CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
The Logic Programming Language λProlog

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

Overview of the Language λProlog

The λProlog language is a realization of *hohh* with some additional features

- polymorphism in the style of ML is provided for
- AUGMENT and GENERIC are exploited to provide a form of modularity that is justified via logic
- to support programming, some "impure" control features are inherited into the language from Prolog

With our focus on specification and reasoning, we will eschew use of the last kind of features

We will not discuss polymorphic typing in detail because it will be largely irrelevant to λ-tree syntax applications

We will discuss modularity features but only at a programming/specification level

The Types Language

The language extends simple types by allowing *type variables* denoted by tokens starting with uppercase letters

Several sorts like `int`, `real`, `string` and `o` are built-in as also the type constructor `list`

The programmer can extend this collection via declarations of the form

```
kind    tycon   type -> ... -> type.
```

The number of occurrences of `type` determines the arity.

For example, the declarations

```
kind    i       type.
kind    pairty  type -> type -> type.
```

make it possible to write the type expression

```
i -> (list int) -> (pairty i (list int))
```


The Terms Language

These are typed λ-terms with $\lambda x\ F$ written as `(x \ F)` and $(F\ T)$ written as `(F T)`

Types of variables are generally inferred but can also be specified at binding site: e.g. `((x:int) \ F)`

The types of constants are identified via declarations such as

```
type  -   int -> int -> int.
type  ::  (A -> (list A) -> (list A)).
```

Constants can also be declared to be operators. e.g.

```
infixl  -  255.
infixr  :: 255.
```

I.e., "`-` and `::` are infix and, resp., left and right associative operators of precedence level 255"

Abstraction has lowest precedence and application highest

Built-In Constants in λProlog

- Usual constants associated with `int`, `real`, `string`, etc
- The following polymorphic constants of `list` type

```
type nil  (list A)
type ::   A -> (list A) -> (list A)
```

- The following *logical constants*

```
type   true  o.
type   &, ;, =>, :-, ,   o -> o -> o.
type   sigma, pi         (A -> o) -> o.
infixl  ,   110.
infixr  &   120.
infixl  ;   100.
infix   :-  0.
infixr  =>  130.
```

`=>` and `,` are intended for use at the top-level in goals

`:-` and `&` are intended for use at the top-level in clauses

Conventions Used in Presenting Clauses and Goals

Programs

Listing of clauses, each terminated with a period

Unbound tokens beginning with upper-case letters are implicitly universally quantified over the clause

For example

```
append nil L L.
append (X::L1) L2 (X::L3) :- append L1 L2 L3.
```

Queries

Unbound tokens beginning with upper-case letters are implicitly existentially quantified over the query, e.g.

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

Similar to Prolog syntax, except programs and goals are typed and curried notation is used

A Conservative Extension to *hohh* Program Clauses

Atoms defined by program clauses can have variable heads provided these are bound by essential universal quantifiers

For example, the following definition is acceptable

```
reverse L1 L2 :-
   (pi rev_ aux \
      (((pi L\ rev_aux nil L L) &
        (pi X\ pi L1\ pi L2\ pi L3\
             rev_aux (X::L1) L2 L3 :-
                     rev_aux L1 (X::L2) L3))
           => rev_aux L1 [] L2)).
```

Extension "works" because by the time the embedded clauses are added to the program, these heads become constants

This extension is also useful: e.g. it allows auxiliary predicate names to be hidden

Modules and Signatures

- In the Teyjus system, code is organized into *modules*:

```
module lists.
type rev  (list A) -> (list A) -> o.
type rev_aux  (list A) -> (list A) -> (list A) -> o.
rev L1 L2  :-  rev_aux L1 nil L2.
rev_aux nil L1 L2.
rev_aux (X::L1) L2 L3 :- rev_aux L1 (X::L2) L3.
end
```

- The external views of a module is provided by a *signature*:

```
sig  lists.
type rev  (list A) -> (list A) -> o.
end
```

- Each module resides in a separate file and the signature resides in a related file
- Hidden constants correspond to existential quantification over clauses "explained" via universal quantification in goals

Modules and Signatures (Continued)

- Modules can also be *composed* using the *accumulation* construct

```
module fo_prover.
  accumulate lists.

   ...
end
```

Predicate definitions and data constructors "exported" from `lists` can be used in `fo_prover`

Accumulation can be understood via the following steps

- A module without accumulation corresponds to a *D*-formula
- Accumulation inserts the formula corresponding to the accumulated module in place
- Existential quantifiers over accumulated module are raised to the top-level in a scope respecting way

Animating Specifications Using the Teyjus System

The Teyjus system consists of three conceptual parts:

- An abstract machine supporting, low-level, λProlog relevant operations
- A compiler for translating to abstract machine programs
- A linker for realizing modularity notions with separate compilation

Running a program involves compiling a module to bytecode form, linking it with other modules and running the result

The full system has other utilities such as a disassembler, a dependency analyzer and a pretty printer

For details, check out `http://teyjus.cs.umn.edu` and the google code page

Higher-Order Programming in *hohh*

Predicate variables permit a familiar style of higher-order programming

```
type mappred (A -> B -> o) ->
        (list A) -> (list B) -> o.
mappred P nil nil.
mappred P (X::L1) (Y::L2) :-
          P X Y, mappred L1 L2.
```

Suppose we are given additionally the following clauses

```
parent bob mary.          parent sue peter.
parent mary john.         parent peter james.
```

The following are then sensible goals

```
?- mappred parent (bob::sue::nil) L.
?- mappred (x\ y\ sigma z\
             ((parent x z),(parent z y)))
           (bob::sue::nil) L.
```


Predicate Substitutions in *hohh*

We could also ask if it is possible to synthesize a predicate substitution

For example, consider the goal

```
?- mappred P
           (bob::sue::nil) (mary::peter::nil).
```

However, the computation of goal solving is too complex to try and run in reverse

Logically, there are also too many solutions to such queries: relevant abstractions over any succeeding goals will work

The meta-theory suggests the "largest" set as a solution

Here that solution would be

```
P = x\ y\ true
```

Mapping Function Evaluation

There is also a functional counterpart to `mappred`:

```
type mapfun (A -> B) -> (list A) -> (list B) -> o.
mapfun F nil nil.
mapfun F (X::L1) ((F X)::L2) :- mapfun F L1 L2.
```

Now mapping over list elements involves only β-conversion

For example consider the following query

```
?- mapfun (x\ (x + 5)) [3,7] L
```

As before, we can ask if a reverse computation is also possible

```
?- mapfun F [3,7] [3 + 5, 7 + 5]
```

In this case, the weakness of evaluation permits a meaningful answer

The answer that is provided also shows an interesting capability to analyze syntax, e.g. abstracting structural properties

Structural Analysis and Binding Scope

The explicit scoping of quantifiers in *hohh* allows richer structure analysis problems to be posed

Suppose that = is defined as follows

```
type  =  A -> A -> o.
infix =  50.
X = X.
```

Then compare the following

```
?- pi g\ pi a\ sigma F\ (F a) = (g a a).
?- pi g\ sigma F\ pi a\ (F a) = (g a a).
```

A similar issue arises simply from the presence of λ-abstraction

```
?- (x\ F + (G x)) = (x\ (2 + x)).
?- (x\ F + (G x)) = (x\ (x + x)).
?- (x\ (F x) + (G x)) = (x\ (x + x)).
```


Limitations of Equation Solving

The simply typed λ-calculus without any defined (non-logical) constants provides a weak setting for equation solving

For example, the following query has no solutions

```
?- X + 3 = 8.
```

even though the substitution of `5` for *X* would work if `+` were interpreted

As another example, consider

```
?- (F a) = c, (F b) = d
```

A solution to this would require a built-in conditional

The structure analysis capability obtained through our term language is quite weak

However, this an asset rather than a problem: this is what makes structure analysis all about syntax and not “semantics”

Higher-Order Abstract Syntax (HOAS)

This term refers to the use of λ-abstraction in the meta-language to represent binding in object languages

There are varied arguments for HOAS, e.g.

- binding is another notion to be rightly abstracted away in treating syntactic objects
- treating binding in this way makes meta-language devices available for treating issues like scope and substitution

However, the interpretation of HOAS depends a lot on the strength of the λ-calculus and how it is used in representation

Using a calculus that supports computation in a significant way also gives rise to problems with representational adequacy

λProlog, which actually pioneered the HOAS idea, supports a particular variant that is free of such problems

λ-Tree Syntax

In this approach, representations use only a *very basic* treatment of binding beyond the encoding of first-order structure

λProlog supports this idea through the following

- only a very weak λ-calculus is used
  λ-terms are for representation, *not* for evaluation
- λ-terms are treated *intensionally*
  I.e., the analysis of syntactic objects can be realized directly through the analysis of their representations
- logical devices are provided for realizing recursion over binding structure
  The use of the relational setting (logic programming) is critical to this

We will illustrate these ideas next through some simple examples

Representing First-Order Formulas

The following λProlog signature defines a representation

```
kind  term, form   type.
type  ff, tt       form.
type  &&, !!, ==>  form -> form -> form.
infixl &&          5.
infixl !!          4.
infixr ==>         3.
type  all, some    (term -> form) -> form.
type   p   term -> form.
type   f   term -> term.
type   a   term.
```

The representation of the formula $\forall x\ \exists y\ ((p(f(x)) \supset p(f(y))))$:

```
(all x\ some y\ ((p (f x)) ==> (p (f y))))
```

Note that equivalence under renaming, analysis of formula structure, etc, is immediate from the representation

Recursion Over Formula Structure

Recursion over propositional structure is straightforward

```
type is_term  term -> o.
type atomic, quant_free  form -> o.
is_term a.
is_term (f T) :- is_term T.
atomic (p T) :- is_term T.
quant_free F :- atomic F.
(quant_free ff) & (quant_free tt).
(quant_free (F1 && F2)) &
(quant_free (F1 !! F2)) &
(quant_free (F1 ==> F2)) :-
      quant_free F1, quant_free F2.
```

However, how about recursion over quantifier structure?

The issue: argument of `all` and `some` have function type

More precisely, we need a means for recursion over binding structure

Recursion over Binding Structure

The approach used in λProlog

- Use GENERIC to introduce a new constant to lower the type
- Use AUGMENT to assign properties to it before continuing the recursion

For example, consider recognizing *fohc* goal formulas

```
type  is_goal  form -> o.
is_goal F :- atomic F.
is_goal tt.
is_goal (F1 && F2) :- is_goal F1, is_goal F2.
is_goal (F1 !! F2) :- is_goal F1, is_goal F2.
(is_goal (some P)) &
(is_goal (all P)) :-
       pi x\ (term x => (is_goal (P x))).
```

Notice also the use of β-conversion to realize substitution

Is GENERIC Really Needed for the Recursion?

Can a designated constant `dummy` be used for lowering the type?

For example, how about

```
is_goal (all P) :- is_goal (P dummy).
```

Two reasons why this is not a great idea

- The logical treatment provides advantages in reasoning

  For example, we get the following kind of property "for free" from the meta-theory of the specification logic

  > *A substitution instance of a goal* `(all P)` *is a goal*

- the *ad hoc* device just gets it wrong when the recursion involves synthesis as well

An Example Involving Synthesis of Structure

Consider converting formulas in classical logic to prenex normal form (pnf), i.e. to a form

$$\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n M$$

where $\mathcal{Q}_i$ is either $\forall$ or $\exists$ and M is quantifier free

The conversion would use rules such as

> *The pnf of* $\forall x\ C$ *is* $\forall x\ B$ *if the pnf of* C *is* B

As a λProlog clause using the `dummy` constant

```
pnf (all C) (all B) :- pnf (C dummy) (B dummy).
```

But now consider the query

```
?- pnf (all x\ (p x)) F.
```


Mobility of Binders

Side conditions often involve the movement of binders from object-level to proof-level

In *hohh*, this mobility is paralleled by one from terms to formulas to derivations

For example, consider a sequent calculus theorem prover

```
type  proves  (list form) -> form -> o.
proves Gamma tt.
proves Gamma F :- member F Gamma.
proves Gamma (F1 && F2) :-
        proves Gamma F1, proves Gamma F2.
proves Gamma (F1 ==> F2) :- proves (F1::Gamma) F2.
 ...
proves Gamma (some P) :- proves Gamma (P T).
proves Gamma (all P) :- pi x\ proves Gamma (P x).
```

The dynamic growth of signatures and restrictions in instantiations realizes derivation level binding

Representing Typed Lambda Terms

The following λProlog signature defines a representation for types and terms

```
kind  ty, tm  type.

type   int   ty.
type   -->   ty -> ty -> ty.
infix  -->   3.

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

Thus, $(\lambda x : \mathit{int} \Rightarrow \lambda y : \mathit{int} \rightarrow \mathit{int} \Rightarrow y\ x)$ would be represented as

```
(abs int (x\
   (abs (int --> int) (y\ (app y x)))))
```

Note: meta-language types *do not* reflect object-language ones

The preferred approach captures such properties in relations instead

Encoding Evaluation and Typing

The following λProlog program constitutes a specification of call-by-name evaluation and typing

```
type  eval  tm -> tm -> o.
type  of    tm -> ty -> o.

eval (abs T R) (abs T R).
eval (app T1 T2) V :-
      eval T1 (abs R), eval (R T2) V.

of (app T1 T2) Ty2 :-
   of T1 (Ty1 --> Ty2), of T2 Ty1.
of (abs Ty1 R) (Ty1 --> Ty2) :-
   pi x\ (of x Ty1) => (of (R x) Ty2).
```

Notice the use of β-conversion to realize substitution and the treatment of recursion

The latter will give us substitution theorems for free in reasoning

The L_λ Sub-Language of *hohh*

Quantifiers can be essentially universal or existential:

- Goals appear positively and program clauses negatively
- Moving to the left of an implication flips polarity
- Moving to a negative context changes existentials to universals and vice versa

The paradigms discussed almost always yield λProlog programs satisfying the following restriction:

> *Essentially existentially quantified variables appear applied only to distinct λ-bound variables or variables essentially universally bound within their scope*

For example, consider

```
pnf (all C) (all B) :- pi x\ pnf (C x) (B x).
```

The (dynamic) L_λ language is λProlog when it satisfies this property (dynamically)

Interesting Properties of L_λ

- Equations that arise in the L_λ setting have unique solutions up to renaming; e.g. contrast the following:

  ```
  ?- pi g\ pi a\ sigma F\ (F a) = (g a a).
  ?- pi g\ sigma F\ pi a\ (F a) = (g a a).
  ?- pi g\ sigma F\ pi a\ (G a a) = (g a a).
  ```

- β-redexes that arise from substitution for essential existential variables have a benign structure

  Specifically, contracting them does not create new redexes

  This style of redexes are called β_0-redexes

- Substitution is one case that violates this restriction, e.g.

  ```
  proves Gamma (some P) :- proves Gamma (P T).
  ```

  However, here the restriction will typically be met dynamically