & \rightarrow B \lor \Gamma \\
\Gamma & \rightarrow (\forall x A) \lor \Gamma \\
\Gamma & \rightarrow (\exists x A) \exists \Gamma \\
\Gamma & \rightarrow (\forall x A) \lor \Gamma \\
\Gamma & \rightarrow (\exists x A) \exists \Gamma
\end{align*}\]

- Proof \((\text{id}_E)\) \(A\) :- proof \((A \land B)\); proof \((B \land A)\).
- Proof \((\forall \text{e} P) (A T) :- proof \((P (A \lor B))\).
- Proof \((\exists \text{e} P) (A \lor B) :- proof \((P (A \land B))\).
- Proof \((\text{imp}_e P) (A \land B) :- proof \((P (A \lor B))\).
- Proof \((\text{neg}_e P) (A \lor B) :- proof \((P (A \land B))\).

Remaining Rules

\[\begin{align*}
\frac{(A)}{\bot} \neg I \\
\frac{\bot \lor I}{A} \\
\frac{(\neg A)}{\bot} \neg C
\end{align*}\]

- Proof \((\text{neg}_I P) \text{ (neg A)} :-\)
  \(\neg \bot p \land (\text{proof (p A)} \Rightarrow \text{proof (p B) false})\).
- Proof \((\text{false}_I P) \text{ (false P) :- proof (false)}\).
- Proof \((\text{false}_C P) \text{ (false P) :-}\)
  \(\bot p \land (\text{proof (p A)} \Rightarrow \text{proof (p B) false})\).

Sequent Calculus I

\[\begin{align*}
\Gamma & \rightarrow A \\
\Gamma & \rightarrow A \land B \\
\Gamma & \rightarrow B \land \Gamma \\
\Gamma & \rightarrow A \lor B \\
\Gamma & \rightarrow B \lor \Gamma \\
\Gamma & \rightarrow (\forall x A) \lor \Gamma \\
\Gamma & \rightarrow (\exists x A) \exists \Gamma \\
\Gamma & \rightarrow (\forall x A) \lor \Gamma \\
\Gamma & \rightarrow (\exists x A) \exists \Gamma
\end{align*}\]

Sequent Calculus II

\[\begin{align*}
A, B, \Gamma & \rightarrow C \\
A \land B, \Gamma & \rightarrow C \land \Gamma \\
A, \Gamma & \rightarrow C \\
B, \Gamma & \rightarrow C \lor \Gamma \\
\Gamma & \rightarrow A \lor B, \Gamma \\
\Gamma & \rightarrow \neg A, \Gamma \\
\Gamma & \rightarrow \bot \lor \Gamma
\end{align*}\]

Proviso on \(\forall\text{-}\text{I}: y\) cannot appear free in the lower sequent.
Specifying Sequent Systems

\[ \Gamma \rightarrow A \quad \Gamma \rightarrow B \quad \Gamma \rightarrow A \land B \]

\( \vdash \rightarrow \)

Antecedent Rules

\[ \Gamma \rightarrow A \quad B, \Gamma \rightarrow C \quad \vdash \rightarrow L \]

\( \vdash \rightarrow \)

Classical Logic

\[ \Gamma \rightarrow A, \Delta \quad \Gamma \rightarrow B, \Delta \quad \Gamma \rightarrow A \land B, \Delta \]

\( \vdash \rightarrow \)

Initial Sequents

\[ A, \Gamma \rightarrow A \]

\( \vdash \rightarrow \)

proof (initial A) (Gamma --> A) :- memb A Gamma.

proof (and_r P1 P2) (Gamma --> (A and B)) :-
  proof P1 (Gamma --> A), proof P2 (Gamma --> B).

proof (imp_l P1 P2) (Gamma --> C) :-
  memb (A imp B) Gamma,
  proof P1 (Gamma --> A),
  proof P2 ((B::Gamma) --> C).
Modal Sequents

\[
\begin{align*}
R; \Gamma &\rightarrow A_w, \Delta \\
R; \Gamma &\rightarrow B_w, \Delta \\
R; \Gamma &\rightarrow (A \land B)_w, \Delta
\end{align*}
\]

\(\land \text{-} R\)

kind mform type.
kind world type.
kind wpair type.

type m form \rightarrow world \rightarrow mform.
type r world \rightarrow world \rightarrow wpair.
type mseq (list wpair) \rightarrow

(proof (and_r P1 P2) (mseq R Gamma Delta) :-
memb (m (A and B) W) Delta,
proof P1 (mseq R Gamma (\(m A W\)::Delta)),
proof P2 (mseq R Gamma (\(m B W\)::Delta)).

Specifying the Modal Introduction Rule

\[
\begin{align*}
R, r(w, x); \Gamma &\rightarrow \Delta, A_x \\
R; \Gamma &\rightarrow \Delta, (\square A)_w
\end{align*}
\]

\(\Box \text{-} R\)

type box form \rightarrow form.

proof (box_r P) (mseq R Gamma Delta) :-
memb (mform (box A) W) Delta,
pi \ x (proof (P x)

(proof ((r W x)::R) Gamma ((mform A x)::Delta))).

The Modal Rules

\[
\begin{align*}
[R; r(w, x)] &\rightarrow R; A_x, \Gamma \rightarrow \Delta & \Box \text{-} L
\end{align*}
\]

\(\Box \text{-} L\)

\[
\begin{align*}
R, r(w, x); \Gamma &\rightarrow \Delta, A_x \\
R; \Gamma &\rightarrow \Delta, (\square A)_w
\end{align*}
\]

\(\Box \text{-} \text{R}\)

Proviso on \(\Box \text{R}\): world \(x\) doesn’t occur in the lower sequent.

\[
\begin{align*}
[r(w, v), r(v, u) ; r(w, u)] &\rightarrow r(w, v), r(v, u); pu \rightarrow pu & \Box \text{-} L
\end{align*}
\]

\(\Box \text{-} \text{L}\)

\[
\begin{align*}
r(w, v), r(v, u); (\square ap)_w &\rightarrow pu & \Box \text{-} \text{R}
\end{align*}
\]

\(\Box \text{-} \text{R}\)

\[
\begin{align*}
(r(w, v); (\square ap)_w &\rightarrow (\square ap)_w) &\rightarrow (\square ap \land \square ap)_w & \Box \text{-} \text{R}
\end{align*}
\]

Specifying the Modal Elimination Rule

\[
\begin{align*}
[R; r(w, x)] &\rightarrow R; A_x, \Gamma \rightarrow \Delta \\
R; \Gamma &\rightarrow \Delta, (\square A)_w
\end{align*}
\]

\(\Box \text{-} \text{L}\)

type related (list wpair) \rightarrow wpair \rightarrow o.

related R (r W X) :- memb (r W X) R.
related R (r W W).
related R (r W X) :- related R (r W Z), related R (r Z X).

proof (box_l P) (mseq R Gamma Delta) :-
memb (mform (box A) W) Gamma,
proof P (mseq R (mform A X) Gamma Delta),
related R (r W X).
\[ \beta\eta\text{-Convertibility for the Untyped } \lambda\text{-Calculus} \]

\[
\begin{align*}
\text{kind} & \quad \text{tm} & \quad \text{type} \\
\text{type} & \quad \text{app} & \quad \text{tm} \to \text{tm} \to \text{tm}. \\
\text{type} & \quad \text{abs} & \quad (\text{tm} \to \text{tm}) \to \text{tm}. \\
\end{align*}
\]

\[
\begin{align*}
\lambda f \lambda n \cdot f(n) & \quad \lambda x \cdot xx \\

\end{align*}
\]

\[
\begin{align*}
\text{(abs } f \text{(abs } n \text{(app } f \text{ (app } f \text{ n))))} \\
\text{(abs } x \text{(app } x \text{ x))} \\
\end{align*}
\]

\[ \beta\eta\text{-Redexes} \]

- \( \beta \)-conversion: \( (\lambda x. s)t = s[t/x] \)
- \( \eta \)-conversion: \( \lambda x.(sx) = s \) provided \( x \) does not occur free in \( s \).

\[
\begin{align*}
\text{type} & \quad \text{redex} & \quad \text{tm} \to \text{tm} \to \text{o}. \\
\text{redex} & \quad (\text{app } (\text{abs } S) \ T) & \quad (S \ T). \\
\text{redex} & \quad (\text{abs } x \text{ (app } S \ x)) & \quad S. \\
\end{align*}
\]

\[ \beta\eta\text{-Convertibility and Normalization} \]

\[
\begin{align*}
\text{conv } M \ N & :- \ \text{redi } M \ N. \\
\text{conv } M \ M. \\
\text{conv } M \ N & :- \ \text{conv } M \ M, \ \text{conv } P \ N. \\
\text{norm } M \ N & :- \ \text{redi } M \ P, \ !, \ \text{norm } P \ N. \\
\text{norm } M \ M. \\
\end{align*}
\]

\[ \text{One-Step Reducibility} \]

\[
\begin{align*}
M \to P & \quad \frac{N \to P}{MN \to PN} \\
M \to N & \quad \frac{N \to MP}{\lambda x.M \to \lambda x.N} \\
\text{type} & \quad \text{redi} & \quad \text{tm} \to \text{tm} \to \text{o}. \\
\text{redi} & (\text{app } M \ N) (\text{app } P \ N) :- \ \text{redi } M \ P. \\
\text{redi} & (\text{app } M \ N) (\text{app } M \ P) :- \ \text{redi } N \ P. \\
\text{redi} (\text{abs } M) (\text{abs } N) :- \ \text{pi } x \text{(redi } M \ x) (N \ x)). \\
\end{align*}
\]
Correctness of Representation of λ-terms

- An Encoding of Untyped Terms to Meta-Terms
  - Given $\Phi$: a mapping from the constants of the object language to a fixed set of constants of the meta-language of type $tm$.  
  - Given $\rho$: a mapping from the variables of the object language to the meta-variables of type $tm$.  
  - Example:

$$\langle \lambda f \lambda n. f(fn) \rangle^\Phi \equiv (\text{abs } f \langle \text{abs } n \langle \text{app } f \langle \text{app } f \ n \rangle \rangle \rangle)$$

Theorem (Correctness of Encoding of Untyped Terms)
The encoding $\langle \rangle^\Phi$ is a bijection from the $\alpha$-equivalence classes of untyped terms to the $\beta\eta$-equivalence classes of meta-terms of type $tm$.

---

Evaluation for a Functional Language

- $app : tm \to tm \to tm$
- $abs : (tm \to tm) \to tm$
- $0 : tm$
- $s : tm \to tm$
- $true : tm$
- $false : tm$
- $if : tm \to tm \to tm \to tm$
- $let : (tm \to tm) \to tm \to tm$
- $nil : tm$
- $cons : tm \to tm \to tm$
- $hd : tm \to tm$
- $tl : tm \to tm$
- $empty : tm \to tm$
- $fix : (tm \to tm) \to tm$
- $\emptyset$ nil $\to$ true
- $\emptyset$ cons $M$ $N$ $\to$ false
- $fix$ $M$ $\to$ $M$ ($fix$ $M$)
- $let$ $M$ $N$ $\to$ $MN$
- $\emptyset$ cons $M$ $N$ $\to$ $M$
- $\emptyset$ cons $M$ $N$ $\to$ $N$
- $\emptyset$ hd $\to$ $M$
- $\emptyset$ tl $\to$ $N$
- $\emptyset$ empty $\to$ true
- $\emptyset$ empty $\to$ false
- $\emptyset$ empty $\to$ $M$
- $\emptyset$ empty $\to$ $N$

---

Correctness of $\beta\eta$-Convertibility Specification

Theorem Let $M$ and $N$ be untyped terms. Then $M =_{\beta\eta} N$ if and only if

$$\text{conv} \langle M \rangle^\Phi \langle N \rangle^\Phi$$

is provable.

---

A Specification of Evaluation

type eval :: tm -> tm -> o.

eval (abs $M$ $N$) $P$ :- eval $N$ $N'$, eval ($M$ $N'$) $P$.
eval (if $C$ $M$ $N$) $M'$ :- eval $C$ true, eval $M$ $M'$.
eval (if $C$ $M$ $N$) $N'$ :- eval $C$ false, eval $N$ $N'$.
eval ($hd$ $L$) $M'$ :- eval $L$ ($cons$ $M$ $N$), eval $M$ $M'$.
eval ($tl$ $L$) $N'$ :- eval $L$ ($cons$ $M$ $N$), eval $N$ $N'$.
eval (empty $L$) true :- eval $L$ nil.
eval (empty $L$) false :- eval $L$ ($cons$ $M$ $N$).
eval (fix $M$) $N$ :- eval ($M$ ($fix$ $M$)) $N$.
eval (let $M$ $N$) $P$ :- eval $N$ $N'$, eval ($M$ $N'$) $P$.  

Some Related Languages

- The Logical Framework (LF) [Harper, Honsell, & Plotkin, JACM 93] is a type theory developed to capture the generalities across a wide variety of object logics. A specification of a logic in LF can be “compiled” rather directly into a set of \( \lambda \)Prolog clauses.
- The Forum logic programming language [Miller, TCS 96] implements an extension of higher-order hereditary Harrop formulas (hohh) to linear logic.
- Isabelle [Paulson 94] is a “generic” tactic theorem prover implemented in ML. It contains a specification language which is a subset of hohh. The two are very close in specification strength.

Reversibility of Rules

\[
\begin{align*}
A, B, \Gamma & \rightarrow \Delta & \Delta & \rightarrow A, \Delta & \Gamma & \rightarrow B, \Delta \\
A \land B, \Gamma & \rightarrow \Delta & \Gamma & \rightarrow A \land B, \Delta \\
\end{align*}
\]

- \( \land \)-L: There is a proof of one of the formulas in \( \Delta \) from \( A \land B \) and \( \Gamma \) if and only if there is a proof of one of the formulas in \( \Delta \) from \( A \) and \( B \) and \( \Gamma \).
- \( \land \)-R: There is a proof of \( A \land B \) or of one of the formulas in \( \Delta \) from \( \Gamma \) if and only if there is a proof of \( A \) or one of the formulas in \( \Delta \) from \( \Gamma \) and there is a proof of \( B \) or one of the formulas in \( \Delta \) from \( \Gamma \).

Non-Reversibility of Rules

The only two rules in the classical sequent calculus presented that are not reversible are:

\[
\begin{align*}
[t/x]A, \Gamma & \rightarrow \Delta & \Delta & \rightarrow [t/x]A, \Delta \\
\forall x A, \Gamma & \rightarrow \Delta & \Gamma & \rightarrow \exists x A, \Delta \\
\end{align*}
\]

For example, there may be a proof of one of the formulas in \( \Delta \) from \( \forall x A \) and \( \Gamma \), but no term \( t \) such that there is a proof of one of the formulas in \( \Delta \) from \( [t/x]A \) and \( \Gamma \). It may be the case that \( \forall x A \) must be instantiated with more than one term.
A Specification that Removes Formulas

\[
\frac{A, B, \Delta}{A \land B, \Delta} \rightarrow \Delta
\]

<table>
<thead>
<tr>
<th>Step 1 of 3: the proof1 procedure</th>
</tr>
</thead>
</table>
| 1. Apply all rules except \( \forall \land \) and \( \exists \land \) until nothing more can be done. The result is a set of sequents with atomic and universally quantified formulas on the left, and atomic and existentially quantified formulas on the right.

\[
\text{proof1 (initial A) (\Gamma \rightarrow \Delta)} :\text{memb A \Gamma, memb A \Delta.}
\]

\[
\text{proof1 (and_r P1 P2) (\Gamma \rightarrow \Delta)} :\text{memb_and_rest (A and B) \Gamma Delta Delta1, proof1 P1 (\Gamma \rightarrow (A : Delta1)), proof1 P2 (\Gamma \rightarrow (B : Delta1)).}
\]

\[
\text{proof1 (imp_l P1 P2) (\Gamma \rightarrow \Delta)} :\text{memb_and_rest (A imp B) Gamma Gamma1, proof1 P1 (Gamma1 \rightarrow (A : Delta1)), proof1 P2 ((B : Gamma1) \rightarrow Delta).}
\]

<table>
<thead>
<tr>
<th>Step 2 of 3: the proof2 procedure</th>
</tr>
</thead>
</table>
| 2. Apply all rules including versions of the rules for \( \forall \land \) and \( \exists \land \) that remove the quantified formula after applying the rule, and try to complete the proof. Stop if a proof is successfully completed.

\[
\text{proof2 (forall_1 P) (Gamma \rightarrow Delta)} :\text{memb_and_rest (forall A) Gamma Gamma1, proof2 P ((A T) : Gamma1 \rightarrow Delta).}
\]

\[
\text{proof2 (exists_r P) (Gamma \rightarrow Delta)} :\text{memb_and_rest (exists A) Delta Delta1, proof2 P (Gamma1 \rightarrow ((A T) : Delta1)).}
\]

(plus duplicates of each of the proof1 clauses)

<table>
<thead>
<tr>
<th>Step 3 of 3: the nprove procedure</th>
</tr>
</thead>
</table>
| 3. Add an additional copy of each quantified formula to the sequents obtained from step 1, and repeat steps 2 and 3.

\[
\text{nprove N P Seq : - amplify N Seq ASeq, proof2 P ASeq.}
\]

\[
\text{nprove N P Seq : - M is (N + 1), nprove M P Seq.}
\]

\[
\text{amplify 1 Seq Seq.}
\]

\[
\text{amplify N (Gamma1 \rightarrow Delta1) (Gamma2 \rightarrow Delta2) : - N > 1, amplifyforall N Gamma1 Gamma2, amplify_exists N Delta1 Delta2.}
\]
Putting it all Together

The top-level predicate is proof1. Add one more clause for it at the end.

proof1 P Seq :- nprove 1 P Seq.
nprove N P Seq :- amplify N Seq ASeq, proof2 P ASeq.
nprove N P Seq :- M is (N + 1), nprove M P Seq.

Completeness follows from the fact proved in [Andrews, JACM 81] that duplication of outermost quantifiers is all that is necessary to obtain a complete procedure, and the fact that step 2 will always terminate.

Examples

The first proof completes at amplification 1. The second needs amplification 2.

\[
\begin{align*}
& p(a) \rightarrow p(a) \quad 3R \\
& p(a) \rightarrow \exists x p(x) \quad 3R \\
& p(b) \rightarrow p(b) \quad 3R \\
& p(a) \lor p(b) \rightarrow \exists x p(x) \quad \lor L \\
& \rightarrow p(a) \lor p(b) \supset \exists x p(x) \quad \lor R
\end{align*}
\]

\[
\begin{align*}
& \rightarrow p(t) \supset p(z), p(z) \supset p(w) \quad \lor R \\
& \rightarrow p(t) \supset p(z), \forall y (p(x) \supset p(y)) \quad \lor R \\
& \rightarrow \forall y (p(t) \supset p(y)), \exists x \forall y (p(x) \supset p(y)) \quad 3R \\
& \rightarrow \exists x \forall y (p(x) \supset p(y)), \exists x \forall y (p(x) \supset p(y)) \quad 3R
\end{align*}
\]

Part IV: Implementing Interactive Tactic Theorem Provers

- Inference Rules as Tactics
- A Goal Reduction Tactical
- Some Common Tactics
- Tactics and Tacticals for Interaction
- An Example Execution
Tactic Theorem Provers

- In general, more flexibility in control of search is needed than can be provided by depth-first search with backtracking.
- Tactics and tacticals have proven to be a powerful mechanism for implementing theorem provers. Example tactic provers (all ML implementations) include:
  - LCF [Gordon, Milner, & Wadsworth]
  - HOL [Gordon]
  - Isabelle [Paulson]
  - Nuprl [Constable et. al.]
  - Coq [Huet et. al.]
- Tactics and tacticals can be implemented directly and naturally in λProlog. They implement an interpreter for goal-directed theorem proving in the logic programming setting.

Inference Rules As Tactics

\[
\begin{array}{c}
A \\
\hline
A \land B
\end{array}
\]

\[
\text{proof (and\_i P1 P2) (A and B)} : - \\
\text{proof P1 A, proof P2 B.}
\]

\[
\text{and\_i\_tac (proof (and\_i P1 P2) (A and B))} \\
\quad ((\text{proof P1 A}) ^^ (\text{proof P2 B})).
\]

Tactics with Assumption Lists

\[
\text{and\_i\_tac (proof (and\_i P1 P2) (A and B))} \\
\quad ((\text{proof P1 A}) ^^ (\text{proof P2 B})).
\]

Goal Constructors

\[
\begin{array}{c}
tt \quad \text{goal.} \\
^^ \quad \text{goal} \rightarrow \text{goal} \rightarrow \text{goal.} \\
vv \quad \text{goal} \rightarrow \text{goal} \rightarrow \text{goal.} \\
all \quad (A \rightarrow \text{goal}) \rightarrow \text{goal.} \\
some \quad (A \rightarrow \text{goal}) \rightarrow \text{goal.} \\
eq \quad o \rightarrow \text{goal} \rightarrow \text{goal.} \\
\end{array}
\]

\[
\begin{array}{c}
infixl \quad 3. \\
infixl \quad 3. \\
infixr \quad 3. \\
\end{array}
\]
A Goal Reduction Tactical

\[
\text{type maptac } (\text{goal} \to \text{goal} \to \text{o}) \to \text{goal} \to \text{goal} \to \text{o}.
\]

\[
\text{maptac Tac tt tt}.
\]

\[
\text{maptac Tac (InGoal1 :: InGoal2) (OutGoal1 :: OutGoal2) :-}
\]

\[
\text{maptac Tac InGoal1 OutGoal1, maptac Tac InGoal2 OutGoal2}.
\]

\[
\text{maptac Tac (all InGoal) (all OutGoal) :-}
\]

\[
\text{pi x (maptac Tac (InGoal x) (OutGoal x)).}
\]

\[
\text{maptac Tac (InGoal1 v InGoal2) OutGoal :-}
\]

\[
\text{maptac Tac InGoal1 OutGoal1; maptac Tac InGoal2 OutGoal2}.
\]

\[
\text{maptac Tac (some InGoal) OutGoal :-}
\]

\[
\text{sigma T (maptac Tac (InGoal T) OutGoal)}.
\]

\[
\text{maptac Tac (D =>> InGoal) (D =>> OutGoal) :-}
\]

\[
D => (\text{maptac Tac InGoal OutGoal}).
\]

\[
\text{maptac Tac InGoal OutGoal :- Tac InGoal OutGoal}.\]

Interactive Theorem Proving

\[
\text{type query } (\text{goal} \to \text{o}) \to \text{goal} \to \text{goal} \to \text{o}.
\]

\[
\text{type inter } (\text{goal} \to \text{o}) \to \text{goal} \to \text{goal} \to \text{o}.
\]

\[
\text{type with_tacs string } \to (\text{goal} \to \text{goal} \to \text{o})
\]

\[
\to \text{goal} \to \text{goal} \to \text{o}.
\]

\[
\text{query PrintPred InGoal OutGoal :-}
\]

\[
\text{PrintPred InGoal, print "Enter tactic: ", readtac Tac, (Tac = backup, !, fail; Tac InGoal OutGoal).}
\]

\[
\text{inter PrintPred InGoal OutGoal :-}
\]

\[
\text{repeat (query PrintPred) InGoal OutGoal}.
\]

\[
\text{with_tacs M Tac InGoal OutGoal :-}
\]

\[
M => (\text{Tac InGoal OutGoal}).
\]

Interactive Tactics for Natural Deduction

- Allowing the User to Specify Substitution Terms

\[
\text{exists_i_tac } \text{(proof}\ (\text{exists}_i P) \text{ (exists} A))
\]

\[
\text{(proof} P \text{ (A T))}.
\]

\[
\text{exists_i_sub } \text{(proof}\ (\text{exists}_i P) \text{ (exists} A))
\]

\[
\text{(proof} P \text{ (A T)) :-}
\]

\[
\text{print "Enter substitution term:"}, \text{ read} T.
\]

- Adding Lemmas

\[
\text{modus_ponens } \text{(proof} P A)
\]

\[
\text{((proof} Q B) \text{ --}
\]

\[
\text{((assume} Q B) \text{ =>} (\text{proof} P A))) :-
\]

\[
\text{print "Enter lemma:"}, \text{ read} B.
\]

\[
\text{close_tac } \text{(proof} P A) \text{ tt :- assume} P A.
\]
Natural Deduction Inference Rule Tactics

\[
\begin{align*}
(A) \\
\frac{B}{A \supset B} \quad \text{-I}
\end{align*}
\]

proof (imp_i P) (A imp B) :-
   pi pA\((\text{proof } pA A) \Rightarrow (\text{proof } (P pA) B))

imp_i_tac (proof (imp_i P) (A imp B))
   (all pA\((\text{assume } pA A) \Rightarrow (\text{proof } (P pA) B))

imp_i_tac (deduct Gamma (proof (imp_i P) (A imp B)))
   (all pA\((\text{deduct } ((\text{proof } pA A) : Gamma)
   (\text{proof } (P pA) B))

All introduction rules can be translated to tactics similarly.

Elimination Rules as Tactics

\[
\begin{align*}
A \wedge B \\
\frac{A}{A} \wedge E \\
\frac{B}{B} \wedge E
\end{align*}
\]

proof (and_e P) A :-
   proof P (A and B); proof P (B and A).

and_e_tac (deduct Gamma (proof P A))
   (\text{deduct Gamma (proof (and_e1 P) (A and B))) vv}
   (\text{deduct Gamma (proof (and_e1 P) (B and A)))}.

and_e_tac (deduct Gamma (proof P A))
   (\text{deduct Gamma (proof (and_e1 P) (A and B))) vv}
   (\text{deduct Gamma (proof (and_e1 P) (B and A)))} :-
   \text{print "Enter second conjunct:"}, \text{read } B.

Forward Proof Using Elimination Rules

and_e_tac N (deduct Gamma (proof PC C))
   (deduct ((\text{proof (and_e1 P) A})::
   (\text{proof (and_e2 P) B)}:Gamma)
   (\text{proof PC C}) :-
   nth_item N (proof P (A and B)) Gamma.

All elimination rules can be implemented as tactics similarly.

An Example Query

?- interactive
\((\text{proof } P (((q a) or (q b)) imp (\text{exists } x\backslash(q x))))\)
OutGoal.

Assumptions:
Conclusion:
(q a or q b) imp (\text{exists } x\backslash(q x))
Enter tactic: ?- imp_i_tac.

Assumptions:
1 q a or q b
Conclusion:
\text{exists } x\backslash(q x)
Enter tactic: ?- exists_i_tac.
Generic Theorem Proving

- Logics in the Isabelle theorem prover [Paulson 94] are specified in a language which is a subset of hohh, while control, including tactics and tacticals, is implemented in ML.
- Here, tactics and tacticals are specified in hohh. The λProlog interpreter associates control primitives (search operations) to the logical connectives of hohh.
- Much work has gone into making Isabelle efficient as well as providing extensive environments for several particular object logics. These environments include efficient specialized tactics as well as large libraries of theorems.
- Such an effort has not been made for λProlog, but could be. Experience with Isabelle demonstrates the effectiveness of generic theorem proving.
Part V: An Implementation of Higher-Order Term Rewriting

- Higher-Order Rewrite Rules
- Some Example Rewrite Systems
- Expressing a Rewrite System as a Set of Tactics
- Tactics and Tacticals for Rewriting

Higher-Order Rewrite Rules

A rewrite rule is a pair \( l \rightarrow r \) such that \( l \) and \( r \) are simply-typed \( \lambda \)-terms of the same primitive type, \( l \) is a term in \( \text{L}_\lambda \), and all free variables in \( r \) also occur in \( l \).

Example 1: \( \beta \eta \)-conversion for \( \lambda \)-terms

- \( \beta \)-conversion: \( (\lambda x.s)t = s[t/x] \)
- \( \eta \)-conversion: \( \lambda x.(sx) = s \) provided \( x \) does not occur free in \( s \).

Three Parts of a Rewriting Procedure

- Rewrite Rules
  
  \[ \text{red}_1 \ M \ N \quad : \quad \text{red}_1 \ M \ N. \]
  
  \[ \text{red}_1 \ (\text{app} \ M \ N) \ (\text{app} \ P \ N) \quad : \quad \text{red}_1 \ M \ P. \]
  
  \[ \text{red}_1 \ (\text{app} \ M \ N) \ (\text{app} \ M \ P) \quad : \quad \text{red}_1 \ N \ P. \]
  
  \[ \text{red}_1 \ (\text{abs} \ M) \ (\text{abs} \ N) \quad : \quad \text{pi} \ x/(\text{red}_1 \ (M \ x) \ (N \ x)). \]

- Congruence and One-Step Rewriting

- \( \beta \)-conversion and \( \eta \)-conversion: 
  
  \[ \text{red}_1 \ M \ N : - \ \text{red}_1 \ M \ N. \]
  
  \[ \text{red}_1 \ (\text{app} \ M \ N) \ (\text{app} \ P \ N) : - \ \text{red}_1 \ M \ P. \]
  
  \[ \text{red}_1 \ (\text{app} \ M \ N) \ (\text{app} \ M \ P) : - \ \text{red}_1 \ N \ P. \]
  
  \[ \text{red}_1 \ (\text{abs} \ M) \ (\text{abs} \ N) : - \ \text{pi} \ x/(\text{red}_1 \ (M \ x) \ (N \ x)). \]

- Multiple Step Reduction

- \( \beta \)-conversion and \( \eta \)-conversion:

- \( \beta \)-conversion and \( \eta \)-conversion:

- \( \beta \)-conversion and \( \eta \)-conversion:
Rewriting in a Tactic Theorem Prover

- The previous example implements the leftmost-outermost rewrite strategy. Using a different order on the red1 clauses can give other rewrite strategies such as bottom-up.
- In a tactic theorem prover, rewrite rules and congruence rules can be implemented as basic tactics. More complex tactics can be implemented for various strategies.

Example 2: Evaluation as Rewriting

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>app</code></td>
<td><code>tm -&gt; tm -&gt; tm</code></td>
</tr>
<tr>
<td><code>abs</code></td>
<td><code>(tm -&gt; tm) -&gt; tm</code></td>
</tr>
<tr>
<td><code>0</code></td>
<td><code>tm</code></td>
</tr>
<tr>
<td><code>s</code></td>
<td><code>tm -&gt; tm</code></td>
</tr>
<tr>
<td><code>true</code></td>
<td><code>tm</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>tm</code></td>
</tr>
<tr>
<td><code>nil</code></td>
<td><code>tm</code></td>
</tr>
<tr>
<td><code>cons</code></td>
<td><code>tm -&gt; tm -&gt; tm</code></td>
</tr>
<tr>
<td><code>fix</code></td>
<td><code>(tm -&gt; tm) -&gt; tm</code></td>
</tr>
<tr>
<td><code>hd</code></td>
<td><code>tm -&gt; tm</code></td>
</tr>
<tr>
<td><code>tl</code></td>
<td><code>tm -&gt; tm</code></td>
</tr>
<tr>
<td><code>empty</code></td>
<td><code>tm</code></td>
</tr>
<tr>
<td><code>let</code></td>
<td><code>(tm -&gt; tm) -&gt; tm -&gt; tm</code></td>
</tr>
</tbody>
</table>

Rewrite and Congruence Rules as Tactics

<table>
<thead>
<tr>
<th>Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type</code></td>
<td><code>A -&gt; A -&gt; goal</code></td>
</tr>
<tr>
<td><code>infix</code></td>
<td><code>7</code></td>
</tr>
<tr>
<td><code>type prim</code></td>
<td><code>goal -&gt; goal</code></td>
</tr>
<tr>
<td><code>type rew</code></td>
<td><code>goal -&gt; goal -&gt; o</code></td>
</tr>
<tr>
<td><code>type cong</code></td>
<td><code>goal -&gt; goal -&gt; o</code></td>
</tr>
<tr>
<td><code>type cong_const</code></td>
<td><code>goal -&gt; goal -&gt; o</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rew</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rew (prim ((app (abs S) T) == (S T)))</code></td>
<td><code>tt</code></td>
</tr>
<tr>
<td><code>rew (prim ((abs x\(app S x)) == S))</code></td>
<td><code>tt</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Congruence</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cong (prim ((app M N) == (app P Q)))</code></td>
<td><code>((prim (M == P)) ^^ (prim (N == Q)))</code></td>
</tr>
<tr>
<td><code>cong (prim ((abs M) == (abs N)))</code></td>
<td><code>((all x\((cong const (prim (x == x))) tt) ==&gt;) (prim ((M x) == (N x))))</code></td>
</tr>
<tr>
<td><code>cong (prim (f == f))</code></td>
<td><code>tt</code></td>
</tr>
</tbody>
</table>

Congruence Tactics for Evaluation

<table>
<thead>
<tr>
<th>Congruence</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cong_const (prim (tru == tru))</code></td>
<td><code>tt</code></td>
</tr>
<tr>
<td><code>cong_const (prim (fals == fals))</code></td>
<td><code>tt</code></td>
</tr>
<tr>
<td><code>cong_const (prim (x == x))</code></td>
<td><code>tt</code></td>
</tr>
<tr>
<td><code>cong (prim ((m M N) == (m P Q)))</code></td>
<td><code>((prim (M == P)) ^^ (prim (N == Q)))</code></td>
</tr>
<tr>
<td><code>cong (prim ((empty M) == (empty N)))</code></td>
<td><code>((prim (M == N)))</code></td>
</tr>
<tr>
<td><code>cong (prim ((if C M N) == (if D P Q)))</code></td>
<td><code>((prim (C == D)) ^^ (prim (M == P)) ^^ (prim (N == Q)))</code></td>
</tr>
<tr>
<td><code>cong (prim ((fix M) == (fix N)))</code></td>
<td><code>((all x\((cong const (prim (x == x))) tt) ==&gt;) (prim ((M x) == (N x))))</code></td>
</tr>
<tr>
<td><code>cong (prim ((let M N) == (let P Q)))</code></td>
<td><code>((all x\((cong const (prim (x == x))) tt) ==&gt;) (prim ((M x) == (P x))))</code></td>
</tr>
<tr>
<td><code>cong (prim ((m M N) == (m P Q)))</code></td>
<td><code>((prim (M == P)) ^^ (prim (N == Q)))</code></td>
</tr>
</tbody>
</table>
Evaluation Rewrite Rules

\[
\begin{align*}
\text{app (abs } M \text{)} N & \rightarrow MN \\
\text{if true } M N & \rightarrow M \\
\text{if false } M N & \rightarrow N \\
\text{hd (cons } M N \text{)} & \rightarrow M \\
\text{tl (cons } M N \text{)} & \rightarrow N \\
\text{empty nil} & \rightarrow \text{true} \\
\text{empty (cons } M N \text{)} & \rightarrow \text{false} \\
\text{fix } M & \rightarrow M (\text{fix } M) \\
\text{let } M N & \rightarrow MN
\end{align*}
\]

Example 3: Negation Normal Forms

- Congruence Rules

\[
\begin{align*}
\text{cong (prim ((A and B) == (C and D)))} \\
& ((\text{prim (A == C)}) \wedge (\text{prim (B == D)})) \\
\text{cong (prim ((forall A) == (forall B)))} \\
& (\forall x ((\text{cong const (prim (x == x)) tt}) \implies (\text{prim ((A x) == (B x))})))
\end{align*}
\]

- Rewrite Rules

\[
\begin{align*}
\text{rew (prim ((neg (A and B)) == (neg A) or (neg B)))) tt.} \\
\text{rew (prim ((neg (forall A)) == (exists x (neg (A x)))) tt.)}
\end{align*}
\]

Tactics Implementing Evaluation Rewrites

\[
\begin{align*}
\text{rew (prim ((app (abs M) N) == (M N))) tt.} \\
\text{rew (prim ((abs X (app M X)) == M)) tt.} \\
\text{rew (prim ((hd (cons M N)) == M)) tt.} \\
\text{rew (prim ((tl (cons M N)) == N)) tt.} \\
\text{rew (prim ((empty nil) == tru)) tt.} \\
\text{rew (prim ((empty (cons M N)) == fals)) tt.} \\
\text{rew (prim ((if tru M N) == M)) tt.} \\
\text{rew (prim ((if fals M N) == N)) tt.} \\
\text{rew (prim ((fix M) == (M (fix M)))) tt.} \\
\text{rew (prim ((let M N) == (M N))) tt.}
\end{align*}
\]

A Modified maptac

\[
\begin{align*}
\text{type maptacC (goal -> goal -> o) -> goal -> goal -> o.} \\
\text{maptacC Tac tt tt.} \\
\text{maptacC Tac (InGoal1 ^^ InGoal2) OutGoal :=} \\
& \text{Tac (InGoal1 ^^ InGoal2) OutGoal.} \\
\text{maptacC Tac (all InGoal) (all OutGoal) :=} \\
& \text{pi x (maptacC Tac (InGoal x) (OutGoal x)).} \\
\text{maptacC Tac (InGoal1 vv InGoal2) OutGoal :=} \\
& \text{maptacC Tac InGoal1 OutGoal;} \\
& \text{maptacC Tac InGoal2 OutGoal.} \\
\text{maptacC Tac (some InGoal) OutGoal :=} \\
& \text{sigma T (maptacC Tac (InGoal T) OutGoal).} \\
\text{maptacC Tac (D ==> InGoal) (D ==> OutGoal) :=} \\
& \text{D -> (maptacC Tac InGoal OutGoal).} \\
\text{maptacC Tac (prim InGoal) OutGoal :=} \\
& \text{Tac (prim InGoal) OutGoal.}
\end{align*}
\]
Rewrite Tactics and Tacticals I

refl (prim (M == N)) tt.
sym (prim (M == N)) (prim (N == M)).
trans (prim (M == N)) ((prim (M == P)) \^\^ (prim (P == N))).
left Tac (prim InG) OutG :- Tac (prim InG) OutG.
left Tac (InG \^\^ G) (OutG \^\^ G) :-
  maptacC (left Tac) InG OutG.
left rew Tac InG OutG :-
  thenC trans (left Tac) InG OutG.

Rewrite Tactics and Tacticals II

right Tac (prim InG) OutG :- Tac (prim InG) OutG.
right Tac (G \^\^ InG) (G \^\^ OutG) :-
  maptacC (right Tac) InG OutG.
right rew Tac InG OutG :-
  thenC trans (right Tac) InG OutG.
first Tac (prim InG) OutG :- Tac (prim InG) OutG.
first Tac (InG \^\^ G) (OutG \^\^ G) :-
  maptacC (first Tac) InG OutG, !.
first Tac (G \^\^ InG) (G \^\^ OutG) :-
  maptacC (first Tac) InG OutG, !.

Bottom-Up Rewriting

bu Cong Rew InG OutG :-
  then (bu_sub Cong Rew)
    (orelse (then (left rew Rew) (bu Cong Rew))
      refl) InG OutG.
bu_sub Cong Rew InG OutG :-
  try (left rew (then Cong (bu Cong Rew))) InG OutG.
**Leftmost-Outermost Rewriting**

\[ \text{lo Cong Rew InG OutG} :- \]
\[ \text{then (repeat (left rew (lo rew Cong Rew)))} \]
\[ \text{refl InG OutG}. \]

\[ \text{lo rew Cong Rew InG OutG} :- \]
\[ \text{orelse Rew (then (then Cong}} \]
\[ \text{(first (lo rew Cong Rew)))} \]
\[ \text{refl) InG OutG}. \]

**An Example**

Let \( APP \) be the following term representing the program for appending two lists in our functional language.

\[
(fix \lambda F. (\text{abs } \lambda l_1. (\text{abs } \lambda l_2. 
\text{(if (empty } l_1 \text{) } l_2 (\text{cons (hd } l_1) (\text{app (app } F (\text{tl } l_1)) l_2))))))
\]

- The **lo** strategy reduces
  \[
  (\text{app (app } APP (\text{cons } 0 \text{ nil})) (\text{cons } (s \ 0) \text{ nil}))
  \]
  to (\text{cons } 0 (\text{cons } (s \ 0) \text{ nil})).

  The **lo** strategy corresponds to lazy evaluation of this language.

- The **bu** strategy loops, repeatedly applying the rewrite rule for \( fix \) and expanding the definition of the function.

**Other Rewrite Strategies**

- The **bu** and **lo** tactics implement common complete strategies for terminating rewrite systems. They illustrate the use of tactics and tacticals for implementing rewrite procedures.

- The real power of the tactic setting is that it provides a set of high-level primitives with which to write specialized strategies. Examples include:
  - Call-by-value vs. call-by-name evaluation. Strong vs. weak evaluation (reducing under a \( \lambda \)-abstraction or not).
    [Hannan, ELP’91]
  - Type-driven rewriting using \( \eta \)-expansion. [Pfenning,91]
  - Layered rewriting where the application of a subset of the possible rewrite rules are applied, and rewriting is interleaved with other reasoning.
  - Tactics specialized to particular applications or domains.

**Part VI: Encoding the Logical Framework in \( \lambda \)Prolog**

- Syntax of the Logical Framework (LF)
  [Harper, Honsell, & Plotkin, JACM 93]

- An Example LF Signature

- Translating LF Signatures to Logic Programming Specifications
**LF Syntax**

Syntax for LF Kinds, Types, Objects

\[\begin{align*}
K & := \text{Type} | \forall x : A.K \\
A & := x | \forall x : A.B | \lambda x : A.B | A.M \\
M & := x | \lambda x : A.M | MN
\end{align*}\]

- **Dependent Types**: Types can depend on terms. In particular, in \(\forall x : A.B\), the variable \(x\) can occur in the type \(B\). \(A \to B\) denotes \(\forall x : A.B\) when \(x\) does not occur in \(B\).
- **Kinds** can depend on terms also.
- **Terms** are similar to the \(\lambda\)-terms of hohh except that in \(\lambda x : A.M\), \(A\) can be a dependent type.

**LF Contexts and Assertions**

Syntax for Contexts (Signatures)

\[\Gamma := \emptyset | \Gamma, x : K | \Gamma, x : A\]

LF Assertions

\[\begin{align*}
\Gamma \vdash K \text{ kind} & \quad (K \text{ is a kind in } \Gamma) \\
\Gamma \vdash A : K & \quad (A \text{ has kind } K \text{ in } \Gamma) \\
\Gamma \vdash M : A & \quad (M \text{ has type } A \text{ in } \Gamma)
\end{align*}\]

Valid Contexts

The empty context is valid and \(\Gamma, x : P\) is a valid context if \(\Gamma\) is a valid context and either \(\Gamma \vdash P \text{ kind}\) or \(\Gamma \vdash P : \text{Type}\).

**An LF Signature for Natural Deduction**

\[\begin{align*}
& \frac{A \land B}{A \land B} \land \text{-I} \\
& \frac{(A)}{B} \land \text{-R} \\
& \frac{[y/x]A}{\forall x A} \forall \text{-I}
\end{align*}\]

**form : Type**

**i : Type**

\[\begin{align*}
\land : \text{form} \to \text{form} \to \text{form} & \\
\lor : (i \to \text{form}) \to \text{form} & \\
\top & \quad \text{true : form} \to \text{Type}
\end{align*}\]

\[\begin{align*}
\land \text{-I} : \Pi A : \text{form.}\Pi B : \text{true.}(A \to \text{true}(B) \to \text{true}(A \land B)) & \\
\lor \text{-I} : \Pi A : \text{form.}\Pi B : \text{true.}(A \land B) & \\
\forall \text{-I} : \Pi A : i \to \text{form.}(\Pi x : i.\text{true}(Ax)) & \to \text{true}(\forall A)
\end{align*}\]
Translating Kind and Type Declarations

- Introducing New Base Types
  \[ \text{form} : \text{Type} \]
  \[ \text{i} : \text{Type} \]
  kind form type.
  kind i type.

- Introducing the Syntax of the Object Logic
  \[ \land : \text{form} \to \text{form} \to \text{form} \]
  type and form -> form -> form.

- Dependent Type Constants as Predicates
  \[ \text{true} : \text{form} \to \text{Type} \]
  type proof form -> o.

Inference Rules as Clauses I

\[ \land I : \Pi A : \text{form}.\Pi B : \text{form}.(\text{true}(A) \to \text{true}(B)) \to \text{true}(A \land B) \]

proof (A and B) :- proof A, proof B.

An LF term inhabiting the type \text{true}(A \land B) will be a proof of the formula \( A \land B \). If we use the above signature item in constructing such a term, this term will have the form:

\[(\land I \ A \ B \ P_1 \ P_2)\]

We can incorporate proof objects of this form into \( \lambda \text{Prolog} \) specifications.

\[ \text{type proof nprf} \to \text{form} \to o. \]
\[ \text{type} \ \text{and_i form} \to \text{form} \to \text{nprf} \to \text{nprf} \to o. \]

proof (and_i A B P1 P2) (A and B) :-
  proof P1 A, proof P2 B.

Inference Rules as Clauses II

\[ \supset I : \Pi A : \text{form}.\Pi B : \text{form}.(\text{true}(A) \to \text{true}(B)) \to \text{true}(A \supset B) \]

\[ \text{type imp_i form} \to \text{form} \to (\text{nprf} \to \text{nprf}) \to \text{nprf}. \]

\[ \text{proof} \ (\text{imp_i A B P}) \ (\text{A imp B}) :- \]
  \[ \text{pi pA} ((\text{proof pA A}) \Rightarrow (\text{proof (P pA) B})). \]

\[ \forall I : \Pi A : i \to \text{form}.(\Pi x : i.\text{true}(Ax)) \to \text{true}(\forall A) \]

\[ \text{type} \ \forall\text{-i} \ (i \to \text{form}) \to (i \to \text{nprf}) \to \text{nprf}. \]

\[ \text{proof} \ (\forall\text{-i} A P) \ (\forall A) :- \]
  \[ \text{pi y} ((\text{proof (P y) (A y)})). \]

Summary

- An LF signature item is translated to a type declaration and a clause. The type declaration is a “flat” version of the LF type, while the clause replaces dependent types with predicates.
- This correspondence is formalized in [Felty\&Miller, CADE'90].
- The translation is fairly direct, so the two are very close in specification strength.
- LF serves as a logical foundation for the logic programming language Elf [Pfenning LICS'89].