# A Tutorial on Lambda Prolog and its Applications to Theorem Proving

**Amy Felty**
*Bell Labs, Lucent Technologies*

**September 1997**

## Outline

1. **The λProlog 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 λProlog**

## Tutorial References

- λProlog [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/lProlog/`

- Theorem Proving Applications:
  - [Felty, JAR'93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
  - [Felty, ELP'91] Amy Felty. A logic programming approach to implementing higher-order term rewriting. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the January 1991 Workshop on Extensions to Logic Programming*, pages 135–161, 1992.
  - [Felty&Miller, CADE'90] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In *Tenth International Conference on Automated Deduction*, pages 221–235, July 1990.

## 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/lfs.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

# Sublanguages of λProlog

$$\begin{tikzcd}
\textbf{fohh} \arrow[r] & \mathbf{L}_\lambda \arrow[r] & \textbf{hohh} \\
\textbf{fohc} \arrow[u] \arrow[rr] & & \textbf{hohc} \arrow[u]
\end{tikzcd}$$

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

# 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

# Kinds and Kind Declarations

- Primitive Types
  - System Types

```
kind    o       type.
kind    int     type.
kind    real    type.
kind    string  type.
```

*(for propositions)*

  - User-Defined Types, *e.g.*,

```
kind    node    type.
```

- Type Constructors

```
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.*,
$\tau_1 \to \tau_2 \to \tau_3$ denotes $\tau_1 \to (\tau_2 \to \tau_3)$.

# 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.
```

Term Syntax

$$t ::= \ \texttt{c} \ \mid \ \texttt{X} \ \mid \ \texttt{x}\backslash t \ \mid \ \texttt{X}\backslash t \ \mid \ (t_1 \ \ t_2)$$

Curried Notation is Used

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

# Clauses and Goals

```
type    true         o.                    infixr  ,  2.
type    ,            o -> o -> o.          infixr  ;  3.
type    :-, =>, ;    o -> o -> o.          infixr  :- 1.
type    pi, sigma    (A -> o) -> o.        infixr  => 4.

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

append nil K K.
append (X :: L) K (X :: M) :- append L K M.

?- append (1 :: nil) (2 :: nil) L.
L == (1 :: 2 :: nil).
```

# First-Order Horn Clauses

- Atomic Formulas:
  $A$ of type `o` whose top-level symbol is not a logical constant.
- Goal Formulas:

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x \ G$$

- Definite Clauses:

$$D := A \mid G \supset A \mid \forall_\tau x \ D$$

## Explicit Quantification

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

```
type    adj, path   node -> node -> o.


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

path X Y :- sigma z\(adj X z, path z Y).

pi x\(pi y\(path x y :-
            sigma z\(adj x z, path z y))).


?- path a X.

?- sigma x\(path a x).
```

## Type and Clausal Order

- Order of a type expression:

$$\begin{aligned} ord(\tau) &= 0 \quad \text{(for atomic type or type variable } \tau\text{)} \\ ord(\tau_1 \to \tau_2) &= max(ord(\tau_1)+1, ord(\tau_2)) \end{aligned}$$

- Clausal order:

$$\begin{aligned} ord(A) &= 0 \quad \text{(if } A \text{ is atomic or } \top\text{)} \\ ord(B_1 \wedge B_2) &= max(ord(B_1), ord(B_2)) \\ ord(B_1 \vee B_2) &= max(ord(B_1), ord(B_2)) \\ ord(B_1 \supset B_2) &= max(ord(B_1)+1, ord(B_2)) \\ ord(\forall x\ B) &= ord(B) \\ ord(\exists x\ B) &= ord(B) \end{aligned}$$

## First-Order Restrictions

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

  Example:

  ```
  int,  int -> int,  int -> o,  int -> int -> int
  ```

  But not:

  ```
  (int -> int) -> int,  o -> o
  ```

- Clausal order is either 0 or 1.

  Example:

  ```
  adj a b.
  path X Y :- adj X Z, path Z Y.
  ```

## First-Order Hereditary Harrop Formulas

- Goal Formulas:

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x\ G \mid D \supset G \mid \forall_\tau x\ G$$

- Definite Clauses:

$$D := A \mid G \supset A \mid \forall_\tau x\ D$$

- 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 $G_1 \wedge G_2$ from $\langle \Sigma, \mathcal{P} \rangle$, attempt to prove both $G_1$ and $G_2$ from $\langle \Sigma, \mathcal{P} \rangle$.

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

**INSTANCE** To prove $\exists_\tau x\ G$ 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 $\forall_\tau x\ G$ 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

- $\alpha$-conversion:

  $\lambda x.s = \lambda y.s[y/x]$ provided $y$ does not occur free in $s$.
- $\beta$-conversion:

  $(\lambda x.s)t = s[t/x]$
- $\eta$-conversion:

  $\lambda x.(sx) = s$ provided $x$ does not occur free in $s$.

$\lambda$Prolog implements $\lambda$-conversion as its notion of equality. The following terms are equivalent.

```
x\(f x)   y\(f y)   (g\x\(g x) f)   f
```

$\lambda$Prolog programs cannot determine the name of a bound variable.

# Substitution and Quantification

```
p X :- pi y\(q X y).
```

- Substitute `(f a)` for `X`.

  `p (f a) :- pi y\(q (f a) y).`
- Substitute `(f y)` for `X`.

  `p (f y) :- pi y\(q (f y) y).`
- *Variable capture* must be avoided.

  `p (f y) :- pi z\(q (f y) z).`

# Implication and Universal Quantification in Goals

```
kind  bug,jar           type.
type  j                 jar.
type  sterile, heated   jar -> o.
type  dead, bug         insect -> o.
type  in                insect -> jar -> o.

sterile J :- pi x\(bug x => in x J => dead x).
dead B    :- heated j, in B j, bug B.
heated j.

?- sterile j.
```

$\langle \Sigma, \mathcal{P} \rangle$ `?- pi x\(bug x => in x j => dead x).`

$\langle \Sigma \cup \{\texttt{g}\}, \mathcal{P} \rangle$ `?- bug g => in g j => dead g.`

$\langle \Sigma \cup \{\texttt{g}\}, \mathcal{P} \cup \{\texttt{bug g, in g j}\} \rangle$ `?- dead g.`

# Logic Variables in Programs

```
type  reverse     list A -> list A -> o.
type  rev         list A -> list A -> list A -> o.
type  rv          list A -> list A -> o.

reverse L K :-
   pi L\(rev nil L L) =>
   pi X\(pi L\(pi K\(pi M\(rev (X::L) K M :- rev L K (X::M)))))
      => rev L K nil.

?- reverse (1::2::nil) K.


reverse L K :-
   rv nil K =>
   pi X\(pi L\(pi K\(rv (X::L) K :- rv L (X::K))))
      => rv L nil.
```

# Abstract Data Types

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

?- pi emp\(pi stk\(
      empty emp =>
      pi S\(pi X\(enter X S (stk X S))) =>
      pi S\(pi X\(remove X (stk X S) S)) =>
          sigma S1\(sigma S2\(sigma S3\(sigma S4\(sigma S5\
            (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\( ... =>
          sigma U\(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 $\supset$.

- Rigid Atomic Formulas:

  $A_r$ is an atomic formula whose top-level symbol is also not a variable.

- Goal Formulas:

  $$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x\ G$$

- Definite Clauses:

  $$D := A_r \mid G \supset A_r \mid \forall_\tau x\ D$$

- No restrictions on order of types. Restrictions on clausal order still hold. Terms instantiating $x$ also cannot contain any occurrences of $\supset$.

# Examples of Higher-Order Programs

```
type  mappred   (A -> B -> o) -> list A -> list B -> o.
type  forevery  (A -> o) -> list A -> o.
type  forsome   (A -> o) -> list A -> o.
type  sublist   (A -> o) -> list A -> list A -> o.

mappred P nil nil.
mappred P (X :: L) (Y :: K) :- P X Y, mappred P L K.

forevery P nil.
forevery P (X :: L) :- P X, forevery P L.

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

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.
```

# The mappred Program

```
type  mappred  (A -> B -> o) -> list A -> list B -> o.
mappred P nil nil.
mappred P (X :: L) (Y :: K) :- P X Y, mappred P L K.

type  age      person -> int -> o.
age bob 23.
age sue 24.
age ned 23.

?- mappred age (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

```
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 forsome Program

```
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.
```

## The forevery Program

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

age bob 23.
age sue 24.
age ned 23.

?- forevery (x\(sigma y\(age x y))) (ned::bob::sue::nil).
yes.

?- forevery (x\(age x A)) (ned::bob::sue::nil).
no.

?- forevery (x\(age x A)) (ned::bob::nil).
A == 23.
```

## Computing with λ-terms

```
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 λterms 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 λterms 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:

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x\ G \mid D \supset G \mid \forall_\tau x\ G$$

- Definite Clauses:

$$D := A \mid G \supset A \mid \forall_\tau x\ D$$

- 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 $\supset$ in a goal formula, then it is a *positive* subformula occurrence. If it occurs to the left of an odd number of occurrences of $\supset$, 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 $\lambda$-abstraction. Otherwise, it is *essentially existential.*
- In terms of the $\lambda$Prolog 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 $\lambda$Prolog

- 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.

## More Implementations of reverse

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

reverse L K :- pi rev\(
   pi L\(rev nil L L) =>
   pi X\(pi L\(pi K\(pi M\(rev (X::L) K M :- rev L K (X::M))))
      => rev L K nil).

reverse L K :- pi rv\(
   rv nil K =>
   pi X\(pi L\(pi K\(rv (X::L) K :- rv L (X::K))))
      => rv L nil).
```

## Discharging a Constant from a Term

$\langle \Sigma, \mathcal{P} \rangle$ `?- pi y\(append (1::2::nil) y X).`

$\langle \Sigma \cup \{$`k`$\}, \mathcal{P} \rangle$ `?- 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.

$\langle \Sigma, \mathcal{P} \rangle$ `?- pi y\(append (1::2::nil) y (H y)).`

$\langle \Sigma \cup \{$`k`$\}, \mathcal{P} \rangle$ `?- append (1::2::nil) k (H k).`

The terms `(H k)` and `(1::2::k)` are unified. Of the two unifiers, `w\(1::2::k)` and `w\(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

# 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.

# 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.

# Declaring local scope to constants

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

  $$E ::= D \mid \exists_\tau x\ E \mid E_1 \wedge E_2$$

- 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 $(\exists x\ D) \supset G \equiv \forall x(D \supset G)$.

## 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.
```

## 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

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

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

## Example Module using import

```
module int_stack.
import stack.

type    stack_test      int -> int -> o.

stack_test A B :-
  sigma S1\(sigma S2\(sigma S3\(sigma S4\(sigma 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 $\mathbf{L}_\lambda$ Sublanguage

- Restricts $\lambda$Prolog by placing the following restriction on variables:

  > For every subterm in formula $B$ of the form $xy_1 \ldots y_n$ ($n \geq 0$) 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.

## L$_\lambda$ Unification Examples

- An example in L$_\lambda$

```
x\y\(g (U x z) (V y))  =  v\w\(X w)

U == x\y\(W y)    X == w\(g (W w) (V w))
```

- An example that is not in L$_\lambda$

```
(F a) = (g a a)

F == x\(g x x)      F == x\(g a x)
F == x\(g x a)      F == x\(g a a)
```

## 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

## 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.

## 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.                        infixl  or  4.
type    f  i -> i -> i.              infixl  and 5.
type    q  form.                     infixr  imp 6.
type    p  i -> form.
```

$$\forall x \exists y\ (p(x) \supset p(y))$$

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

# Negation Normal Form

$$\begin{aligned} \neg\neg A &\equiv A \\ \neg(A \wedge B) &\equiv \neg A \vee \neg B \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B \\ \neg(A \supset B) &\equiv A \wedge \neg B \\ \neg(\forall x A) &\equiv \exists x(\neg A) \\ \neg(\exists x A) &\equiv \forall x(\neg A) \end{aligned}$$

# 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

$$\frac{A \qquad B}{A \wedge B}\,\wedge\text{-I} \qquad \frac{A}{A \vee B}\,\vee\text{-I} \qquad \frac{B}{A \vee B}\,\vee\text{-I} \qquad \frac{\begin{matrix}(A)\\ B\end{matrix}}{A \supset B}\,\supset\text{-I}$$

$$\frac{\begin{matrix}(A)\\ \bot\end{matrix}}{\neg A}\,\neg\text{-I} \qquad \frac{[t/x]A}{\exists x A}\,\exists\text{-I} \qquad \frac{[y/x]A}{\forall x A}\,\forall\text{-I}$$

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

$$\frac{A \wedge B}{A}\,\wedge\text{-E} \qquad \frac{A \wedge B}{B}\,\wedge\text{-E} \qquad \frac{A \qquad A \supset B}{B}\,\supset\text{-E} \qquad \frac{\forall x A}{[t/x]A}\,\forall\text{-E}$$

$$\frac{A \vee B \qquad \begin{matrix}(A)\\ C\end{matrix} \qquad \begin{matrix}(B)\\ C\end{matrix}}{C}\,\vee\text{-E} \qquad \frac{A \qquad \neg A}{\bot}\,\neg\text{-E} \qquad \frac{\exists x A \qquad \begin{matrix}([y/x]A)\\ B\end{matrix}}{B}\,\exists\text{-E}$$

$$\frac{\bot}{A}\,\bot_{\text{I}} \qquad \frac{\begin{matrix}(\neg A)\\ \bot\end{matrix}}{A}\,\bot_{\text{C}}$$

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

$$\dfrac{\dfrac{p(a) \vee p(b) \qquad \dfrac{p(a)}{\exists x\, p(x)}\,\exists\text{-I} \qquad \dfrac{p(b)}{\exists x\, p(x)}\,\exists\text{-I}}{\exists x\, p(x)}\,\vee\text{-E}}{p(a) \vee p(b) \supset \exists x\, p(x)}\,\supset\text{-I}$$

## Specifying Natural Deduction Rules

$$\frac{A \qquad B}{A \wedge B}\,\wedge\text{-I}$$

$$\frac{A}{A \vee B}\,\vee\text{-I} \qquad\qquad \frac{B}{A \vee B}\,\vee\text{-I}$$

```
type  proof     nprf -> form -> o.
type  and_i     nprf -> nprf -> nprf.

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

$$\frac{[t/x]A}{\exists x A}\ \exists\text{-I}$$

```
type  exists_i  nprf -> nprf.

proof (exists_i P) (exists A) :- proof P (A T).


type  exists_i  i -> nprf -> nprf.

proof (exists_i T P) (exists A) :- proof P (A T).
```

# Specifying Universal Introduction

$$\frac{[y/x]A}{\forall x A}\ \forall\text{-I}$$

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\(proof (P y) (A y)).

type  forall_i  (i -> nprf) -> nprf.
```

# Specifying the Discharge of Assumptions

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{A \supset B}\ \supset\text{-I}$$

```
proof (imp_i P) (A imp B) :-
    pi pA\((proof pA A) => (proof (P pA) B)).

type  imp_i   (nprf -> nprf) -> nprf.
```

# Example Execution

$$\frac{q}{q \supset q}\ \supset\text{-I}$$

```
proof (imp_i P) (A imp B) :-
    pi pA\((proof pA A) => (proof (P pA) B)).
```

$\langle \Sigma, \mathcal{P} \rangle$ `?- proof R (q imp q).`

$\langle \Sigma, \mathcal{P} \rangle$ `?- pi pA\((proof pA q) => (proof (R1 pA) q)).`

$\langle \Sigma \cup \{\texttt{pa}\}, \mathcal{P} \rangle$ `?- (proof pa q) => (proof (R1 pa) q)).`

$\langle \Sigma \cup \{\texttt{pa}\}, \mathcal{P} \cup \{\texttt{proof pa q}\}$ `?- proof (R1 pa) q.`

Unification Problem: `R` = `(imp_i R1)`, `(R1 pa)` = `pa`<br>
Solution: `R1` := `x\x`, `R` := `(imp_i x\x)`<br>
Not a Solution: `R1` := `x\pa`

# Elimination Rules

$$\frac{A \wedge B}{A}\,\wedge\text{-E} \qquad \frac{A \wedge B}{B}\,\wedge\text{-E} \qquad \frac{A \qquad A \supset B}{B}\,\supset\text{-E} \qquad \frac{\forall x A}{[t/x]A}\,\forall\text{-E}$$

$$\frac{A \vee B \qquad \begin{array}{c}(A)\\ C\end{array} \qquad \begin{array}{c}(B)\\ C\end{array}}{C}\,\vee\text{-E} \qquad \frac{A \qquad \neg A}{\bot}\,\neg\text{-E} \qquad \frac{\exists x A \qquad \begin{array}{c}([y/x]A)\\ B\end{array}}{B}\,\exists\text{-E}$$

```
proof (and_e P) A :- proof P (A and B); proof P (B and A).
proof (forall_e P) (A T) :- proof P (forall A).
proof (exists_e P1 P2) B :- proof P1 (exists A),
  pi y\(pi p\ ((proof p (A y)) => (proof (P2 Y p) B))).
proof (or_e P P1 P2) C :- proof P (A or B),
  pi pA\((proof pA A) => (proof (P1 pA) C)),
  pi pB\((proof pB B) => (proof (P2 pB) C)).
proof (imp_e P1 P2) B :- proof P1 A, proof P2 (A imp B).
proof (neg_e P1 P2) false :- proof P1 A, proof P2 (neg A).
```

# Remaining Rules

$$\frac{\begin{array}{c}(A)\\ \bot\end{array}}{\neg A}\,\neg\text{-I} \qquad \frac{\bot}{A}\,\bot_{\mathrm{I}} \qquad \frac{\begin{array}{c}(\neg A)\\ \bot\end{array}}{A}\,\bot_{\mathrm{C}}$$

```
proof (neg_i P) (neg A) :-
    pi pA\((proof pA A) => (proof (P pA) false)).
proof (false_i P) A :- proof P false.
proof (false_c P) A :-
  pi p\((proof p (neg A)) => (proof (P p) false)).
```

# Sequent Calculus I

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\,\wedge\text{-R} \qquad \frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\,\vee\text{-R} \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\,\vee\text{-R}$$

$$\frac{A, \Gamma \longrightarrow B}{\Gamma \longrightarrow A \supset B}\,\supset\text{-R} \qquad \frac{A, \Gamma \longrightarrow \bot}{\Gamma \longrightarrow \neg A}\,\neg\text{-R}$$

$$\frac{\Gamma \longrightarrow [y/x]A}{\Gamma \longrightarrow \forall x A}\,\forall\text{-R} \qquad \frac{\Gamma \longrightarrow [t/x]A}{\Gamma \longrightarrow \exists x A}\,\exists\text{-R}$$

Proviso on ∀-R: $y$ cannot appear free in the lower sequent.

# Sequent Calculus II

$$\frac{A, B, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C}\,\wedge\text{-L} \qquad \frac{A, \Gamma \longrightarrow C \qquad B, \Gamma \longrightarrow C}{A \vee B, \Gamma \longrightarrow C}\,\vee\text{-L}$$

$$\frac{\Gamma \longrightarrow A \qquad B, \Gamma \longrightarrow C}{A \supset B, \Gamma \longrightarrow C}\,\supset\text{-L} \qquad \frac{\Gamma \longrightarrow A}{\neg A, \Gamma \longrightarrow \bot}\,\neg\text{-L}$$

$$\frac{[t/x]A, \Gamma \longrightarrow C}{\forall x A, \Gamma \longrightarrow C}\,\forall\text{-L} \qquad \frac{[y/x]A, \Gamma \longrightarrow C}{\exists x A, \Gamma \longrightarrow C}\,\exists\text{-L}$$

$$\frac{\Gamma \longrightarrow \bot}{\Gamma \longrightarrow A}\,\bot\text{-R} \qquad A, \Gamma \longrightarrow A \qquad \text{(initial)}$$

Proviso on ∃-L: $y$ cannot appear free in the lower sequent.

# Specifying Sequent Systems

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge\text{-R}$$

```
type    -->     (list form) -> form -> seq.
infix   -->     4.
type    proof   sprf -> seq -> o.
type    and_r   sprf -> sprf -> sprf.

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

# Antecedent Rules

$$\frac{\Gamma \longrightarrow A \qquad B, \Gamma \longrightarrow C}{A \supset B, \Gamma \longrightarrow C}\ \supset\text{-L}$$

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

memb X (X :: L).
memb X (Y :: L) :- memb X L.


type    imp_l   sprf -> sprf -> sprf.

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

# Initial Sequents

$$A, \Gamma \longrightarrow A$$

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

# Classical Logic

$$\frac{\Gamma \longrightarrow A, \Delta \qquad \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \wedge B, \Delta}\ \wedge\text{-R}$$

```
type    -->     (list form) -> (list form) -> seq.

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

# Modal Sequents

$$\dfrac{R;\Gamma \longrightarrow A_w,\Delta \qquad R;\Gamma \longrightarrow B_w,\Delta}{R;\Gamma \longrightarrow (A\wedge B)_w,\Delta}\ \wedge\text{-R}$$

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

type    m       form -> world -> mform.
type    r       world -> world -> wpair.
type    mseq    (list wpair) ->
                   (list form) -> (list form) -> seq.

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)).
```

# The Modal Rules

$$\dfrac{[R\vdash r(w,x)] \qquad R;A_x,\Gamma \longrightarrow \Delta}{R;(\Box A)_w,\Gamma \longrightarrow \Delta}\ \Box\text{-L}$$

$$\dfrac{R,r(w,x);\Gamma \longrightarrow \Delta,A_x}{R;\Gamma \longrightarrow \Delta,(\Box A)_w}\ \Box\text{-R}$$

Proviso on □-R: world $x$ doesn't occur in the lower sequent.

$$\dfrac{\dfrac{\dfrac{\dfrac{[r(w,v),r(v,u)\vdash r(w,u)] \qquad r(w,v),r(v,u);p_u \longrightarrow p_u}{r(w,v),r(v,u);(\Box_a p)_w \longrightarrow p_u}\ \Box\text{-L}}{r(w,v);(\Box_a p)_w \longrightarrow (\Box_a p)_v}\ \Box\text{-R}}{(\Box_a p)_w \longrightarrow (\Box_a\Box_a p)_w}\ \Box\text{-R}}{\longrightarrow (\Box_a p \supset \Box_a\Box_a p)_w}\ \supset\text{-R}$$

# Specifying the Modal Introduction Rule

$$\dfrac{R,r(w,x);\Gamma \longrightarrow \Delta,A_x}{R;\Gamma \longrightarrow \Delta,(\Box A)_w}\ \Box\text{-R}$$

```
type    box     form -> form.

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

# Specifying the Modal Elimination Rule

$$\dfrac{[R\vdash r(w,x)] \qquad R;A_x,\Gamma \longrightarrow \Delta}{R;(\Box A)_w,\Gamma \longrightarrow \Delta}\ \Box\text{-L}$$

```
type    related     (list wpair) -> wpair -> 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).
```

## $\beta\eta$-Convertibility for the Untyped $\lambda$-Calculus

```
kind  tm      type.

type  app     tm -> tm -> tm.
type  abs     (tm -> tm) -> tm.
```

$\lambda f \lambda n.f(fn)$ $\qquad\qquad$ $\lambda x.xx$

```
(abs f\(abs n\(app f (app f n))))
(abs x\(app x x))
```

## $\beta\eta$-Redexes

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

```
type  redex   tm -> tm -> o.

redex (app (abs S) T)  (S T).

redex (abs x\(app S x)) S.
```

## One-Step Reducibility

$$\frac{M \to P}{MN \to PN} \qquad \frac{N \to P}{MN \to MP} \qquad \frac{M \to N}{\lambda x.M \to \lambda x.N}$$

```
type  red1    tm -> tm -> o.

red1 M N :- redex M N.
red1 (app M N) (app P N) :- red1 M P.
red1 (app M N) (app M P) :- red1 N P.
red1 (abs M) (abs N) :- pi x\(red1 (M x) (N x)).
```

## $\beta\eta$-Convertibility and Normalization

```
conv M N :- red1 M N.
conv M M.
conv M N :- conv N M.
conv M N :- conv M P, conv P N.



norm M N :- red1 M P, !, norm P N.
norm M M.
```

## Correctness of Representation of $\lambda$-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:

$$\langle\!\langle \lambda f \lambda n. f(fn) \rangle\!\rangle_{\rho}^{\Phi} \equiv$$

```
(abs f\(abs n\(app f (app f n))))
```

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

## Correctness of $\beta\eta$-Convertibility Specification

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

$$\texttt{(conv}\ \langle\!\langle M \rangle\!\rangle_{\rho}^{\Phi}\ \langle\!\langle N \rangle\!\rangle_{\rho}^{\Phi}\texttt{)}$$

is provable.

## Evaluation for a Functional Language

$$\begin{array}{rlrl}
app & : tm \rightarrow tm \rightarrow tm & nil & : tm \\
abs & : (tm \rightarrow tm) \rightarrow tm & cons & : tm \rightarrow tm \rightarrow tm \\
0 & : tm & hd & : tm \rightarrow tm \\
s & : tm \rightarrow tm & tl & : tm \rightarrow tm \\
true & : tm & empty & : tm \rightarrow tm \\
false & : tm & fix & : (tm \rightarrow tm) \rightarrow tm \\
if & : tm \rightarrow tm \rightarrow tm \rightarrow tm & let & : (tm \rightarrow tm) \rightarrow tm \rightarrow tm
\end{array}$$

$$\begin{array}{rlrl}
app\ (abs\ M)\ N & \rightarrow MN & empty\ nil & \rightarrow true \\
if\ true\ M\ N & \rightarrow M & empty\ (cons\ M\ N) & \rightarrow false \\
if\ false\ M\ N & \rightarrow N & fix\ M & \rightarrow M\ (fix\ M) \\
hd\ (cons\ M\ N) & \rightarrow M & let\ M\ N & \rightarrow MN \\
tl\ (cons\ M\ N) & \rightarrow N & &
\end{array}$$

## 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 tru, eval M M'.
eval (if C M N) N' :- eval C fals, 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) tru :- eval L nill.
eval (empty L) fals :- 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.

## Part III: Implementing Automatic Theorem Provers

An Automatic Prover for First-Order Classical Logic

- A strategy for finding sequent proofs
- An implementation using three subprocedures

## Reversibility of Rules

$$\frac{A, B, \Gamma \longrightarrow \Delta}{A \wedge B, \Gamma \longrightarrow \Delta}\ \wedge\text{-L} \qquad \frac{\Gamma \longrightarrow A, \Delta \qquad \Gamma \longrightarrow B, \Delta}{\Gamma \longrightarrow A \wedge B, \Delta}\ \wedge\text{-R}$$

$\wedge$-L: There is a proof of one of the formulas in $\Delta$ from $A \wedge B$ and $\Gamma$ if and only if there is a proof of one of the formulas in $\Delta$ from $A$ and $B$ and $\Gamma$.

$\wedge$-R: There is a proof of $A \wedge B$ 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:

$$\frac{[t/x]A, \Gamma \longrightarrow \Delta}{\forall x A, \Gamma \longrightarrow \Delta}\ \forall\text{-L} \qquad \frac{\Gamma \longrightarrow [t/x]A, \Delta}{\Gamma \longrightarrow \exists x A, \Delta}\ \exists\text{-R}$$

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

$$\frac{A, B, \Gamma \longrightarrow \Delta}{A \wedge B, \Gamma \longrightarrow \Delta}\,\wedge\text{-L}$$

```
type  memb_and_rest   A -> (list A) -> (list A) -> o.

memb_and_rest A (A::L) L.
memb_and_rest A (B::L) (B::K) :- memb_and_rest A L K.

proof1 (and_l P) (Gamma --> Delta) :-
   memb_and_rest (A and B) Gamma Gamma1,
   proof1 P ((A::B::Gamma1) --> Delta).
```

## Step 1 of 3: the proof1 procedure

1. Apply all rules except ∀-L and ∃-R 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.

```
proof1 (initial A) (Gamma --> Delta) :-
   memb A Gamma, memb A Delta.

proof1 (and_r P1 P2) (Gamma --> Delta) :-
   memb_and_rest (A and B) Delta Delta1,
   proof1 P1 (Gamma --> (A::Delta1)),
   proof1 P2 (Gamma --> (B::Delta1)).

proof1 (imp_l P1 P2) (Gamma --> Delta) :-
   memb_and_rest (A imp B) Gamma Gamma1,
   proof1 P1 (Gamma1 --> (A::Delta)),
   proof1 P2 ((B::Gamma1) --> Delta).
⋮
```

## Step 2 of 3: the proof2 procedure

2. Apply all rules including versions of the rules for ∀-L and ∃-R that remove the quantified formula after applying the rule, and try to complete the proof. Stop if a proof is successfully completed.

```
proof2 (forall_l P) (Gamma --> Delta) :-
   memb_and_rest (forall A) Gamma Gamma1,
   proof2 P (((A T)::Gamma1) --> Delta).

proof2 (exists_r P) (Gamma --> Delta) :-
   memb_and_rest (exists A) Delta Delta1,
   proof2 P (Gamma1 --> ((A T)::Delta1)).
```

⋮

(plus duplicates of each of the `proof1` clauses)

## Step 3 of 3: the nprove procedure

3. Add an additional copy of each quantified formula to the sequents obtained from step 1, and repeat steps 2 and 3.

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

amplify 1 Seq Seq.
amplify N (Gamma1 --> Delta1) (Gamma2 --> Delta2) :-
  N > 1,
  amplify_forall N Gamma1 Gamma2,
  amplify_exists N Delta1 Delta2.
```

```
add_copies 1 A L (A::L).
add_copies N A L (A::K) :-
  N > 1, M is (N - 1),
  add_copies M A L K.

amplify_forall N nil nil.
amplify_forall N ((forall A)::Gamma) Gamma2 :-
  amplify_forall N Gamma Gamma1,
  add_copies N (forall A) Gamma1 Gamma2.
amplify_forall N (A::Gamma) (A::Gamma1) :-
  amplify_forall N Gamma Gamma1.

amplify_exists N nil nil.
amplify_exists N ((exists A)::Delta) Delta2 :-
  amplify_exists N Delta Delta1,
  add_copies N (exists A) Delta1 Delta2.
amplify_exists N (A::Delta) (A::Delta1) :-
  amplify_exists N Delta Delta1.
```

# 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.

$$\dfrac{\dfrac{\dfrac{p(a)\longrightarrow p(a)}{p(a)\longrightarrow \exists x\, p(x)}\;\exists\text{-R}\qquad\dfrac{p(b)\longrightarrow p(b)}{p(b)\longrightarrow \exists x\, p(x)}\;\exists\text{-R}}{p(a)\vee p(b)\longrightarrow \exists x\, p(x)}\;\vee\text{-L}}{\longrightarrow p(a)\vee p(b)\supset \exists x\, p(x)}\;\supset\text{-R}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\longrightarrow p(t)\supset p(z), p(z)\supset p(w)}{\longrightarrow p(t)\supset p(z), \forall y\ (p(z)\supset p(y))}\;\forall\text{-R}}{\longrightarrow p(t)\supset p(z), \exists x\ \forall y\ (p(x)\supset p(y))}\;\exists\text{-R}}{\longrightarrow \forall y\ (p(t)\supset p(y)), \exists x\ \forall y\ (p(x)\supset p(y))}\;\forall\text{-R}}{\longrightarrow \exists x\ \forall y\ (p(x)\supset p(y)), \exists x\ \forall y\ (p(x)\supset p(y))}\;\exists\text{-R}}{\longrightarrow \exists x\ \forall y\ (p(x)\supset p(y))}\;\text{amplify } 2$$

# Part IV: Implementing Interactive Tactic Theorem Provers

- Inference Rules as Tactics
- A Goal Reduction Tactical
- Some Common Tacticals
- 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 $\lambda$Prolog. They implement an interpreter for goal-directed theorem proving in the logic programming setting.

## Inference Rules As Tactics

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-I}$$

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

```
and_i_tac (proof (and_i P1 P2) (A and B))
          ((proof P1 A) ^^ (proof P2 B)).
```

```
type  and_i_tac  goal -> goal -> o.
type  proof      nprf -> form -> goal.
type  ^^         goal -> goal -> goal.
```

```
infix ^^         3.
```

## Tactics with Assumption Lists

```
and_i_tac (proof (and_i P1 P2) (A and B))
          ((proof P1 A) ^^ (proof P2 B)).
```

```
kind  judg    type.
type  proof   nprf -> form -> judg.
type  deduct  (list judg) -> judg -> goal.
```

```
and_i_tac (deduct Gamma (proof (and_i P1 P2) (A and B)))
          ((deduct Gamma (proof P1 A)) ^^
           (deduct Gamma (proof P2 B))).
```

## Goal Constructors

```
type    tt          goal.
type    ^^          goal -> goal -> goal.
type    vv          goal -> goal -> goal.
type    all         (A -> goal) -> goal.
type    some        (A -> goal) -> goal.
type    ==>>        o -> goal -> goal.
```

```
infixl  ^^          3.
infixl  vv          3.
infixr  ==>>        3.
```

# A Goal Reduction Tactical

```
type  maptac  (goal -> goal -> o) -> goal -> goal -> o.

maptac Tac tt tt.
maptac Tac (InGoal1 ^^ InGoal2) (OutGoal1 ^^ OutGoal2) :-
   maptac Tac InGoal1 OutGoal1,
   maptac Tac InGoal2 OutGoal2.
maptac Tac (all InGoal) (all OutGoal) :-
   pi x\(maptac Tac (InGoal x) (OutGoal x)).
maptac Tac (InGoal1 vv InGoal2) OutGoal :-
   maptac Tac InGoal1 OutGoal; maptac Tac InGoal2 OutGoal.
maptac Tac (some InGoal) OutGoal :-
   sigma T\(maptac Tac (InGoal T) OutGoal).
maptac Tac (D ==>> InGoal) (D ==>> OutGoal) :-
   D => (maptac Tac InGoal OutGoal).
maptac Tac InGoal OutGoal :- Tac InGoal OutGoal.
```

# Tacticals

```
then Tac1 Tac2 InGoal OutGoal :-
   Tac1 InGoal MidGoal,
   maptac Tac2 MidGoal OutGoal.

orelse Tac1 Tac2 InGoal OutGoal :-
   Tac1 InGoal OutGoal; Tac2 InGoal OutGoal.

idtac Goal Goal.

repeat Tac InGoal OutGoal :-
   orelse (then Tac (repeat Tac)) idtac InGoal OutGoal.

try Tac InGoal OutGoal :-
   orelse Tac idtac InGoal OutGoal.
```

# Interactive Theorem Proving

```
type    query       (goal -> o) -> goal -> goal -> o.
type    inter       (goal -> o) -> goal -> goal -> o.
type    with_tacs   string -> (goal -> goal -> o)
                       -> goal -> goal -> o.

query PrintPred InGoal OutGoal :-
  PrintPred InGoal,
  print "Enter tactic:", readtac Tac,
  (Tac = backup, !, fail; Tac InGoal OutGoal).

inter PrintPred InGoal OutGoal :-
  repeat (query PrintPred) InGoal OutGoal.

with_tacs M Tac InGoal OutGoal :-
  M ==> (Tac InGoal OutGoal).
```

# Interactive Tactics for Natural Deduction

- Allowing the User to Specify Substitution Terms

```
exists_i_tac (proof (exists_i P) (exists A))
             (proof P (A T)).

exists_i_sub (proof (exists_i P) (exists A))
             (proof P (A T)) :-
   print "Enter substitution term:", read T.
```

- Adding Lemmas

```
modus_ponens (proof P A)
             ((proof Q B) ^^
              ((assump Q B) ==>> (proof P A))) :-
   print "Enter lemma:", read B.

close_tac (proof P A) tt :- assump P A.
```

## Natural Deduction Inference Rule Tactics

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{A \supset B}\ \supset\text{-I}$$

```
proof (imp_i P) (A imp B) :-
    pi pA\((proof pA A) => (proof (P pA) B)).
```

```
imp_i_tac (proof (imp_i P) (A imp B))
    (all pA\((assump pA A) ==>> (proof (P pA) B))).
```

```
imp_i_tac (deduct Gamma (proof (imp_i P) (A imp B)))
    (all pA\(deduct ((proof pA A)::Gamma)
                    (proof (P pA) B)).
```

All introduction rules can be translated to tactics similarly.

## Elimination Rules as Tactics

$$\frac{A \wedge B}{A}\wedge\text{-E} \qquad\qquad \frac{A \wedge B}{B}\wedge\text{-E}$$

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

```
and_e_tac (deduct Gamma (proof P A))
   ((deduct Gamma (proof (and_e1 P) (A and B))) vv
    (deduct Gamma (proof (and_e1 P) (B and A)))).
```

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

## Forward Proof Using Elimination Rules

```
and_e_tac N (deduct Gamma (proof PC C))
            (deduct ((proof (and_e1 P) A)::
                     (proof (and_e2 P) B)::Gamma)
                    (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
    (proof P (((q a) or (q b)) imp (exists x\(q x))))
    OutGoal.

Assumptions:

Conclusion:
(q a or q b) imp (exists x\(q x))
Enter tactic:  ?- imp_i_tac.

Assumptions:
1 q a or q b

Conclusion:
exists x\(q x)
Enter tactic:  ?- exists_i_tac.
```

```
Assumptions:
1 q a or q b

Conclusion:
q T
Enter tactic:  ?- or_e_tac 1.

Assumptions:
1 q a
2 q a or q b

Conclusion:
q T
Enter tactic:  ?- close_tac 1.
```

```
Assumptions:
1 q b
2 q a or q b

Conclusion:
q a

Enter tactic:  ?- backup.
⋮
Enter tactic:  ?- backup.
⋮
Enter tactic:  ?- backup.
```

```
Assumptions:
1 q a or q b

Conclusion:
exists x\(q x)
Enter tactic:  ?- then (or_e_tac 1)
                    (then exists_i_tac
                          (close_tac 1)).

P = imp_i p\(or_e p p1\(exists_i a p1)
                    p2\(exists_i b p2))
OutGoal = all p\((all p1\tt) ^^ (all p2\tt))
```

## 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 $\lambda$Prolog 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 $\lambda$Prolog, 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 Rewriting

- Higher-order rewrite systems use $\lambda$-terms as a meta-language for expressing the equality relation for object languages that include notions of bound variables [Nipkow LICS'91, Klop 80, Aczel 78]
- Many operations on formulas and programs can be naturally expressed as higher-order rewrite systems.
- Capabilities for rewriting can be added to tactic style theorem provers, used to reason about the equality relation of a particular object logic, and combined with other theorem proving techniques.
- Higher-order logic programming allows:
  - a natural specification of higher-order rewrite systems
  - powerful mechanisms for descending through terms and matching terms with rewrite templates

# Higher-Order Rewrite Rules

A rewrite rule is a pair $l \longrightarrow r$ such that $l$ and $r$ are simply-typed $\lambda$-terms of the same primitive type, $l$ is a term in $\mathrm{L}_\lambda$, 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 $s$.

```
type  app    tm -> tm -> tm.
type  abs    (tm -> tm) -> tm.
type  redex  tm -> tm -> o.

redex (app (abs S) T)  (S T).
redex (abs x\(app S x)) S.
```

# Three Parts of a Rewriting Procedure

- Rewrite Rules

  ```
  redex (app (abs S) T)  (S T).
  redex (abs x\(app S x)) S.
  ```

- Congruence and One-Step Rewriting

  ```
  red1 M N :- redex M N.
  red1 (app M N) (app P N) :- red1 M P.
  red1 (app M N) (app M P) :- red1 N P.
  red1 (abs M) (abs N) :- pi x\(red1 (M x) (N x)).
  ```

- Multiple Step Reduction

  ```
  reduce M N :- red1 M P, reduce P N.
  reduce M M.
  ```

## Rewriting in a Tactic Theorem Prover

- The previous example implements the leftmost-outermost rewrite stategy. 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.

## Rewrite and Congruence Rules as Tactics

```
type    ==          A -> A -> goal.
infix   ==          7.
type    prim        goal -> goal.
type    rew         goal -> goal -> o.
type    cong        goal -> goal -> o.
type    cong_const  goal -> goal -> o.

rew (prim ((app (abs S) T) == (S T))) tt.
rew (prim ((abs x\(app S x)) == S)) tt.

cong (prim ((app M N) == (app P Q)))
     ((prim (M == P)) ^^ (prim (N == Q))).
cong (prim ((abs M) == (abs N)))
     (all x\((cong_const (prim (x == x)) tt) ==>>
               (prim ((M x) == (N x))))).

cong_const (prim (f == f)) tt.
```

## Example 2: Evaluation as Rewriting

$$\begin{array}{rlcrl}
app & : tm \to tm \to tm & \qquad & nil & : tm \\
abs & : (tm \to tm) \to tm & & cons & : tm \to tm \to tm \\
0 & : tm & & hd & : tm \to tm \\
s & : tm \to tm & & tl & : tm \to tm \\
true & : tm & & empty & : tm \to tm \\
false & : tm & & fix & : (tm \to tm) \to tm \\
if & : tm \to tm \to tm \to tm & & let & : (tm \to tm) \to tm \to tm
\end{array}$$

## Congruence Tactics for Evaluation

```
cong_const (prim (tru == tru)) tt.
cong_const (prim (fals == fals)) tt.
cong_const (prim (z == z)) tt.
cong_const (prim (nill == nill)) tt.

cong (prim ((s M) == (s N))) (prim (M == N)).
cong (prim ((cons M N) == (cons P Q)))
     ((prim (M == P)) ^^ (prim (N == Q))).
cong (prim ((hd M) == (hd N))) (prim (M == N)).
cong (prim ((tl M) == (tl N))) (prim (M == N)).
cong (prim ((empty M) == (empty N))) (prim (M == N)).
cong (prim ((if C M N) == (if D P Q)))
     ((prim (C == D)) ^^ (prim (M == P)) ^^ (prim (N == Q))).
cong (prim ((fix M) == (fix N)))
     (all x\((cong_const (prim (x == x)) tt) ==>> (prim ((M x) == (N x))))).
cong (prim ((let M N) == (let P Q)))
     ((all x\((cong_const (prim (x == x)) tt) ==>>
                (prim ((M x) == (P x))))) ^^ (prim (N == Q))).
```

## Evaluation Rewrite Rules

$$
\begin{array}{rclcrcl}
\mathit{app}\ (\mathit{abs}\ M)\ N & \rightarrow & M N & \qquad & \mathit{empty}\ \mathit{nil} & \rightarrow & \mathit{true} \\
\mathit{if}\ \mathit{true}\ M\ N & \rightarrow & M & & \mathit{empty}\ (\mathit{cons}\ M\ N) & \rightarrow & \mathit{false} \\
\mathit{if}\ \mathit{false}\ M\ N & \rightarrow & N & & \mathit{fix}\ M & \rightarrow & M\ (\mathit{fix}\ M) \\
\mathit{hd}\ (\mathit{cons}\ M\ N) & \rightarrow & M & & \mathit{let}\ M\ N & \rightarrow & M N \\
\mathit{tl}\ (\mathit{cons}\ M\ N) & \rightarrow & N & & & &
\end{array}
$$

## Tactics Implementing Evaluation Rewrites

```
rew (prim ((app (abs M) N) == (M N))) tt.
rew (prim ((abs X (app M X)) == M)) tt.
rew (prim ((hd (cons M N)) == M)) tt.
rew (prim ((tl (cons M N)) == N)) tt.
rew (prim ((empty nill) == tru)) tt.
rew (prim ((empty (cons M N)) == fals)) tt.
rew (prim ((if tru M N) == M)) tt.
rew (prim ((if fals M N) == N)) tt.
rew (prim ((fix M) == (M (fix M)))) tt.
rew (prim ((let M N) == (M N))) tt.
```

## Example 3: Negation Normal Forms

- Congruence Rules

```
cong (prim ((A and B) == (C and D)))
     ((prim (A == C)) ^^ (prim (B == D))).
cong (prim ((forall A) == (forall B)))
     (all x\((cong_const (prim (x == x)) tt) ==>>
              (prim ((A x) == (B x))))).
```

- Rewrite Rules

```
rew (prim ((neg (A and B)) ==
           ((neg A) or (neg B)))) tt.
rew (prim ((neg (forall A)) ==
           (exists x\(neg (A x))))) tt.
```

## A Modified maptac

```
type  maptacC (goal -> goal -> o) -> goal -> goal -> o.

maptacC Tac tt tt.
maptacC Tac (InGoal1 ^^ InGoal2) OutGoal :-
   Tac (InGoal1 ^^ InGoal2) OutGoal.
maptacC Tac (all InGoal) (all OutGoal) :-
   pi x\(maptacC Tac (InGoal x) (OutGoal x)).
maptacC Tac (InGoal1 vv InGoal2) OutGoal :-
   maptacC Tac InGoal1 OutGoal;
   maptacC Tac InGoal2 OutGoal.
maptacC Tac (some InGoal) OutGoal :-
   sigma T\(maptacC Tac (InGoal T) OutGoal).
maptacC Tac (D ==>> InGoal) (D ==>> OutGoal) :-
   D => (maptacC Tac InGoal OutGoal).
maptacC Tac (prim InGoal) OutGoal :-
   Tac (prim InGoal) OutGoal.
```

# Modified Tacticals Using maptacC

```
thenC Tac1 Tac2 InGoal OutGoal :-
   Tac1 InGoal MidGoal,
   maptacC Tac2 MidGoal OutGoal.

repeatC Tac InGoal OutGoal :-
   orelse (thenC Tac (repeatC Tac)) idtac InGoal OutGoal.
```

# 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

```
lo Cong Rew InG OutG :-
  then (repeat (left_rew (lo_rew Cong Rew)))
       refl InG OutG.

lo_rew Cong Rew InG OutG :-
  orelse Rew (then (thenC Cong
                          (first (lo_rew Cong Rew)))
                   refl) InG OutG.
```

## An Example

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

$$\begin{array}{l}(fix\ \lambda F.(abs\ \lambda l_1.(abs\ \lambda l_2.\\ \quad (if\ (empty\ l_1)\ l_2\ (cons\ (hd\ l_1)\ (app\ (app\ F\ (tl\ l_1))\ l_2))))))\end{array}$$

- The `lo` strategy reduces

  $$(app\ (app\ APP\ (cons\ 0\ nil))\ (cons\ (s\ 0)\ nil))$$

  $$\text{to } (cons\ 0\ (cons\ (s\ 0)\ nil)).$$

  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

$$\begin{tikzcd}
 & \textbf{LF} & \\
\textbf{fohh} \arrow[ur] \arrow[r] & \mathbf{L}_\lambda \arrow[r] & \textbf{hohh} \\
\textbf{fohc} \arrow[u] \arrow[rr] & & \textbf{hohc} \arrow[u]
\end{tikzcd}$$

- LF allows function types of any order, but does not allow *Type* (which is the LF equivalent of o) anywhere in types except as a target type. There is no quantification over *Type*.
- Unlike L$_\lambda$ which restricts the form of terms, LF extends them to allow *dependent types*.
- Note that our example specifications don't use predicate quantification (though the implementation of tactics and tacticals use it extensively). Our encoding "compiles" LF signatures into the sublanguage of hohh without predicate quantification.

## LF Syntax

Syntax for LF Kinds, Types, Objects

$$\begin{array}{rcl} K & := & \mathit{Type} \mid \Pi x : A.K \\ A & := & x \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\ M & := & x \mid \lambda x : A.M \mid MN \end{array}$$

- *Dependent Types*: Types can depend on terms. In particular, in $\Pi x : A.B$, the variable $x$ can occur in the type $B$. $A \to B$ denotes $\Pi x : A.B$ 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)

$$\Gamma \;:=\; \langle\rangle \mid \Gamma, x : K \mid \Gamma, x : A$$

LF Assertions

$$\begin{array}{ll} \Gamma \vdash K\ \mathit{kind} & (K \text{ is a kind in } \Gamma) \\ \Gamma \vdash A : K & (A \text{ has kind } K \text{ in } \Gamma) \\ \Gamma \vdash M : A & (M \text{ has type } A \text{ in } \Gamma) \end{array}$$

Valid Contexts

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

## An LF Signature for Natural Deduction

$$\frac{A \qquad B}{A \wedge B}\,\wedge\text{-I} \qquad \frac{\begin{array}{c}(A)\\ B\end{array}}{A \supset B}\,\supset\text{-I} \qquad \frac{[y/x]A}{\forall x A}\,\forall\text{-I}$$

$\mathit{form} : \mathit{Type}$
$i : \mathit{Type}$

$\wedge : \mathit{form} \to \mathit{form} \to \mathit{form}$
$\forall : (i \to \mathit{form}) \to \mathit{form}$
$\vdots$
$\mathit{true} : \mathit{form} \to \mathit{Type}$

$\wedge\text{-I} : \Pi A : \mathit{form}.\Pi B : \mathit{form}.\mathit{true}(A) \to \mathit{true}(B) \to \mathit{true}(A \wedge B)$
$\supset\text{-I} : \Pi A : \mathit{form}.\Pi B : \mathit{form}.(\mathit{true}(A) \to \mathit{true}(B)) \to \mathit{true}(A \supset B)$
$\forall\text{-I} : \Pi A : i \to \mathit{form}.(\Pi x : i.\mathit{true}(Ax)) \to \mathit{true}(\forall A)$
$\vdots$

## Translating Kind and Type Declarations

- Introducing New Base Types

  $\mathit{form} : \mathit{Type}$
  $i : \mathit{Type}$

```
kind  form    type.
kind  i       type.
```

- Introducing the Syntax of the Object Logic

  $\wedge : \mathit{form} \to \mathit{form} \to \mathit{form}$

```
type  and     form -> form -> form.
```

- Dependent Type Constants as Predicates

  $\mathit{true} : \mathit{form} \to \mathit{Type}$

```
type  proof   form -> o.
```

## Inference Rules as Clauses I

$\wedge\text{-I} : \Pi A : \mathit{form}.\Pi B : \mathit{form}.\mathit{true}(A) \to \mathit{true}(B) \to \mathit{true}(A \wedge B)$

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

An LF term inhabiting the type $\mathit{true}(A \wedge B)$ will be a proof of the formula $A \wedge B$. If we use the above signature item in constructing such a term, this term will have the form:
$(\wedge\text{-I}\ A\ B\ P_1\ P_2)$
We can incorporate proof objects of this form into λProlog specifications.

```
type  proof   nprf -> form -> o.
type  and_i   form -> form -> nprf -> nprf -> o.

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

## Inference Rules as Clauses II

$\supset\text{-I} : \Pi A : \mathit{form}.\Pi B : \mathit{form}.(\mathit{true}(A) \to \mathit{true}(B)) \to \mathit{true}(A \supset B)$

```
type  imp_i form -> form -> (nprf -> nprf) -> nprf.

proof (imp_i A B P) (A imp B) :-
    pi pA\((proof pA A) => (proof (P pA) B)).
```

$\forall\text{-I} : \Pi A : i \to \mathit{form}.(\Pi x : i.\mathit{true}(Ax)) \to \mathit{true}(\forall A)$

```
type  forall_i  (i -> form) -> (i -> nprf) -> nprf.

proof (forall_i A P) (forall A) :-
   pi y\(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].