Meta-Programming in λProlog

UKALP meeting, Edinburgh, 10 April 1991

Dale Miller

dmi@lfcs.ed.ac.uk

University of Edinburgh and

University of Pennsylvania

Abstract

Meta-programming generally requires the manipulation of data objects that contain internal abstractions. For examples, formulas contain quantification and programs contain parameters and local scopes. First-order terms are not well suited for implementing such structures since the central notions of scope, substitution, and bound and free variable occurrences are not directly supported and must be implemented by a programmer. Such implementations are often difficult to get correct and seldom form a clean interface with other parts of a larger system.

Since λProlog replaces first-order terms with λ-terms it offers a new approach to meta-programming. In this tutorial, I will present the basic principles of λProlog that make it suitable for meta-programming. Several examples from theorem proving and program transformation will be presented. Familiarity with λProlog will not be assumed.

Overview

Part I: Requirements of Abstract Syntax

- Review differences between concrete and abstract syntax.
- Motivate and define a new abstract syntax.

Part II: Logic Programming Language Design

- Describe a logic programming language that incorporates this abstract syntax.

Part III: Example Meta-Programs

- Horn clause interpreter
- Prenex normal form
- Untyped λ-calculus, type inference, and λ-conversion

Part I: Requirements of Abstract Syntax

Review of Concrete Syntax

Implementation

Strings, text (arrays or lists of characters)

Access

Parsers, editors

Good points

Readable, publishable

Simple computational models for implementation (arrays, iteration)

Bad points

Contains too much information not important for many manipulations:

- white space, infix/prefix notation, key words

Important information is not represented explicitly

- recursive structure
- function–argument relationship
- term–subterm relationship

Review of Abstract Syntax

Implementation

first-order terms, parse trees

Access

car/cdr/cons (Lisp)

first-order unification (Prolog) or matching (ML)

Good points

Recursive structure is immediate: recursion over syntax is easy to specify.

Term–subterm relationship is identified with tree-subtree relationship.

Algebra provides a model for many operations on syntax.

Bad points

Requires higher-level language support: pointers, linked lists, garbage collection, structure sharing.

Notions of scope, abstraction, substitution, and free and bound variables occurrences are not supported.

Immediate Notions Regarding Abstractions Are Not Immediate Using First-Order Terms

Bound variables are, like constants, global.

Thus, concepts like *free* and *bound variables occurrences* are derivative notions.

Although *alphabetic variants* generally denote the same intended object, the correct choice of such variants is unfortunately very often important.

Substitution is generally difficult to implement correctly.

An implementation of substitution for one data structure, say first-order formulas, will not work for another, say functional programs.

Computer Systems That Use a Different Approach to Syntax

Mentor (Huet & Lang): second-order matching.

Isabelle (Paulson): fragment of intuitionistic logic with quantification at higher-order types.

λProlog (Miller, Nadathur, Pfenning): a larger fragment.

Elf (Pfenning): an implementation of LF in a style similar to λProlog.

All these systems use first-order terms modulo the equations of α, β, and η-conversion and, therefore, employ aspects of "higher-order" unification.

All but the first permit contexts (signatures) to be dynamic (resembling stacks).

Structure of First-Order Terms

$$\Sigma = \{a : i, \quad b : i, \quad f : i \to i, \quad g : i \to i \to i\}$$

$$\frac{\Sigma \vdash X : i}{\Sigma \vdash f\ X : i} \qquad \frac{\Sigma \vdash X : i \qquad \Sigma \vdash Y : i}{\Sigma \vdash g\ X\ Y : i}$$

$$\frac{}{\Sigma \vdash a : i} \qquad \frac{}{\Sigma \vdash b : i}$$

$$\frac{\dfrac{\dfrac{}{\Sigma \vdash a : i}}{\Sigma \vdash f\ a : i} \qquad \dfrac{}{\Sigma \vdash b : i}}{\Sigma \vdash g\ (f\ a)\ b : i}$$

Structure of λ-Terms

$$\Sigma' = \Sigma \cup \{h : (i \to i) \to i\}$$

$$\frac{\Gamma \vdash U : i \to i}{\Gamma \vdash h\, U : i} \qquad \frac{\Gamma,\ x : i \vdash V : i}{\Gamma \vdash \lambda x.V : i \to i}$$

provided that Γ is an extension of Σ' and x is not in Γ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Sigma',\ x : i \vdash x : i} \quad \dfrac{\overline{\Sigma',\ x : i \vdash x : i}}{\Sigma',\ x : i \vdash f\ x : i}}{\Sigma',\ x : i \vdash g\ x\ (f\ x) : i}}{\Sigma' \vdash \lambda x.g\ x\ (f\ x) : i \to i}}{\Sigma' \vdash h\ (\lambda x.g\ x\ (f\ x)) : i}}{\Sigma' \vdash f\ (h\ (\lambda x.g\ x\ (f\ x))) : i}$$

Designing a New Notion of Abstract Syntax

First: Recursion over terms with abstraction requires signatures (contexts) to be dynamically augmented.

Second: Equality of terms is (at least) α-conversion.

Since terms are not freely generated, simple destructuring is not a sensible operation.

$$\lambda x(fxx) \qquad\qquad \lambda y(fyy)$$

$$x \quad (fxx) \qquad\qquad y \quad (fyy)$$

This, of course, suggests unification modulo α-conversion.

Unification Modulo β_0-Conversion

$$\forall : (i \to b) \to b \qquad r : i \to b$$
$$\wedge : b \to b \to b \qquad s : i \to b$$
$$\supset : b \to b \to b \qquad t : b$$

$$\forall \lambda x (P \wedge Q) = \forall \lambda y ((ry \supset sy) \wedge t)$$

This pair has no unifiers (modulo α-conversion).

$$\forall \lambda x (Px \wedge Q) = \forall \lambda y ((ry \supset sy) \wedge t)$$

This pair has one unifier:

$$\{P \mapsto \lambda w (rw \supset sw), Q \mapsto t\}$$

provided a wee bit of β-conversion is permitted.

$$\forall \lambda x ([\lambda w (rw \supset sw) x] \wedge t) = \forall \lambda y ((ry \supset sy) \wedge t)$$

$$(\lambda x.B)x = B \qquad \beta_0\text{-conversion}$$

Some Matching Examples

Logic variables (meta-variables) can be applied only to distinct, λ-bound variables.

$$a : i \qquad f : i \rightarrow i \qquad g : i \rightarrow i \rightarrow i$$

(1) $\lambda x \lambda y(f(Hx))$	$\lambda u \lambda v(f(fu))$
(2) $\lambda x \lambda y(f(Hx))$	$\lambda u \lambda v(f(fv))$
(3) $\lambda x \lambda y(g(Hyx)(f(Lx)))$	$\lambda u \lambda v(gu(fu))$
(4) $\lambda x \lambda y(g(Hx)(Lx))$	$\lambda u \lambda v(g(gau)(guu))$

(1)	$H \mapsto \lambda w(fw)$	
(2)	match failure	
(3)	$H \mapsto \lambda y \lambda x.x$	$L \mapsto \lambda x.x$
(4)	$H \mapsto \lambda x.(gax)$	$L \mapsto \lambda x.(gxx)$

Restriction on Functional Logic Variables

$$fun\ F\ y\ z = t$$

$$\Sigma\ \vdash \ldots \forall x \ldots \exists F \ldots \forall y \ldots \forall z \ldots [\ \ldots\ F\ y\ z = t\ \ \ldots\ ]$$

$$F \mapsto \lambda y \lambda z.t$$

Under β_0 the λ-expression $\lambda x.B$ has a very weak functional interpretation:

- $\lambda x.B$ takes an increment of a signature to a term over the incremented signature.

Properties of β_0-Unification

Such unification is decidable and most general unifiers exist if unifiers exist.

η-conversion can be added and these properties persist. (α-conversion is assumed)

β_0-unification appears to be the simpliest extension to first-order unification that "respects" bound variables.

β_0-unification does not require type information to determine unifiers or the possibility of unifiers.

$\beta\eta$-unification of simply typed λ-terms (sometimes called "higher-order" unification) can be encoded directly as logic programming using only $\beta_0\eta$-unification.

When functional variables are restricted, β is conservative over β_0.

Higher-Order Abstract Syntax

Implementation

α-equivalence classes of $\beta\eta$-normal λ-terms of simple types

Access

β_0-unification (L_λ) or matching (ML_λ)

Good points

Bound variable names are inaccessible so many technical problems regarding them disappear.

Substitution is easy to support for every data structure containing abstracted variables.

Semantics should be provided by proof theory, logical relations, and Kripke models.

Bad points

Requires higher-level support: dynamic contexts, extended first-order unification, and a richer notion of equality.
No robust, well-defined, and available programming language supports this notion of syntax.

Part II: Logic Programming Language Design
Sublanguages of λProlog

`hohh`

hh^ω Elf

L_λ

`hohc` `fohh`

`fohc`

`ho`	higher-order: predicate and function quantification
`fo`	first-order
`hc`	Horn clauses
`hh`	hereditary Harrop formulas
hh^ω	`hohh` without predicate quantification
L_λ	hh^ω without full β-conversion

Higher-Order Hereditary Harrop Formulas

`hohh` was an attempt to find a very rich logic that supported a "goal-directed" interpretation. λProlog has been designed on top of this language.

Various aspects of `hohh` have been used to understand the following with a logic programming setting.

- higher-order programming
- modules, abstract data types
- hypothetical reasoning
- meta-programming

For particular tasks, weaker languages might supply a tighter fit. For meta-programming, L_λ is a very tight fit (maybe too tight).

Some λProlog Syntax

```
kind  list    type -> type.

type  nil     list A.
type  '::'    A -> list A -> list A.
type  memb    A -> list A -> o.

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

```
kind i         type.

type sterile  i -> o.
type bug       i -> o.
type in        i -> i -> o.
type dead      i -> o.

sterile J :- pi b\((bug b, in b J) => dead b).
```

$$\forall J(\forall b((bug\ b \wedge in\ b\ J) \supset dead\ b) \supset sterile\ J).$$

Interpreting `=>` and `pi` in Goals

Use the syntax

```
K ; P ?- G.
```

to mean "attempt a proof of `G` from signature `K` and program `P`."

To prove an implication, add the hypothesis to the program and prove the conclusion:

`K ; P ?- D => G.` reduces to
`K ; P, D ?- G.`

To prove a universal quantifier, pick a new constant and prove that instance of the quantified goal:

`K ; P ?- pi x\ G.` reduces to
`K, c ; P ?- G [c/x].`

Enforcing the Scope of Constants

When reducing

`K ; P ?-  pi x\ G` to `K, c ; P ?- G [c/x]`,

all currently free, logic variables of `P` and `G` must be restricted so that they are not instantiated with a term containing the scoped constant `c`.

```
 ... ?- pi c\(append (1 :: 2 :: nil) c K).
```

requires the unification `K == (1 :: 2 :: c)`, which must fail.

```
 ... ?- pi c\(append (1 :: 2 :: nil) c (H c)).
```

requires the unification `(H c) == (1 :: 2 :: c)`. This has two possible unifiers

```
H == w\(1 :: 2 :: c)
H == w\(1 :: 2 :: w)
```

of which only the second is permitted.

The Sterile Jar Problem

```
sterile Y :- pi X\(bug X=> in X Y=> dead X).
dead X    :- heated Y, in X Y, bug X.
heated j.


        ?- sterile j
        ?- pi X\(bug X => in X j => dead X)
        ?- bug b => in b j => dead b
bug b   ?- (in b j) => (dead b)
in b j ?- dead b
        ?- heated j, in b j, bug b
        ?- heated j
        ?- in b j
        ?- bug b
```

Meta-Level Properties of `=>` and `pi`

If `M` is both a goal formula and a definite clause, then

if `K ; P ?- M` and `K ; P ?- M => G`
then `K ; P ?- G`.

Similarly, if `K ; P ?- pi x\G` and `t` is some (K-)term, then

`K ; P ?- G [t/x].`

These results follow from the fact that the interpretation given for the logical connectives is sound and complete for intuitionistic logic and that intuitionistic logic has the *cut-elimination* property. For example, if it is provable that `g` is a bug in jar `j`, then it is provable that `g` is dead.

The L_λ-Restriction in λProlog

Notice the equivalences

$$\lambda x.t = \lambda x.s \quad \text{if and only if} \forall x.t = s$$

A functional variable F that can become a logic variable must have occurrence only of the form $(F x_1 \ldots x_n)$ where $x_1 \ldots x_n$ are distinct variables that are either

- λ-bound, or
- are universally bound (at the goal level) in the scope of the binding occurrence of F.

$$\forall_{i \to j} x \forall_i y (p\ (x\ y) \supset p\ (f\ y))$$

is an example of both a goal and program clause for `hohh`; it is only a legal goal in L_λ. As a clause of L_λ, it has a subterm occurrence $(x\ y)$ where both x and y can become logic variables.
First-order Horn clauses are both goals and clauses in L_λ.

L_λ and Abstract Syntax

We shall now argue that L_λ directly supports much of higher-order abstract syntax.

It is possible to weaken L_λ in the following two ways and still maintain this support:

- Remove meta-level typing. β_0-unification can be done without types.
- Remove implications (`=>`) in goals. Hypotheses can be passed around as arguments to predicates. Such a reduction is rather unpleasant, however, and might be best left to a compiler.

It does not seem possible to remove universal quantification or simplify β_0-unification to be simply first-order unification.

Part III: Example Meta-Programs

The Signature of a First-Order Object-Logic

```
kind  term   type.
kind  form   type.

type  all    (term -> form) -> form.
type  some   (term -> form) -> form.
type  and    form -> form -> form.
type  imp    form -> form -> form.

type  a     term.
type  f     term -> term.
type  g     term -> term -> term.
type  p     term -> form.
type  q     term -> term -> form.
```

A Few Very Simple Programs

```
type term   term -> o.
type atom   form -> o.

term a.
term (f X)   :- term X.
term (g X Y) :- term X, term Y.
atom (p X)   :- term X.
atom (q X Y) :- term X, term Y.

type    quant_free      form -> o.

quant_free A :- atom A.
quant_free (and B C) :-
   quant_free B, quant_free C.
quant_free (imp B C) :-
    quant_free B, quant_free C.
```

Recognizing Object-Level Horn Clauses

```
type  hornc  form -> o.
type  conj   form -> o.

hornc (all C) :- pi x\(term x => hornc (C x)).
hornc (imp G A) :- atom A, conj G.
hornc A :- atom A.

conj (and B C) :- conj B, conj C.
conj A :- atom A.


?- hornc (all u\(all v\(imp (p u)
                        (and (q v a) (q a u)))))

{C = u\(all v\(imp (p u)(and (q v a)(q a u))))}

term d ?- hornc (all v\(imp (p d)
                        (and (q v a) (q a d))))
```

Implementing Object-Level Equality

```
type   copytm    term -> term -> o.
type   copyfm    form -> form -> o.

copytm a a.
copytm (f X) (f U) :- copytm X U.
copytm (g X Y) (g U V) :-
                            copytm X U, copytm Y V.

copyfm (p X) (p U) :- copytm X U.
copyfm (q X Y) (q U V) :-
                            copytm X U, copytm Y V.
copyfm (and X Y) (and U V) :-
                            copyfm X U, copyfm Y V.
copyfm (imp X Y) (imp U V) :-
                            copyfm X U, copyfm Y V.
copyfm (all X) (all U) :-
 pi y\(pi z\(copytm y z => copyfm (X y)(U z))).
copyfm (some X) (some U) s:-
 pi y\(pi z\(copytm y z => copyfm (X y)(U z))).
```

$$[\![t, s : \texttt{term}]\!] = \texttt{copytm}\ t\ s$$

$$[\![t, s : \texttt{form}]\!] = \texttt{copyfm}\ t\ s$$

$$[\![t, s : \tau\ \texttt{->}\ \sigma]\!] = \forall x \forall y ([\![x, y : \tau]\!] \supset [\![t\ x, s\ y : \sigma]\!])$$

Implementing Object-Level Substitution

```
type subst  (term -> form) -> term -> form -> o.
subst M T N :-
      pi c\(copytm c T => copyfm (M c) N).
```

Here, the first argument of `subst` is an abstraction over formulas. Compare this to the somewhat simpler specification:

```
subst M T (M T).
```

```
type uni_instan  form -> term -> form -> o.
uni_instan (all B) T C :- subst B T C.
```

Using meta-level β-conversion:

```
uni_instan (all B) T (B T).
```

Several Additional Examples

The following programs make use of meta-level β-conversion to do substitution.

```
type double (term -> term) -> term -> term -> o.
double F X (F (F X)).

type mapfun (term -> term) ->
            term list -> term list -> o.

mapfun F nil nil.
mapfun F (cons X L) (cons (F X) K) :-
   mapfun F L K.
```

To make substitution explicit, write instead:

```
type substterm  (term -> term) ->
                 term -> term -> o.
substterm M T N :-
   pi c\(copytm c T => copytm (M c) N).

double F X S :-
   substterm F X T, substterm F T S.
mapfun F (cons X L) (cons T K) :-
   substterm F X T, mapfun F L K.
```

Reversing Substitutions

```
subst F a (g a a)
```

This query yields four answer substitutions for `F`:

```
w\(g w w)   w\(g w a)   w\(g a w)   w\(g a a).
```

```
copytm a a.
copytm (g X Y) (g U V) :-
                         copytm X U, copytm Y V.
```

```
?- substterm F a (g a a).
?- pi c\(copytm c a => copytm (F c) (g a a)).
copytm c a.  ?- copytm (F c) (g a a).
     {F c = (g (F1 c) (F2 c))}
copytm c a ?- copytm (F1 c) a, copytm (F2 c) a.
copytm c a ?- copytm (F1 c) a.
     {F1 c = a}   or   {F2 c = a}
```

Interpreting Object-Level Horn Clauses

```
type  interp    list form -> form -> o.
type  instan    form -> form -> o.
type  backchain list form -> form -> form -> o.

interp Cs (and B C) :- interp Cs B, interp Cs C.
interp Cs A :- atom A, memb D Cs,
                instan D E, backchain Cs E A.

instan (all A) B :-
           pi x\(copytm x T => instan (A x) B).
instan B C :- quant_free B, copyfm B C.

backchain Cs A A.
backchain Cs (imp G A) A :- interp Cs G.
```

Computing Prenex Normal Forms

```
?- prenex (and (all x\(q x x))
               (all z\(all y\(q z y)))) P.

all z\(all y\(and (q z z) (q z y)))
all x\(all z\(all y\(and (q x x) (q z y))))
all z\(all x\(and (q x x) (q z x)))
all z\(all x\(all y\(and (q x x) (q z y))))
all z\(all y\(all x\(and (q x x) (q z y))))

type    prenex       form -> form -> o.
type    merge        form -> form -> o.

prenex B B :- atom B.
prenex (and B C) D :-
   prenex B U, prenex C V, merge (and U V) D.
prenex (imp B C) D :-
   prenex B U, prenex C V, merge (imp U V) D.
prenex (all B)  (all D) :-
   pi x\(term x => prenex (B x) (D x)).
prenex (some B) (some D) :-
   pi x\(term x => prenex (B x) (D x)).
```

```
merge (and (all B) (all C)) (all D) :-
  pi x\(term x => merge (and (B x)(C x))(D x)).
merge (and (all B) C) (all D) :-
  pi x\(term x => merge (and (B x) C)(D x)).
merge (and B (all C)) (all D) :-
  pi x\(term x => merge (and B (C x))(D x)).
merge (and (some B) C) (some D) :-
  pi x\(term x => merge (and (B x) C)(D x)).
merge (and B (some C)) (some D) :-
  pi x\(term x => merge (and B (C x))(D x)).

merge (imp (all B) (some C)) (some D) :-
   pi x\(term x => merge (imp (B x)(C x))(D x)).
merge (imp (all B) C) (some D) :-
   pi x\(term x => merge (imp (B x) C) (D x)).
merge (imp B (some C)) (some D) :-
   pi x\(term x => merge (imp B (C x)) (D x)).
merge (imp (some B) C) (all D) :-
   pi x\(term x => merge (imp (B x) C) (D x)).
merge (imp B (all C)) (all D) :-
   pi x\(term x => merge (imp B (C x)) (D x)).

merge B B :- quant_free B.
```

The Untyped λ-Calculus

```
kind tm type.

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

type copy  tm -> tm -> o.
type subst (tm -> tm) -> tm -> tm -> o.

copy (abs M) (abs N) :-
  pi x\(pi y\(copy x y => copy (M x) (N y))).
copy (app M N) (app P Q) :- copy M P, copy N Q.

subst M N P :- pi x\(copy x N => copy (M x) P).

bnorm (abs M) :- pi x\(head x => bnorm (M x)).
bnorm H :- hnorm H.

hnorm (app M N) :- hnorm M, bnorm N.
hnorm H :- head H.
```

Head Normal Form and β-Reduction

```
type hnf      tm -> tm -> o.

hnf (abs M) (abs M).
hnf (app M N) P :-
    hnf M (abs R), subst M N Q, hnf Q P.

type redex   tm -> tm -> o.
type red1    tm -> tm -> o.
type reduce  tm -> tm -> o.

redex (abs x\(app M x)) M.
redex (app (abs M) N) P :- subst M N P.

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\(copy x x => red1 (M x) (N x)).

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

Simple Type Checking

```
kind ty type.

type arr    ty -> ty -> ty.

type typeof tm -> ty -> o.

typeof (app M N) A :-
    typeof M (arr B A), typeof N B.

typeof (abs M) (arr A B) :-
    pi x\(typeof x A => typeof (M x) B).
```