PROSHEE: Developing a Constraint-Based Type Analysis Tool For Prolog

Asad B. Sayeed
asayeed@users.sourceforge.net

December 23, 2004

Abstract

There have been many schemes for type inference for many languages. Logic programming provides its own special challenges for type inference, particular given that procedures are not clearly specified in terms of their incoming and outgoing data—execution of a program binds variables in ways that cannot be predicted in advance, a problem compounded by the nonexplicitly typed and highly polymorphic nature of common logic programming languages such as Prolog. We describe approaches to handling different aspects of this problem using abstract interpretation and set constraint theory, and then we describe the process of development of a simple framework for Prolog type-analysis by harnessing an efficient set-constraint solver called BANSHEE to a version of Prolog (SWI-Prolog). Given that the nonexplicit typing in Prolog can be highly unsafe, we describe a lattice of types that adequately captures much of the normal safe usage of Prolog and describe how this lattice can be implicitly enforced using BANSHEE.

1 Introduction

Type inference and type checking theories and systems have been developed for many languages and applications. It is, however, particularly challenging for languages such as ML and Prolog, where types of variables are not explicitly given in the code. Types in ML, however, are mostly easy to infer using a Hindley-Milner style type inference algorithm. While ML allows ample type polymorphism, it does not easily permit union types: a match statement can only use the constructors of a single type.

Prolog, on the other hand, not only does not have explicit types, it is not strongly typed like ML either. There is no requirement to declare type constructors, even polymorphic ones, in advance. At every step of computation, the programmer can define arbitrary structures, and all variables are assumed from the beginning to be $\bot$, in that they have no type at all. What is worse is that after execution, it is often reasonable to assume that all variables are $\top$, or some form of more restricted union type. There is, however, obvious reasons to

imagine that union types are not always safe; as well, because of the limited approximation of nondeterminism that logic programming provides, it is not always easy to see what sorts of union types would be achieved by different parts of the program.

There have been many attempt to rectify this situation, either by creating variants of Prolog that are explicitly typed [Somogyi et al., 1995], or by developing more complicated type inference algorithms based on existing ones for other languages such as Hindley-Milner [Henglein, 1988]. We discuss one approach using Banshee, a set-constraint solver for static program analysis, that also limits the production of union types by imposing a simple lattice of types onto Prolog.

1.1 Organization of this paper

This paper is organized into several parts. First, we discuss the theoretical and practical background of this project. We discuss logic programming in terms sufficient to understand how the program analysis proposed here would be implemented: the structures involved, and a high-level view of how the interact. Then we discuss a proposal for representing logic programs in general as set constraints representing substitutions at various points in the program. Eventually, we discuss the role of abstract analysis in the type inference of logic programs, illustrating the complexity and versatility of this technique. We then describe Proshee, a system for Prolog type inference based on Banshee, a set constraint solver for program analysis. Finally, we discuss some remaining issues in the implementation and some future avenues for work.

2 Background and related work

2.1 Logic programming

This section discusses logic programming with reference to the most common logic programming environment, Prolog. It attempts to do so in such a way that provides precisely the information required to understand the subsequent sections discussing the design and implementation of the Proshee system. Thus illustrations are kept to a minimum, and the description is spare and high-level.

2.1.1 Structures in the language

There are many languages that have been developed for use in logic programming, such as Mercury and λProlog. More often then not, however, these languages are simply variants of "pure" Prolog, a language very commonly used in artificial intelligence and knowledge engineering research. Prolog is presently defined by a somewhat incomplete ISO specification, and the multitude of implementations such as GNU Prolog, SWI Prolog, and Amzi! Prolog that have been developed all have their own particular variations to make up for the deficiencies in the standard. For the purpose of this paper, however, we will assume that we are using SWI Prolog, in case it makes any difference.

A prolog program consists of a database of clauses. Some of these clauses are built in to the basic Prolog environment and thus their contents are not (intended to be) visible to the programmer, although some interfaces to these clauses are exposed and documented for the programmer's use—this is particularly important for the clauses that produce side-effects, such as printing to the screen. Most of the remainder can actually themselves be implemented by programmers using the basic Prolog execution mechanism, but they are provided for convenience and because the developers of the (SWI) environment have provided complex optimizations for them.

The form of a clause is as follows:

```
head :- body.
```

The head consists of a callable term and the body consists of a comma-separated list of callable terms.

Callable terms are usually of the two following forms:

```
atom
atom(term, term, ...)
```

Atoms are strings that begin with a small letter and contain no spaces; otherwise, they are strings contained in single quotes. Numbers, variables, and lists are terms, but they are not callable terms without being contained in quotes. The atom at the head of a compound term (the second one above) is called a functor, and the remaining terms (which can include other callable terms) are called its arguments. Callable terms can thus be characterized as a functor/arity pair. Functors without arguments are thus considered to have zero arity. A functor/arity pair that characterizes the head of one or more clauses is called a predicate.

Lists are structures represented in a manner similar to that of list processing languages such as Lisp: a binary tree rooted in a null element that is considered to be an empty list.

```
list → [] (the empty list)
list → [term | list] (recusive definition of a list)
```

Variables are terms that consist of a string beginning with a capital letter that is not enclosed in single quotes. A variable is typically considered to be existentially quantified in its clause. Variables cannot be directly used in any sort of computation until they are bound. Binding is dependent on the execution model of Prolog, which we discuss in the next section.

2.1.2 Execution model

The underlying execution procedure for logic programs is often known as Warren's Abstract Machine (WAM). The WAM [Aït-Kaci, 1991] is the most dominant model of logic program execution, and most recent Prolog implementations have been developed using it. However, many of its details are not salient to this discussion, particularly those that pertain to things like backtracking and cuts; so we will focus on those details that are.

A prolog clause is activated when, in the environment, its head is declared to be valid in that it is unifiable with a particular term that is placed on a stack of "goals." Unification,

in short, is pattern matching over a pair of terms where variables in either term are bound to the values that they match in the other term.

Then the body of the clause must be executed by placing each term in the body on the stack as a goal, immediately executing a corresponding clause. When a clause completes (in other words, succeeds), the goal that was used to invoke it is removed from the stack. However, the variable bindings created by the execution of this goal survive for the execution of the next goal in the body.

If a goal unifies with a head of a clause with an empty body (simply a term asserted in the database as depicted in the specification in the previous section), then the goal simply succeeds. No further goals need be proven. This is a "base" or limiting case for logic program execution. This allows us to describe the execution of a Prolog program as a depth-first search for clauses with heads unifiable with goals on the stack but without bodies. When every branch of the search achieves such a clause, the program as a whole is said to succeed.

What happens when a goal on the stack cannot be unified with a clause? This is where Prolog makes use of one of its most important features: backtracking. Prolog attempt to "re-prove" previous goals, generating alternate bindings for variables that may cause the failed goal to succeed. Sometimes an entire clause containing the goal may fail. In which case, the next clause for that predicate is tried. A goal is also said to fail when all of the clauses that belong to the predicate it represents together fail.

Some clauses have empty heads, such as

```
:- f(X).
```

An empty head is unifiable with anything. This means that it unifies with the top of an empty goal stack! The bodies of such clauses are immediately executed. Such clauses are known as queries or top-level goals. A query is required to activate any computation—it is the starting point of the program. The result of computation, from the user's point of view, is a list of bindings for variables in the query.

2.2 Environment constraints for logic programming

In this paper, we seek to provide a method for the type analysis of Prolog using set constraints. In order to do this, we first need to decide on a set constraint representation for Prolog programs in general. Heintze [1992] provides a means for computing a representation. While his approach is mathematically rigorous and detailed, we will extract the parts that are salient to the discussion here.

Before and after every goal in the body of a clause is executed, there is a set of substitutions for variables in that clause: a substitution environment, in other words. The successful execution of a goal adds substitutions to the substitution environment at the program point prior to the goal, producing a new substitution environment after the execution of the goal.

Heintze annotates Prolog programs, therefore, with flow-insensitive program point markers. So for instance, the clause

```
p(f(X, Y)) :- q(X, Y).
```

is annotated by Heintze[1] as

```
p(f(X, Y)) :- 1 q(X, Y). 2
```

In other words, **1** is the program point prior to the execution of `q(X, Y)`, while **2** is the program point after the completion of the body of the clause.

So let us take an example program from him:

```
:- 1 p(W), 2 q(W). 3
p(f(X)). 4
p(g(Y)). 5
q(f(Z)). 6
```

Each program point α corresponds to a substitution environment Ψ^α. Given goal A, $A.\Psi^\alpha$ represents the set of substitutions brought about when A is executed in Ψ^α. From this immediately follow equations from the arrangement of program points and goals.

Heintze provides a uniform, mathematical way to compute these goals for various models of logic program execution. However, the only model in which we are interested is the top-down, left-to-right DFS model described in the previous sections. In which case, the above program is represented by the following equations:

$$\begin{aligned}
\Psi^1 &\supseteq \{\} \\
\Psi^2 &\supseteq \{p(W) \in p(f(X)).\Psi^4\} \\
\Psi^2 &\supseteq \{p(W) \in p(g(X)).\Psi^5\}
\end{aligned}$$

In other words, at **1**, where nothing has happened. There can be no substitutions. At **2**, however, there can be substitutions generated by executing the goal, whose possible completion points are **4** and **5**, both clauses of `p/1`. At **3**, however, we get

$$\begin{aligned}
\Psi^3 &\supseteq \{p(W) \in p(f(X)).\Psi^4, q(W) \in q(f(X)).\Psi^6\} \\
\Psi^3 &\supseteq \{p(W) \in p(f(X)).\Psi^5, q(W) \in q(f(X)).\Psi^6\}
\end{aligned}$$

The two substitutions created by `p(W)` each have to also be matched at program point **2** with a substitution for `q(W)`.

Finally,

$$\begin{aligned}
\Psi^4 &\supseteq \{p(f(X)) \in p(W).\Psi^1\} \\
\Psi^5 &\supseteq \{p(g(Y)) \in p(W).\Psi^1\} \\
\Psi^6 &\supseteq \{q(f(Z)) \in q(W).\Psi^2\}
\end{aligned}$$

Not only are substitution environments affected by the predicates they invoke, as in Ψ^2 and Ψ^3, but the environments immediately prior to invocation also affect them. Hence Ψ^4, Ψ^5, and Ψ^6 contain bindings caused by the calling goals.

[1] We have taken the liberty of changing the style of his labelling, since we find his less obvious for human beings to interpret.

These cannot in themselves be given to a set constraint solver, as they are too exact. To be implemented practically, something must interpret $A.\Psi^{\alpha}$ to mean a specific set of substitutions created by interpreting the goals themselves. In later sections, we thus deal both with the representation of elements in the environment as well as capturing their bindings.

Note that in a logic program, a predicate can obviously be called from multiple points in the program. But Heintze's analysis only gives a single assignment of program points to a particular clause. So in solving these environment constraints, substitution environments must be valid for all execution paths on which a goal may be reached. This means that Heintze's analysis is flow-insensitive.

Lastly, Heintze allows any kind of arbitrary structure to become part of the solution to these substitution environment constraints. This suggests that his analysis computes all possible substitutions of all possible structures at all program points. Our analysis is rather less ambitious than that.

2.3 Abstract interpretation for logic programming

Just like in other areas of program analysis, abstract interpretation has a large literature in the analysis of logic programs. A representative example of this is Lu [2000]. Lu writes precisely about union types, the major consuming issue of logic program type analysis. Lu discusses several proposals for union types and suggests that they are not true set union. For instance, Lu notes that Codish and Lagoon [2000] define a union-approximating operator $\oplus$ that is not true union as it is "distributive over type constructors." For example, according to Lu, *list(atom* $\oplus$ *float) = list(atom)* $\oplus$ *list(float)*, which necessarily means that one cannot have mixed lists of atoms and floats. Lu mentions a number of other examples for which these approximations are inadequate. Lu rectifies this by proposing a special abstract unification operation which computes the correct equivalence classes in a lattice of union types; these classes represent true disjunctions.

Just as Lu is preoccupied by the task of enabling true union types, we are occupied with a much simpler task: preventing them. We will use abstract interpretation as well, in that we will propose below what is effectively a lattice of types in which $\top$ is the only union type for goal argument variables. $\top$, however, represents an undesirable type-conflict.

2.4 Banshee

Banshee is an efficient set constraint solver for program analysis developed in C. It was developed at Berkeley and provided for free to the world.

Banshee enables programmers to develop problem-specific set constraint solvers by simply specifying data constructors relevant to the kind of analysis (in this case, type analysis) that they are trying to do. Banshee processes these specifications and then generates an engine written in C customized to accomodate constraint equations specified in terms of these constructors. The programmer then writes a front end that makes use of the engine for whatever purpose the programmer intends.

Banshee supports set inclusion constraints [Aiken and Wimmers, 1993] as well as term equality constraints. Whether constructors can be used in one or the other depends on their definition. Constructors must be defined as either term constructors or set-compatible constructors in their Banshee specificiation. This limits the kinds of equations in which they can participate.

Unfortunately, Banshee does not appear to be well-documented, and we have been unable to use many of its features as yet for this reason. Nevertheless, we have been able to make use of sufficient features to build a prototype type-inference system for Prolog, which we call Proshee for obvious reasons.

3 Details of the implementation

Much of the implementation of Proshee can be derived from the discussion above, but it is not immediately obvious how the pieces might fit together. In this case, the pieces are the type analysis that is to be imposed on Prolog and the interaction with Banshee that needs to be performed in order to compute the type analysis using a set constraint paradigm. These are discussed in the sections below.

3.1 The Types

Prolog is a language without explicit type declarations or type-checking. This means that we must impose our own type scheme onto Prolog in order to check it. This means, as we encounter values through the execution of goals, we must convert them to an abstract representation that satisfy our requirement, in this case, of maximum safety in type-checking. Consequently, we must define our own system of types.

Some types in Prolog are obvious. Numbers, for instance. Atoms and lists, as well. But compound terms present more of a challenge. For instance, given the clauses

```
f(a).
f(b).
```

it may be easy to say that given a query/goal `:- f(A)`, `A` is simply of an atom type. But what do we do when we have more complex structures:

```
f(a(q)).
f(b).
```

One clause of the `f/1` predicate has an argument with itself an arity of one, and the other has an arity of zero. In large applications, this may cause problems; for example if all the values of `A` for `f(A)` are gathered into a list (assuming there are many more of them), then the list would have heterogeneous membership in terms of the arities of compound terms. The problem is that this list could, for the most part, be processed correctly by the rest of the program, but in rare (and critical) cases, contain a functor with an arity that cannot be

processed. If this is buried deep in a large program, this may be very difficult to discover and debug.

On the other hand, there are places where heterogeneous lists are actually useful. For instance, many Prolog builtins take lists of options specified as compound terms as arguments. These option lists are often heterogeneous in the arities of their terms. We are thus faced with a balancing act: whether to force structural compatibility between bindings in different clauses for the same predicate or to allow term heterogeneity.

In our implementation, we have decided that our type system will allow heterogeneous lists, but not heterogeneous types for the same argument position of clause heads belonging to a given predicate. In other words,

```
f([a, a(1)]).
f([x]).
```

would be allowed, but

```
f([x]).
f(1).
```

would not. As well, we pay no attention to the structure of compound terms.

In more formal terms, this means that we permit union types in lists but forbid union types in goal arguments. This means that our abstract domain for this problem actually consists of simply three types: numbers, compound terms (including bare atoms), and lists. $\bot$ represents unbound variables, and actually refers to polymorphism in a predicate. $\bot$ is benign. $\top$, however, is not benign; it is the objectionable generic union type for the arguments of goals.

Abstract interpretation must be implemented explicitly in the Banshee code generation scheme described in the following section in that the flow-sensitive analysis we propose below is directly affected by evaluation of goals in the abstract domain. As well, the constants in the program being analyzed must correspond to generated C code that represents types in the lattice. However, abstract interpretation is also implicitly enforced through the Banshee type constructor specification; by using term equality constraints and term "sorts," the achievement of a union type automatically results in an Banshee error on execution of the analysis generated in C.

3.2 The generation scheme

There is no better system for analyzing Prolog programs than Prolog itself. Prolog has simple but extremely powerful reflection. Clauses in the database can themselves simply be represented as terms consisting of the head and the body. Prolog offers the facility to retrieve them as such without executing them. Once a clause has been retrieved, it is sufficient to simply simulate the execution of the clause in the abstract domain[2]. Given that the actual values are not compared for unification purposes in the abstract domain, but rather their

[2]A similar observation was made long ago in Mycroft and O'Keefe [1984].

types are used given the system above, all goals must succeed if there is a clause head with the correct functor/arity combination.

Then if we were to follow the execution model above, only the first clause of a predicate would be used in the development of a typing. This is clearly too much of a commitment for this analysis. We really want the types to be determined given all of the clauses. This is a much more conservative analysis; it permits the maximum opportunity for incompatible types to be given to variables for the same predicate. Consequently, in the abstract domain we assume that every clause fails at the end and move on the to the next clause of the predicate. This ensures that all possible type-bindings for variables are considered.

Each invocation of a goal generates a set of constraints for the Banshee-generated type environment constraint solver. Constraint generation follows the process described here. Given a goal

1. Retrieve all its corresponding clauses.
2. For each clause,
   (a) Collect all the variables in the clause.
   (b) Bind them to C variable names and declare them as Banshee variables in the generated C code.
   (c) Recursively call this process on each term in the body of the clause, performing an occurs check using the stack of goals to prevent infinite mutual recursion. If the occurs check succeeds then the term will not be processed as a goal. Infinite recursion would otherwise happen since no goal fails due to incompatible terms as above.
3. Go through each argument in the goal, and generate a Banshee constraint equation in C between arguments of the goal and corresponding arguments of the head.

Given that in our case we are attempting to limit union types, the only constraint equations we use are term equality equations. However, it is not difficult to permit various kinds of union types by making minor alterations to the proposed abstract domain and generating set inclusion constraints instead.

The generated C code is compiled using the Banshee libraries and then run. If the program terminates early with an error, it means that the program did not type check: a type equality constraint was proposed that resulted in a contradiction. Unfortunately, the error handling interfaces of Banshee do not appear to be documented; otherwise we would have implemented a more sophisticated way to deal with programs that do not type-check.

This system represents in C code what the substitution environment constraints represent mathematically. Substitutions that occured prior to the execution of the given goal are type-assignments of Banshee variables. They are not explicitly unified with the variables in the corresponding clause prior to execution (as specified in the top-down environment constraint form), but they appear on the left side of a constraint equation when the clause

is finished being processed. Similarly, at the end of clause, the variables that were bound during execution of the clause appear on the right side of constraint equations.

One difference, however, between this process and the environment constraint representation is that program points in the mathematical representation were assigned flow-insensitively. However, our implementation creates a different program point each time a goal is considered on a different execution path, as it simply follows the depth-first search of Prolog execution. Developing the Banshee code generator in Prolog was thus conducive to producing a flow-sensitive type analysis.

4 Conclusions and future work

We wrote Proshee implementing the above in SWI Prolog and tested it with a small number of sample programs. While union types as goal arguments are prevented by the program, and the correct types usually inferred, unfortunately it seems to be impossible for Banshee to allow union types for lists at the same time, if lists are to be used as arguments as well. The reason for this is that if type equality constraints are to be used on prolog terms in general, it is impossible to specifically exclude lists and instead use type inclusion constraints. The only way to do this would be—potentially—to implement a new kind of Banshee "sort" (beyond its builtin term and set constructors) that might allow one to specify type constructors that were specifically excluded from the type equality constraints or were subject to weaker constraints. Until this is implemented, Prolog lists are always incompatible with each other in Proshee unless they share the same specific type structure. This can be easily fixed by ignoring the types of elements within the list, but then this impoverishes the detail in the analysis proposed above.

From a more immediate practical standpoint, a need for Proshee would be to develop a better user interface. At this point, type incompatibilities are marked off by Banshee errors. These errors are not very descriptive and they do not suggest precisely where the error actually occured. Banshee has an error handling mechanism, but like many things it does not appear to be documented. Finally, integrating this type-checking system into a Prolog system (at present it compiles it externally as a C program and prints the results in a way that cannot be directly used by another Prolog program) would be a great improvement.

For Proshee as it stands, the lattice of types is fixed. One further innovation would be to allow the dynamic generation of Banshee specifications from Prolog-internal type definitions. This would allow Prolog programmers to permit certain kinds of union argument types in a more specific manner.

However, as it stands, Proshee functions well as a prototype for Banshee-based Prolog type analysis. With minor changes, it can easily be extended to identify other kinds of Prolog structure or even permit union types in a completely descriptive manner.

References

Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41. ACM Press, 1993. ISBN 0-89791-595-X.

Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991. Update from 1997 at `http://www.vanx.org/archive/wam/wam.html`.

Michael Codish and Vitaly Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238(1–2):131–159, 2000. URL `citeseer.ist.psu.edu/article/codish96type.html`.

Nevin Heintze. *Set-based program analysis*. PhD thesis, Carnegie Mellon University, 1992.

Fritz Henglein. Type inference and semi-unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 184–197. ACM Press, 1988. ISBN 0-89791-273-X.

Lunjin Lu. A precise type analysis of logic programs. Principles and Practice of Declarative Programming, ACM, 2000.

Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984. ISSN 0004-3702.

Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language, 1995. URL `citeseer.ist.psu.edu/somogyi95mercury.html`.