

Moving SWI and Yap gradually to Typed Prolog

Tom Schrijvers^{1*}, Vitor Santos Costa², Jan Wielemaker³, and Bart Demoen¹

¹ Department of Computer Science, K.U.Leuven, Belgium

² CRACS & FCUP, Universidade do Porto, Portugal

³ HCS, University of Amsterdam, The Netherlands

Abstract. Prolog is traditionally not statically typed. Since the benefits of static typing are huge, it was decided to grow a portable type system inside two widely used open source Prolog systems: SWI-Prolog and Yap. This requires close cooperation and agreement between the two systems. The type system is Hindley-Milner. The main characteristics of the introduction of types in SWI and Yap are that typing is not mandatory, that typed and untyped code can be mixed, and that the type checker can insert dynamic type checks at the boundaries between typed and untyped code. The basic decisions and the current status of the *Typed Prolog* project are described, as well as the remaining tasks and problems to be solved.

1 Introduction

We resolutely choose for the most established type system, that of Hindley and Milner [4]. It is in wide-spread use in functional programming languages and has already been proposed various times for logic programming. The first and seminal proposal in the context of LP is by Mycroft and O’Keefe [5], and the most notable typed Prolog variants are Gödel [2], Mercury [7], CIAO [1] and Visual Prolog [6]. However, traditional Prolog systems have not followed that trend towards types, and many Prolog programmers continue to use an untyped Prolog, because switching to a new language is usually not an option. Our approach intends to remedy this by addressing the following critical issues:

- Our type system is presented as an add-on (a library) for *currently used Prolog systems*, SWI and YAP, rather than being part of yet another LP language. This means that programmers just need to learn the type system and can stay within their familiar programming language.
- The type system is *optional* with granularity the predicate. This allows users to gradually migrate their existing untyped code, to interface with untyped legacy code (e.g. libraries) and to hold on to Prolog idioms and built-ins for which Hindley-Milner typing is not straightforward.

* Tom Schrijvers is a post-doctoral researcher of the Fund for Scientific Research - Flanders.

- Particular care goes to *interfacing typed with untyped code*. Our approach can introduce a runtime¹ type check at program points on the border between typed to untyped code. In this way, bugs in untyped code are caught at the boundary and do not propagate into the typed code, i.e. the user knows where to put the blame.

In its current incarnation, our system only type checks predicate clauses with respect to programmer-supplied type signatures. In the future, we intend also to automatically infer signatures to simplify the programmer’s job.

2 The Hindley-Milner type system

In order to support the Hindley-Milner type system, we follow standard practice, with a syntax that is nearly identical to the Mercury syntax. *Types* are represented by terms e.g. `boolean`, `list(integer)`, ... Types can also be or contain variables; those are named type variables and polymorphic types respectively, e.g. `T` and `list(T)`.

A *type definition* introduces a new type, a so-called *algebraic data type*. It is of the form `:- type t(\bar{X}) ---> f1($\bar{\tau}_1$) ; ... ; fn($\bar{\tau}_n$).`, which defines a new polymorphic type `type t(\bar{X})`. The type variables \bar{X} must be mutually distinct. The $\bar{\tau}_i$ are arbitrary types whose type variables are a subset of \bar{X} . Also, the function symbols f_i/a_i must be mutually distinct, but they may appear in other type definitions.

A *type signature* is of the form `:- pred p($\bar{\tau}$)` and declares a type τ_i for every argument of predicate p . If a predicate’s signature contains a type variable, we call the predicate polymorphic.

A fully *typed program*, i.e., there is a signature for each predicate, is well-typed iff each clause is well-typed.

A clause is well-typed if we can find a consistent typing of all variables in the clause such that the head and body of the clause respect the supplied type signatures. The arguments of the head must have the same type (up to variable renaming) as the corresponding predicate’s signature. The types of the arguments in body calls must be instances of the corresponding predicates’ signatures. We refer the reader to [5] for a formal treatment and for concrete examples to later sections.

While this works fine for fully typed programs that do not use the typical Prolog built-in constructs, some care is needed for programs calling built-ins or containing a mix of typed and untyped code. Sections 3 and 4 deal with these issues.

3 Support for Prolog Features

Arithmetic Expressions Prolog-style arithmetic does not fit well in the Hindley-Milner type system. The problem is that variable `X` in `Y is X + 1` can be a

¹ In contrast with the compile time checking of typed code.

number, or a full-fledged arithmetic expression. Hence, numbers are a subtype of arithmetic expressions. Unfortunately, it is an old result that subtyping in Prolog can go wrong [3]!

In the current implementation, variables in arithmetic expressions can be of a numeric type only. We are considering to relax this by overloading the expression argument types to be *either* arithmetic expression *or* numeric types.

Built-ins Some Prolog built-ins cannot be given a sensible Hindley-Milner type, such as `arg/3`², which extracts an argument of a term. In general, the type of the argument depends on the index number, which may not be statically known.

Nevertheless, for many Prolog built-ins there is a straightforward signature. Some of the built-ins our system supports are:

```
:- pred var(T).      :- pred ground(T).    :- pred write(T).
:- pred (T == T).   :- pred (T @< T).

:- pred compare(cmp,T,T).          :- pred reverse(list(T),list(T)).
:- type cmp ---> (<) ; (=) ; (>).  :- type list(T) ---> [] ; [T|list(T)].
```

Meta-Predicates Meta-predicates take goals as arguments. They are supported through the higher-order type `pred`. For instance, the types of some well-known built-in meta-predicates are:

```
:- pred \+(pred).    :- pred once(pred).    :- pred setof(T,pred,list(T)).
```

It may seem problematic in a goal like `setof(X,Goal,List)` to figure out the type of `X`. This is not so: the necessary information is usually provided by an earlier goal, e.g. `Goal = between(1,10,X)`. The former forces the type of `Goal` to be `pred`. Hence, from the latter it follows that `X` has type `integer`, assuming the signature `:- pred between(integer,integer,integer)`.

The meta-predicate support is generalized to higher-order predicates with closures as arguments, i.e. goals missing one or more arguments. For instance, the well-known `maplist/3` predicate has the signature:

```
:- pred maplist(pred(X,Y),list(X),list(Y)).
```

Atoms For lack of the conventional strings, many Prolog programmers resort to using atoms instead. In order to support this convention, our type system offers the `atom` type containing all atoms. Hence, the ISO-Prolog built-in `atom_concat/3` has signature

```
:- pred atom_concat(atom,atom,atom).
```

Note that a true `string` type would offer a cleaner solution.

² Its type-friendly counterpart are typed (record) field selectors.

4 Interfacing Typed and Untyped Code

One of the most distinguishing properties of our type system is its support for interfacing typed with untyped code.

Untyped to Typed While typed code is statically verified by the type checker, untyped code is not. Hence, any call from untyped code (or the Prolog toplevel) to typed code can go wrong, if the provided arguments are not of the required types. If left unchecked, such an ill-typed call may manifest itself elsewhere far away in the code and greatly complicate the debugging process.

By default, we prevent this scenario by performing a runtime type check on any call from untyped to typed code (by means of a simple program transformation). If the call is ill-typed, it is caught before the actual call is executed. Then, the programmer knows the untyped code leading up to the call is to blame.

Typed to Untyped Also in the inverse situation, when calling untyped code from typed code, we want to catch type violations early on in order to blame the untyped code. In order to do so, the programmer has to supply a type annotation for the call to untyped code. This allows to statically verify whether the surrounding typed code is consistent with this annotation. On top of that, a runtime check whether the untyped code satisfies the type annotation is inserted. The check is performed right after the call returns: any logical variables improperly bound by the call are detected in this way.

As an example, consider the following predicate from Santos Costa's red-black tree library:

```
:- pred list_to_rbtree(list(pair(K,V)),rbtree(K,V)).

list_to_rbtree(List, T) :-
    sort(List,Sorted) :: sort(list(pair(K,V)),list(pair(K,V))),
    ord_list_to_rbtree(Sorted, T).
```

Assume the `sort/2` predicate is untyped, whereas the other predicates are typed. The programmer has annotated the call (after `::`) with the missing type information `sort(list(pair(K,V)),list(pair(K,V)))`. Based on the annotation, the type checker assumes that the arguments `List` and `Sorted` both have the type `list(pair(K,V))`. Moreover, a runtime type check is inserted right after the call, to check whether the two arguments actually have this type.

The programmer can optionally declare that the runtime check need not be performed.

Untyped Terms The programmer is forced to make a single one-off decision for a predicate: either she provides a signature and the predicate is typed, or she does not and the predicate is untyped. The former choice is the most desirable, but may require pervasive changes to the code as all terms handled by the predicate must be typeable, and hence be made to respect the Hindley-Milner data type conventions.

We provide the programmer a way out of this dilemma with the universal type `any`, which covers all possible terms. Now the programmer only provides precise types for the terms she wants, and defers the job for the others by typing them with `any`. For the subsequent gradual and localized intruction of more precise types, terms of type `any` can be coerced to other types, and vice versa.

5 Conclusion

The Typed Prolog project is based on the belief that it is better to gradually introduce types in an existing language than to start from scratch with a new language. People tend not to migrate to another system just because of types, hence our decision to introduce types into Yap and SWI, two widely used Prolog systems, and in such a way that users can gradually adapt to the use of types.

We are aiming at making this process as pleasant as possible, with special support for Prolog language features and for interfacing typed with untyped code, and while not forcing the Prolog programmer to give up essential functionality.

The Typed Prolog project started in the spring of 2008 and now consists of about 1,000 lines of code. It is no surprise that there are still many issues to tackle: error messages, handling floats and rationals, complete integration with the module system, dealing with large sets of facts, adaptation of the runtime checks to delayed execution, general support for constraint solvers, . . .

The simultaneous introduction of the same type system into SWI-Prolog and Yap is furthermore a clear sign of the commitment of their respective development teams to unify their functionality. Library `type_check` will be available in their next release.

Acknowledgements The authors are grateful to Roberto Bagnara, Fred Mesnard and Ulrich Neumerkel for there helpful comments.

References

1. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
2. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
3. P. M. Hill and R. Topor. A semantics for Typed Logic Programs. In M. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.
4. R. Milner. A theory of type polymorphism in programming. *Journal of Computer System Sciences*, 17:348–375, 1978.
5. A. Mycroft and R. A. O’Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.
6. Prolog Development Center. Visual Prolog. <http://www.visual-prolog.com>.
7. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.