Optimizing the Runtime Processing of Types in Polymorphic Logic Programming Languages

Gopalan Nadathur[1] and Xiaochu Qi[2]

Digital Technology Center and Dept. of CS&E, University of Minnesota, USA

Dept. of CS&E, University of Minnesota, USA

LPAR 2005 – Montego Bay, Jamaica
December 2, 2005

The Motivation for this Work

- (Polymorphic) types can be useful in logic programming
  - can help catch program errors at compile time
  - essential for higher-order notions
  - polymorphism provides conciseness and flexibility
- Runtime computations over types may be necessary
  - clause based definitions lead to *ad hoc* polymorphism
  - unification may require type information
- Computations over types can be costly
  - types have the structure of first order terms
  - polymorphism leads to unification over types

Question: Can type computations be made redundant by compile time analysis?

Outline of the Talk

- Types and their consequences in λProlog
- Processing model based on higher-order pattern unification
- Simplifying type annotations with constructors
- Eliminating type annotations in predicate definitions
- Concluding remarks

Types in λProlog

λProlog is a higher-order, strongly typed language with a polymorphic typing discipline

```
kind  list  type -> type.

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

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

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

Types and Program Checking

Compiler ensures that all expressions it admits are type correct. For example, consider

```
type sum_list (list int) -> int -> o.

sum_list nil 0.
sum_list (X :: L) N :-
         sum_list L N1, N is N1 + 1.

?- append ("a" :: "b" :: nil) nil L,
   sum_list L N.
```

Compiler will flag an error with this query.

Ad Hoc Polymorphism in λProlog

Clauses defining predicates may be sensitive to type instances:

```
type  print  A -> o.
print (X:int) :-
      {code for printing integer X}.
print (X:string) :-
      {code for printing string X}.
```

Predicates that are defined in terms of *ad hoc* predicates also need to carry types at runtime:

```
type  print_list  (list A) -> o.
print_list nil.
print_list (X :: L) :- print X, print_list L.
```

Here `print_list` must have the type of the list elements available to pass on to *print*.

Types and Higher-Order Unification

In addition to determining unifiability, types can determine the *shapes* of unifiers.

For example, consider the equation

$$(F\ X) = (g\ a)$$

where g has type $i \rightarrow i$.

If F has type $int \rightarrow i$ then there is an mgu:

$$\{\langle F, \lambda x\,(g\ a)\rangle\}.$$

If F has type $i \rightarrow i$ there are two other incomparable unifiers:

$$\{\langle F, \lambda x\, x\rangle, \langle X, (g\ a)\rangle\} \text{ and } \{\langle F, \lambda x\,(g\ x)\rangle, \langle X, a\rangle\}.$$

The *Teyjus* implementation must, as a result, calculate and carry around types with *every* constant and variable.

Higher-Order Pattern Unification (HOPU)

A form of higher-order unification with several pleasing features:

- solves most higher-order unification problems that occur in practice
- sometimes even solves pairs that are left over as constraints by the usual procedure
- has first-order like behaviour over the class it covers completely,
  e.g. is decidable, admits most general unifiers, etc.

Thus, processing in λProlog and other higher-order languages can be oriented around HOPU.

Types and Higher-Order Pattern Unification

Unification can be carried out as the combination of two phases:

- A simplification phase

  Peeling off top-level constants in equations of the form

  $$(c\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_n)$$

  Types of the two constant heads must be matched

- A variable binding phase

  Finding substitutions for solving equations of the form

  $$(F\ u_1\ \ldots\ u_n) = t$$

  Types are irrelevant to this computation

Thus, types are needed only with constants and, that too, only declared ones.

Types and Higher-Order Pattern Unification

Unification can be carried out as the combination of two phases:

- A simplification phase

  Peeling off top-level constants in equations of the form

  $$(c\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_n)$$

  Types of the two constant heads must be matched

- A variable binding phase

  Finding substitutions for solving equations of the form

  $$(F\ u_1\ \ldots\ u_n) = t$$

  Types are irrelevant to this computation

Thus, types are needed only with constants and, that too, only declared ones.

Types and Higher-Order Pattern Unification

Unification can be carried out as the combination of two phases:

- A simplification phase

  Peeling off top-level constants in equations of the form

  $$(c\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_n)$$

  Types of the two constant heads must be matched

- A variable binding phase

  Finding substitutions for solving equations of the form

  $$(F\ u_1\ \ldots\ u_n) = t$$

  Types are irrelevant to this computation

Thus, types are needed only with constants and, that too, only declared ones.

Types and Higher-Order Pattern Unification

Unification can be carried out as the combination of two phases:

- A simplification phase

  Peeling off top-level constants in equations of the form

  $$(c\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_n)$$

  Types of the two constant heads must be matched

- A variable binding phase

  Finding substitutions for solving equations of the form

  $$(F\ u_1\ \ldots\ u_n) = t$$

  Types are irrelevant to this computation

Thus, types are needed only with constants and, that too, only declared ones.

Types and Higher-Order Pattern Unification

Unification can be carried out as the combination of two phases:

- A simplification phase

  Peeling off top-level constants in equations of the form

  $$(c\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_n)$$

  Types of the two constant heads must be matched

- A variable binding phase

  Finding substitutions for solving equations of the form

  $$(F\ u_1\ \ldots\ u_n) = t$$

  Types are irrelevant to this computation

Thus, types are needed only with constants and, that too, only declared ones.

Types and Higher-Order Pattern Unification

Unification can be carried out as the combination of two phases:

- A simplification phase

  Peeling off top-level constants in equations of the form

  $$(c\ t_1\ \ldots\ t_n) = (c\ s_1\ \ldots\ s_n)$$

  Types of the two constant heads must be matched

- A variable binding phase

  Finding substitutions for solving equations of the form

  $$(F\ u_1\ \ldots\ u_n) = t$$

  Types are irrelevant to this computation

Thus, types are needed only with constants and, that too, only declared ones.

Simplifying Type Annotations for Constants

An observation:

> *Every instance of a declared constant must have a type that matches the declared one.*

For example, every occurrence ot `::` must have as type an instance of

```
A -> (list A) -> (list A).
```

Thus, only bindings for the type variables in the 'skeleton' need be stored and compared.

Using this idea, `(1 :: 2 :: nil)` can be encoded as

```
(1 (:: [int]) 2 (:: [int]) (nil [int])).
```

Further Simplifying Constant Types

A further observation:

Unification proceeds outside in and compares only terms with identical types

For example, given the problem

```
(1 :: 2 :: nil) = (X :: L)
```

the context automatically ensures that the second list is of type `(list int)`.

The upshot: bindings for variables that also appear in the target type of the skeleton can be dropped from type annotations.

A special case: no type annotations are needed with *type preserving* constants.

Types and Predicate Constants

Considering the target type does not remove type annotations from clause definitions.

For example, the definition of `append` becomes

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

However the type annotation is redundant even in this case:

- type unification in clause head will always succeed
- behaviour repeats with type passed on to recursive call

Such type annotations can be eliminated by a *usage* analysis.

Determining Redundancy for Predicate Types

Suppose every clause for a predicate p has the form

$$\texttt{p [ty}_1\texttt{,...,ty}_j\texttt{,X,...] ... :- ...}$$

i.e., the annotation in the $(j+1)^{th}$ position is always a variable.

Suppose further that

- X does not appear again in the types list for p,
- X does not appear in the types list for any non-predicate constant in the clause, and
- X appears at most in a redundant type position for a goal in the clause body.

Then the type binding for X does not affect computation.

The last condition requires a (least) fixed-point computation of neededness in the context of recursive definitions.

Eliminating Types with Predicates

Type annotations that are determined to be not “needed” can safely be dropped.

Using this idea, the definition of `append` becomes

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

More generally, *all* type annotations can be removed when

- all constructors are *type preserving*, and
- all clause definitions are *type general*.

Conclusion

- Practical treatment of typed unification in the presence of polymorphism
  - Subsumes earlier approaches that restrict the language
  - Degrades gracefully when conditions are not met
- Based on pattern unification in the higher-order context [Nadathur and Linnell, ICLP'05]
- Underlies a new compiler based implementation of λProlog [Part of X. Qi's doctoral research]