A Gradual Polymorphic Type System with Subtyping for Prolog

Spyros Hadjichristodoulou

Computer Science Department, Stony Brook University
New York, U.S.A.

Abstract

Although Prolog was designed and developed as an untyped language, there have been numerous attempts at proposing type systems suitable for it. The goal of research in this area has been to make Prolog programming easier and less error-prone not only for novice users, but for the experienced programmer as well. Despite the fact that many of the proposed systems have deep theoretical foundations that add types to Prolog, most Prolog vendors are still unwilling to include any of them in their compiler's releases. Hence standard Prolog remains an untyped language. Our work can be understood as a step towards typed Prolog. We propose an extension to one of the most extensively studied type systems proposed for Prolog, the Mycroft-O'Keefe type system, and present an implementation in XSB-Prolog. The resulting type system can be characterized as a Gradual type system, where the user begins with a completely untyped version of his program, and incrementally obtains information about the possible types of the predicates he defines from the system itself, until type signatures are found for all the predicates in the source code.

1998 ACM Subject Classification D.1.6. Logic Programming, D.3.3. Language Constructs and Features

Keywords and phrases Type Inference, Polymorphic Type System, Gradual Typing, Tabling, Answer Subsumption

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.451

1 Introduction and problem description

Since the seminal work of Mycroft and O'Keefe [5] in introducing a polymorphic type system for Prolog, there has been vast research on the area. Some of it followed their direction, and is essentially about extending or reconstructing the Mycroft-O'Keefe type system [7, 2], while others take different paths for introducing types in logic programs [6, 1, 3].

The common denominator in these approaches, however, is that most of them are about *theoretically* defining and constructing a type system for Logic Programming. Although some early Prolog implementations contained type checking mechanisms based on the Mycroft-O'Keefe type system (e.g. the DEC-10 compiler), most modern systems tend to keep Prolog as an untyped language [1]. Of course, there is the exception of Mercury (`http://www.mercury.csse.unimelb.edu.au/`), which has become famous for the type-checking abilities it offers to programmers; however, this comes with the price of strong typing and "limited" polymorphism of Mercury programs.

As discussed in [5], the main purpose of developing a polymorphic type system for Prolog is to provide the programmer with another tool which will make programming easier and

[1] As we will discuss later, an implementation for the Mycroft-O'Keefe type system was developed with the intention of being distributed with SWI-Prolog and YAP

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 451–457

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

less error-prone. We consider our research as a step towards that direction; our goal is to build a working type system which will enable programmers to write correct Prolog programs more easily than before. The challenge is to somewhat combine various aspects of the approaches introduced in the literature and use modern techniques to implement a robust type inference system which will be distributed with XSB Prolog. Our type checking and inference mechanism will offer users two modes of operation; firstly, they will be able to type-check their program, if they provide a type signature for every predicate they define. Secondly, if some of these signatures are missing, the type inference engine will be able to infer types for the respective predicates, in order to provide the user with information about their newly defined predicates. This process will be conducted in an incremental, *gradual* manner; the user will start with a completely untyped version of his program, and type inference will gradually give more information about what the types of the defined predicates may be. If the user is satisfied with the type inference engine's suggestions, then these types will be considered by the system as if they were declared by the user as type signatures. This process will continue until each defined predicate in the source code has a type signature. This kind of type systems, called *Gradual Type Systems* was introduced in [8].

2 Background and overview of the existing literature

The first work introducing some kind of type checking and inference in Prolog was Mycroft and O'Keefe's Polymorphic type system, [5]. It is based on the seminal work by Robin Milner, [4], who created a polymorphic type system for the ML family of functional programming languages, and first introduced the notion of "Well-typedness". In the Mycroft-O'Keefe type system, type signatures are provided by the user for each defined predicate, and the type checker's task is to verify that each definition respects the signature declaration. The only notion of *inference* in this type system is with **Variables**; when a predicate `p(X,Y)` is type-checked against its signature, a type is inferred for both `X` and `Y`. Also, it allows for *polymorphism* in the usual meaning; arguments of predicates can be of *any* type, denoted by type variables, and allows for user-defined type constructors as well as some predefined ones (i.e. for lists, `type list(A) --> [] ; [A|list(A)]`). Moreover, a connection is made between the Prolog program to be type-checked and the type checker, which is itself another Prolog meta-program. Finally, the notion of "Well-typedness" is introduced, and has the same meaning as in the Hindley-Milner type system: "Well-typed programs can't go wrong". In the context of Prolog, this means that no predicate will be called with arguments that don't respect the type signature declared by the user. The following example illustrates the operations described above:

▶ **Example 1** (`append/3`)**.** The user declares a type for the well-known `append/3` predicate, as `:- pred append(list(A),list(A),list(A))`. This means that `append/3` has 3 arguments, and each has the same type, namely `list(A)`. So, each argument can be a list of anything, as long as all 3 arguments have the same type.

The user may have the definition of `append/3` available:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Each clause of `append/3` will now be type-checked against the given signature. It's easy to see that each clause respects the declared signature, and so `append/3` is well-typed. Finally, if the following clause appears in the program, `p(X,Y,Z) :- ... , append(X,Y,Z), ...`, then types for `X`, `Y` and `Z` are inferred according to `append/3`'s signature (and all are `list(A)`).

A recent implementation of the Mycroft-O'Keefe type system was developed in 2009 [7]. The authors' aim was to gradually[2] introduce types in Prolog using a type-checking library that was planned to be shipped with two of the most popular Prolog implementations, SWI-Prolog and YAP. This type checking library makes it easy to interface typed and untyped code, by performing runtime checks when typed predicates (i.e. predicates for which the user has provided a type signature) are called. By doing that, the authors make sure that the type system can be used somewhat "on-demand", i.e. only when the programmer gives type signatures for certain predicates, and this makes the migration from untyped to typed Prolog easier.

A rather different approach was introduced in [1]. The authors employ a fixed-point, bottom-up, abstract interpretation technique in order to infer types for Horn-Clause programs. Type declarations for predicates and constructors are very similar to the Mycroft-O'Keefe type system [5], but in this approach, the system can infer types for predicates while not depending on a type signature provided by the user. For example, the type of `append/3` can be inferred only by its definition and the definition of the `list(A)` constructor, as given above.

3 Goal of the research

Our goal is to combine the advantages of each of these approaches in order to build a robust inference system for types of both predicates and constructors. The system will be distributed along with XSB-Prolog, and is implemented as a preprocessor of the original source code. This allows the user to start with a completely untyped Prolog program, and with the help of the inference engine to *gradually* learn more about the types of the predicates he defines. We believe that the existence of such a system will make programming in Prolog easier and less error-prone, while at the same time maintaining the flexibility that Prolog gives.

4 Current status of the research

Our first task was to port the type-checking library from [7] which was written for SWI-Prolog and YAP, to work in XSB-Prolog. Based on the XPP-Preprocessor [3], we implemented a preprocessor that gets invoked by the user with a compiler flag. If the user desires to provide type signatures for any of the predicates he has defined, the only thing to do is include the following declaration in his source code: `:- compiler_options(xpp_on(typecheck)).`

After successfuly porting the type-checking library to XSB-Prolog, we used the approach in [1] to build a type-inference engine. We use the same notations for declaring the types as in [5]. Defined predicates for which a type signature has been provided get type-checked, whereas the type of the others is inferred. Despite the similarities between our approach and the ones discussed previously, there are basic differences:

- We extended the Mycroft-O'Keefe type system in order to lift the limitation that each clause of a predicate must have the same type. Prolog programmers often want to define facts which may have different types, and we didn't want our engine to infer that this

[2] The use of the term "gradually" here has a different meaning that the one we used to describe our approach in the previous section. In [7] it is used to describe the *migration* process from untyped Prolog to typed Prolog, whereas in our approach, it is used to describe the process of adding types to a single program, starting from a completely untyped program and moving towards a fully typed program

[3] `http://www.cross-browser.com/x/docs/xpp_reference.php`

kind of predicates is ill-typed. If, for example, the user has defined two facts as `p(42)` and `p(a)`, the type-inference engine will give `p/1` the type `p(atomic)`, instead of failing. For this scheme to work, we have introduced simple-fixed subtyping rules between the primitive types that each program can have. For example, `integer`, `atom` and `float` are all subtypes of `atomic`.

- In [1], the authors use a "cut-off" point to stop their inference when the type of a predicate grows bigger at each step. Instead of doing this, we are using `unify_with_occurs_check`, so that when two clauses of a predicate give types that can't be unified with the occurs check, our system infers that the predicate is ill-typed. This approach was also taken in type-checking in [7].
- None of the previous approaches were able to handle the case of inferring types for type constructors. We are currently developing a type inference mechanism which will be invoked when the user requests, which will try to infer what the type of a defined constructor may be. This may be particularly useful when using large libraries with many new constructors but no documentation on what each constructor does. We hope that being able to see the type of each constructor will be useful to the programmer for better understanding external code.

Algorithm 1 Outer fixed-point

```
1: do_type_inference_batch(PredList,TypedListIn,TypedList) :-  {Let PredList
   be the list of predicates we want to infer types for, TypedListIn be the list of types for
   the predicates in PredList, TypedList be the list of types that will be inferred in one
   step}
2: for all Pred in PredList do
3:    type_inference_batch(Pred,Type,TypedListIn)
4: end for
5: Let TypeListTemp be the list consisting of all the returned Types
6: if TypeListIn != TypeListTemp then
7:    call do_type_inference(PredList,TypeListTemp,TypedList)
8: else
9:    set TypedList = TypeListIn
10: end if
```

Algorithm 2 Inner fixed-point

```
1: type_inference_batch(Pred,Type,TypedList) :-  {Let Pred be the predicate we
   want to infer types for, Type be the type we will infer for Pred, TypedList be the list of
   types that were inferred in the previous step, TypeIn be the type of Pred computed in
   the previous step as it resides in TypedList}
2: for all clauses of Pred do
3:    call type_inference(Pred,TypedList,TypeIn)
4: end for
5: Gather all newly constructed TypeIns in a list, TList
6: Unify all elements of TList wich each other
7: Unify Type with the Head of TList
```

We currently have 3 versions of the type-inference engine. In the first, which is described in algorithms 1, 2 and 3 below, the predicates that perform type-inference are all non-tabled. The problem with this approach, is that we needed to pass around a list of all the

Algorithm 3 Find a type from only one clause of the predicate

```
1: type_inference(Pred,TypedList,Type) :- {Let Pred be the predicate we want to
   infer types for, Type be the type we will infer for Pred, TypedList be the list of types
   that will be inferred in one step}
2: for each clause of Pred do
3:    Find a type for Pred from the body of the clause and store it in Type
4:    Find a type for Pred from the head of the clause and store it in PredType
5:    if Type and PredType can be unified with occurs check then
6:       succeed
7:    else
8:       throw error
9:    end if
10: end for
```

(intermediate) types that had been inferred for each predicate of the source code, in order to use the newest information at each step.

In order to remedy this, we re-wrote the basic type-inference predicates into a tabled version. Each time a new type is inferred for a predicate, a record is entered in the global table kept by XSB, so when the type of any predicate is requested during the process, it can be obtained by looking-up the table, instead of passing around a list. This approach also enabled us to be able to give some type to mutually recursive predicates.

For the final version of our type inference engine, we employed the principle of `answer subsumption` as described in [9]. In essence, whenever a new answer is produced for a predicate that is tabled with answer subsumption, it's joined with the answer that already resides in the table, and that join is now the only answer for that predicate. Now, instead of using `findall/3` in lines 5-7 and 6-8 of algorithms 1 and 2 respectively to get all the types for each clause and until the fixed point is reached, we use `answer subsumption` and always keep the most *specific* type found for each predicate. This may seem rather illogical in the beginning, since when two answers are produced for a predicate, we tend to keep the most general one. The reason is that when trying to find answers for goals in Prolog, we don't care which clause of the goal will make the answer found true, as long as there is one that does. However, in type inference, the type inferred must respect *all* the clauses of the predicate. We can think of this difference as the duality between *union* and *intersection*; when we want answers for a goal we are looking for the *union* of answers, whereas when we want to find a type for a predicate, we want the *intersection* of types found.

5 Preliminary results accomplished

▶ **Example 2.** We will start with a simple recursive predicate, `reverse_acc/3`. It's the tail-recursive version of `reverse/2`, which binds the output with the input list, reversed.

```
reverse_acc([], Acc, Acc).
reverse_acc([Head|Tail], Acc, Reversed) :-
      reverse_acc(Tail, [Head|Acc], Reversed).
```

Asking our engine to infer the type of `reverse_acc/3` will yield:

```
| ?- infer_types('test.P').

Inferred types for the following 1 predicates:
reverse_acc(list(A),list(A),list(A))
```

▶ **Example 3.** In this second example, we will show how our extensions to the Mycroft-O'Keefe type system behave. The predicate we want to infer a type for is `foo/1`:

```
foo(42).
foo(bar).
```

The original Mycroft-O'Keefe type system would not be able to give a type to `foo/1`. Our extensions make it possible for the engine to assign the `atomic` type:

```
| ?- infer_types('test.P').

Inferred types for the following 1 predicates:
foo(atomic)
```

▶ **Example 4.** For this last example, we will show some preliminary results of our constructor type inference engine. We assume the following code snippet:

```
:- type natural ---> 0 ; s(natural).

formula(0).
formula(s(0)).
formula(s(s(0))).
formula(0 + s(0)).
formula(s(s(0)) - s(0)).
```

The above code declares a new type constructor `natural` for natural numbers, and various versions of the same predicate, `formula/1`. The task is to find what is the type of the constructors of `formula/1`'s argument:

```
| ?- infer_constructors('test.P',formula(_)).

Inferred the following constructors:
formula(natural)
formula(+(natural,natural))
formula(-(natural,natural))
```

The engine has managed to infer that the argument of `formula/1` can be either a `natural` (as per the type constructor above), a `+(natural,natural)` or a `-(natural,natural)`.

6 Open issues and expected achievements

Although the implementation of the type inference engine has progressed over the few last months, there are still issues that need to be resolved

- The final version of the code where answer subsumption is used must be tested thoroughly and compared to the other versions. It will be interesting to see the differences in both runtime and table usage for large source files
- The type inference for constructors must be refined; the correct type constructors for the last example of the previous section would be `expr --> natural ; expr + expr ; expr - expr`, so our engine must somehow recognize that the `+` and `-` combine more complex things than simple `natural`s

References

1 R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Sci. Comput. Program.*, 19(3):281–313, 1992.

2 T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp*, pages 202–217, 1991.

3 L. Lu. Polymorphic type analysis in logic programs by abstract interpretation. *The Journal of Logic Programming*, 36(1):1–54, 1998.

4 R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

5 A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

6 T. Schrijvers and M. Bruynooghe. Towards constraint-based type inference with polymorphic recursion for functional and logic languages. In *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, pages 1–16, 2005.

7 T. Schrijvers, V. Santos Costa, J. Wielemaker, and B. Demoen. Towards Typed Prolog. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 693–697. Springer-Verlag, 2008.

8 J.G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, page 7. ACM, 2008.

9 T. Swift and D. S. Warren. Tabling with answer subsumption: implementation, applications and performance. In *Proceedings of the 12th European conference on Logics in artificial intelligence*, JELIA'10, pages 300–312. Springer-Verlag, 2010.