The Metalanguage λProlog and Its Implementation

Gopalan Nadathur

Computer Science Department

University of Minnesota

(currently visiting INRIA and LIX)

The Role of Metalanguages

Many computational tasks involve the manipulation of linguistic objects:

- prototyping programming languages
- implementing compilers and program development systems
- manipulating mathematical expressions
- realizing (interactive) proof systems

Emerging applications involve the *integration* of many of these computations.

Can programming language support be provided for such activities?

Metalanguages and Logic Programming

Prolog-like languages contain two features important to symbolic computation:

- First-order terms generalize traditional abstract syntax

$$B \wedge C$$

$$\Downarrow$$

$$\mathrm{and}(\widehat{B},\widehat{C})$$

- Horn clauses naturally translate structural operational semantics rules

$$\frac{\Gamma \vdash B \qquad \Gamma \vdash C}{\Gamma \vdash B \wedge C}$$

$$\Downarrow$$

prove(Gamma,and(B,C)) :-
  prove(Gamma,B), prove(Gamma,C).

An Inadequacy of Traditional Abstract Syntax

Binding notions are not supported in the syntax representation.

The 'first-order' rendition of the formula $\forall x P(x)$:

$$all(x, \widehat{P(x)})$$

Respecting scope issues becomes the programmer's burden with such a representation.

For example consider instantiating the outer quantifier in

$$all(x,\ all(y,\ Q(x,y)))$$

with the term $f(y)$.

In general, 'proper' substitution can be a complex operation to capture correctly.

Higher-Order Treatment of Syntax

Scoping notions arise in many symbolic structures:

- Quantified formulas in non-classical logic
- Side conditions in inference rules
- Proofs for implicational and universal statements
- Binding and bound variable occurrences in programs

A common core of binding related operations apply to all these situations.

A uniform treatment of these aspects can be provided by incorporating binding into syntax representation.

Structure of the Rest of the Talk

- Higher-Order Abstract Syntax in λProlog
- Issues in Realizing the Metalanguage Features
- Structure of the *Teyjus* Implementation
- Concluding Remarks

Higher-Order Abstract Syntax in λProlog

Richer view of object language syntax is supported through the following new features:

- Using lambda terms as data structures
- Incorporating an understanding of lambda conversion into unification
- Allowing for *GENERIC* goals

$$\forall x G$$

"Solve G after replacing x with a new constant"

- Allowing for *AUGMENT* goals

$$D \texttt{ => } G$$

"Add D to program before solving G"

Representing the Lambda Calculus

Term formation through application and abstraction has to be captured.

The HOAS approach:

- Use constructors to distinguish between object language application and abstraction
- Use λProlog abstraction to represent object language binding

Thus

$$\overline{(M\ \ N)} \quad \longrightarrow \quad (\mathit{app}\ \overline{M}\ \overline{N})$$

$$\overline{(\text{lambda (x) M})} \quad \longrightarrow \quad (\mathit{abs}\ \lambda x\ \overline{M})$$

Representing Functional Programs

- Introduce new constructors to represent programming language primitives
- Utilize λProlog abstraction to translate object language binding
- Use syntactic de-sugaring and the basic translation scheme to render programs into terms

An Example

*fact m n = if (m = 0) then n else (fact (m - 1) (m * n))*

$\Updownarrow$

fact =
(fixpt (f)
(lambda (m) (lambda (n)
*(if (m = 0) then n else (f (m - 1) (m * n))))))*

$\downarrow$

fact =
(fix λf
(abs λm(abs λn
(cond (eq m 0) n
(app (app f (minus m 1)) (times m n))))))

Usefulness of HOAS Representation

Primitives in λProlog provide direct support for logical operations on "program terms"

- Lambda conversion rules build in an understanding of binding structure and substitution
- Higher-order unification is a useful tool for examining program structure
- Scoping devices support recursion over binding structure

Pattern Recognition through Unification

Program terms may contain substitutible variables.

However, substitutions for these variables must respect scope restrictions.

For instance, the 'pattern'

$$(abs\ \lambda x(abs\ \lambda y(C\ x)))$$

can match with

$$(abs\ \lambda x(abs\ \lambda y(less\ x\ 0)))$$

but not with

$$(abs\ \lambda x(abs\ \lambda y(less\ x\ y)))$$

Thus, unification provides a sophisticated means for dependency analyses.

Recognizing (Binary) Tail Recursive Functions

Consider the following "template"

(fix λf (abs λx (abs λy
(cond (C x y) (H1 x y) (app (app f (H2 x y)) (H3 x y))))))

Notice that *C*, *H1*, *H2* and *H3* *cannot* be instantiated so as to depend on *f*, *x* or *y*.

Thus, this term recognizes only those recursive two argument 'conditional' programs in which

- there is *no* recursive call in the condition or *then* branch, and
- the value returned in the *else* branch is *completely* determined by recursive call.

Such programs must be tail recursive.

Limitations of Template Matching

Unfortunately, templates alone have limited applicability.

For example, what if

- the recursive call is in the *then* branch of conditional?
- there are embedded conditionals?

Thus, our template will not recognize tail recursiveness of the following program:

gcd x y =
  if (x = 1) then 1
  else if (x < y) then (gcd y x)
    else if (x = y) then x else (gcd (x - y) y)

Worse still, there is no *finite* set of templates covering *all* mentioned cases and recognizing *only* tail recursive programs.

Recursion over Conditional Structure

However, a satisfactory *recursive* description of such program terms *can be* provided:

- A program with *no* recursive calls

  $tr\ (fix\ \lambda f(abs\ \lambda x(abs\ \lambda y(H\ x\ y)))).$

- A program comprising *only* a recursive call

  $tr\ (fix\ \lambda f(abs\ \lambda x(abs\ \lambda y\ (app\ (app\ f\ (H\ x\ y))\ (G\ x\ y))))).$

- A conditional program with *no* recursion in the test *and* with 'tail recursive' *then* and *else* branches

$$
\begin{array}{l}
tr\ (fix\ \lambda f(abs\ \lambda x(abs\ \lambda y \\
\quad (cond\ (C\ x\ y)\ (H1\ f\ x\ y)\ (H2\ f\ x\ y))))) \texttt{ :-} \\
\qquad tr\ (fix\ \lambda f(abs\ \lambda x(abs\ \lambda y(H1\ f\ x\ y)))), \\
\qquad tr\ (fix\ \lambda f(abs\ \lambda x(abs\ \lambda y(H2\ f\ x\ y)))).
\end{array}
$$

Recursion Over Binding Structure

Recognizing tail-recursiveness of *arbitrary* arity functions requires an explicit recursion over *binding* structure:

- Given an expression of the form

  $$(fix\ (\lambda f\ F))$$

  analyze F after replacing f with a new constant whose occurrences must be restricted.

- Given an expression of the form

  $$(abs\ (\lambda x\ R))$$

  analyze R after replacing x with a new constant whose usage can be arbitrary.

- Check the eventual "first-order" structure for satisfaction of usage constraints.

Can be realized using *GENERIC*, *AUGMENT* and application.

A Recognizer for Tail Recursive Functions

Assume that *(term T)* succeeds just in case T is a 'program term.'

tr (fix M) **:-** $\forall f$ *((recfn f)* **=>** *(trfn (M f)))*.

trfn (abs R) **:-** $\forall x$ *((term x)* **=>** *(trfn (R x)))*.
trfn R **:-** *trbody R*.

trbody (cond C M N) **:-** *term C, trbody M, trbody N*.
trbody (app M N) **:-** *trbody M, term N*.
trbody M **:-** *recfn M*.

Representation of Lambda Terms

Lambda terms are being used as *data structures*.

Thus, the representation should satisfy the following criteria:

- Structure should be accessible
- Equality under renaming should be easy to determine
- The operation of β-reduction should be efficiently supported

A Complication: In the context of interest, it may be necessary to look *inside* abstractions.

A Consideration in Beta Reduction

Support for *laziness* in reduction substitutions could be useful:

- Provides the basis for combining structure traversals in reductions
- Actual substitution may sometimes be delayed to a point where it becomes unnecessary

Explicit treatment of substitution is an essential ingredient to realizing such benefits.

Actual Lambda Term Representation

The representation used in *Teyjus* has the following characteristics:

- Utilizes the deBruijn scheme for eliminating (bound variable) names
- Based on an explicit substitution notation called the *suspension* notation
- Uses a demand driven approach to reduction and substitution, thereby interleaving these with comparison operations
- Exploits annotations indicating closedness status of terms
- Implements reduction using a graph-based scheme

Dealing with GENERIC

Idea of instantiating with new constant may be used

However, there is interference with usual treatment of free (existential) variables

Program: $\forall x\, p(x, x)$

Goal: ?- $\forall x\, p(Y, x)$

$$\Big\downarrow \; c/x$$

?- $p(Y, c)$

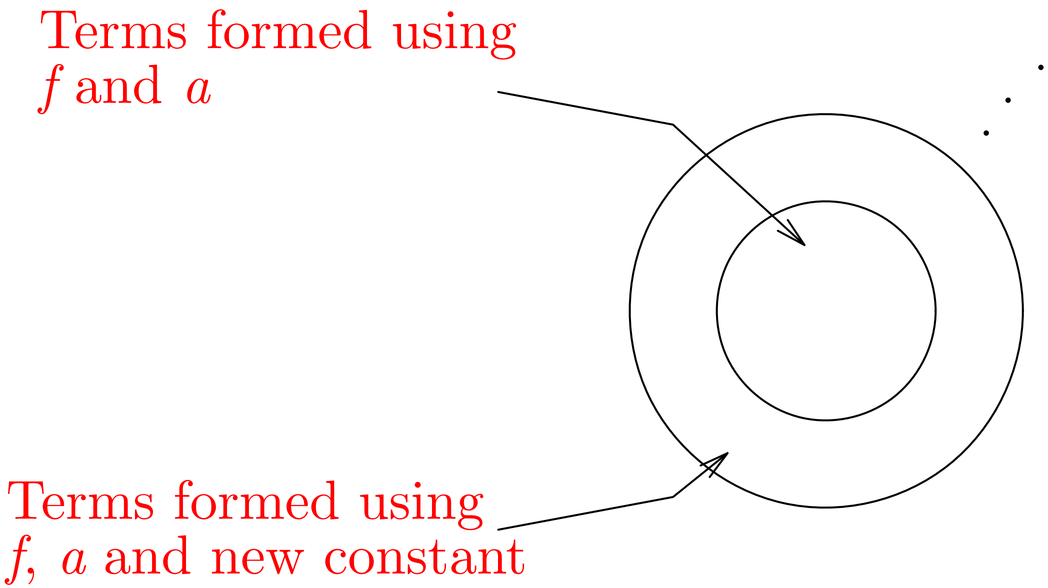
Unification has to be somehow constrained to cause failure in this situation

A Possible Solution

Maintain term universes as hierarchy

Introduce new levels in the hierarchy when processing GENERIC

$$\forall y\; p(a, f(X), y)$$



Label constants to determine 'place' in hierarchy

Label variables to constrain possible instantiations

Details of the Solution

The scheme can be realized as follows:

- Maintain the current highest universe level in a special register
- Translate GENERIC into register increment on entry and decrement on success
- Label constants and variables with register value at creation
- When binding a variable, check also the consistency of labelling

Most actions can be realized though low-level instructions.

Realizing Higher-Order Unification

Multiple most general unifiers may exist.

For example, consider the problem

$$(F\ 1) = (g\ 1\ 1)$$

This problem has four distinct unifiers:

$$F \mapsto \lambda x(g\ x\ x)$$
$$F \mapsto \lambda x(g\ x\ 1)$$
$$F \mapsto \lambda x(g\ 1\ x)$$
$$F \mapsto \lambda x(g\ 1\ 1)$$

An implementation must correspondingly manifest a branching character.

Moreover, it should be able to sometimes suspend unification problems to avoid redundant search.

Treatment of Higher-Order Unification

Our implementation of this operation has the following characteristics:

- Supports compilation of first-order like processing
- Attempts to exploit determinism in unification
- Provides an explicit representation for unification problems that supports sharing
- Has efficient mechanisms for realizing branching in unification

Higher-Order Pattern Unification

Decidability and unicity properties hold when existential variables are applied to distinct variables universally quantified within their scope.

For example,

$$\forall v \exists X \forall u \exists Z \forall w \, ((X \ u) = (v \ (Z \ w)))$$

has the solution

$$X \mapsto \lambda x (v \ (Y \ x))$$

$$Z \mapsto \lambda x (Y \ u)$$

where Y is a new variable existentially quantified at the same level as X.

Generating this substitution involves *pruning* and *raising* steps.

A new algorithm that does these steps on the fly has been developed.

Dealing with AUGMENT

Two new issues arise in a sequential implementation with a central program:

- Incremental programs changes must be modelled

Thus, solving the goal

$$D \texttt{ => } G$$

This involves adding and removing code

- Backtracking behavior requires old programs to be remembered

For example, consider the goal

$$(D_1 \texttt{ => } G_1(X)) \wedge (D_2 \texttt{ => } G_2(X))$$

Both code access and context switching must be efficient.

An Implementation Scheme

An efficient implementation can be realized using the following ideas:

- Represent program via an *implication point record* (IPR) containing
  - access function to new code layer
  - pointer to previous IPR
- Compile *AUGMENT* goals into creation and 'removal' of IPRs
- Maintain a special *program* register pointing to most recent IPR
- At choice point creation, store also the contents of program register

The scheme permits compilation of the antecedents of *AUGMENT* goals.

Bringing it All Together

The *Teyjus* system embodies a solution to all the problems and comprises three parts:

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

The abstract machine has been realized through a software emulator.

The entire system has been implemented in C.

Directions of Ongoing Research

- Improved support for modularity
- Compiled treatment of higher-order pattern unification based language
- Evaluation of choices in the representation of lambda terms
- Modularization of implementation technology

Resources

- The λProlog web page

  `http://www.cse.psu.edu/~dale/lProlog/`

- The *Teyjus* web page

  `http://teyjus.cs.umn.edu/`

- Papers providing the basis for this talk

  `http://www.cs.umn.edu/~gopalan/papers.html`