ADDING REDUNDANCY TO OBTAIN MORE RELIABLE AND MORE READABLE PROLOG PROGRAMS

by M. Bruynooghe

Report CW 27, June 1982

Departement Computerwetenschappen
KATHOLIEKE UNIVERSITEIT LEUVEN

To appear in the Proceedings of the First International Logic Programming Conference.

Adding redundancy to obtain more reliable and more readable Prolog programs.

Maurice Bruynooghe
Departement Computerwetenschappen
K.U.Leuven

keywords : Prolog logic programming
data flow type checking software engineering

Abstract

Prolog programs are very error-prone, small typografical errors do not result in compile time errors but in programs with different unintended meanings. The paper contains a proposal to improve the situation. It suggests to add redundant information about the flow of data through clauses and about the possible values of arguments and gives a method to analyse the consistency between this additional information and the text of the prolog clauses.

1. Motivation

Prolog is considered as a very high level language by its followers due to its expressive power, its declarative reading, its ease of creating and manipulating data structures, ... [1]. We could say that the language has a high level contents or semantics. However, its form or syntax is rather low level compared to the current practice in high level languages. Prolog programs are very error-prone. Small syntactical deviations, i.e. spelling errors in names of variables, procedures or functors, not enough or to much arguments for a predicate or functor, ... are unlikely to produce compile-time errors. They simply result in a different undesired computation. It is a frustrating task to debug such programs. The quality of the resulting programs can be doubtfull due to the lack of thoroughly testing.

The reason is the complete lack of redundancy in the programs : procedure definitions are not introduced by a declaration defining the number, the type and the call mechanism of their parameters; variables are not declared and typeless (see common errors in [1]). This has also an attractive size, it reduces the clerical work of writing programs and entering them on a terminal. However, the disadvantage seems greater, it is below the current standards of software engineering and creates resistance in novice-users with prior programming experience.

This paper describes an attempt to improve the situation by requiring more information from the user. Due to the experimental nature of our approach, the idea is to add formal comments to the programs. A normal prolog interpreter can simply discard this comments. However, the comments are playing a double role :

1) They can increase the readability of the program.

2) A special program can verify whether the formal comments are consistent with the actual program text.

When writing a Prolog procedure, or reading our own procedures, we have a number of things in our mind :

- Although procedures can be called with any input-output pattern, we have only one or a few patterns in our mind. Using the procedure with other patterns is potentially dangerous. Such a call can have an infinite set of solutions, can result in very inefficient computations, or can result in unintended erroneous results due to a difference between the procedural and declarative semantics.

- We associate a set of meaningfull values (a 'type') with each argument in a procedure definition.

- We associate a set of meaningfull values (a 'type') withh each variable. In fact, variables are used inside arguments, their meaningfull values can be derived from those of the arguments.

Usually, people do not take the time to write this information as comments in their programs. This makes programs rather unreadable for everyone except their author.

We have made an attempt to formalize the above information and to develop a method to verify the consistency between the formalized comments and the actual programtext.

To be acceptable, our approach should not cause an explosion of the program size. Declaring all variables of each clause causes such an explosion. We have abolished it, we attempt to derive the types of the variables from the types of the arguments. Our formalized comments consists thus of

- type declarations.

- a declaration for each procedure stating the number and types of the arguments and the intended patterns of the calls.

2. Possible call patterns

While reading a prolog clause procedurally, we simulate its execution. For each call, we ponder how the arguments looks like, i.e. which (parts) are known before execution and what is the effect on the arguments of executing the call, i.e. which (parts) are known after execution. In fact, we are looking how data flows through the clause.

An example

```
Quicksort (x.list,sort) <-- Partition (x,list,l1,l2),
        Quicksort (l1,s1), Quicksort (l2,s2),
        Append (s1,x.s2,sort)
```

Quicksort is a procedure to sort lists, given a complete list as first argument, it should compute its second argument completely (the sorted list). To simulate its execution we start with Partition. This call has access to two completely known arguments x and list. Execution of the call will result in the complete computation of l1 and l2. This means that both recursive calls to Quicksort know their first argument. Thus they will compute their second arguments, s1 and s2. As a result, the call to Append has access to two complete arguments and will compute its third argument, sort, completely. This is exactly the desired result for Quicksort.

To formalize this validation of the data flow through Quicksort, it suffices to associate a pattern with each procedure, i.e. (i,o) with Quicksort, (i,i,o,o) with Partition and (i,i,o) with Append. In such a pattern i (input) means that the argument is completely instantiated before execution and o (output) that the argument is not known before execution but becomes completely instantiated by the execution of the call.

A typografical error, i.e. changing an occurrence of l1 in l, will destroy this dataflow, it will be impossible to conclude that Quicksort computes its second argument.

Some procedures are general purpose, Append, for example, can also be used to split a list in different parts, to subtract two lists,... . Thus we can associate different patterns with the same procedure e.g. (i,i,o), (o,o,i), (i,o,i),... with Append.

Also, an argument can be incomplete before and after execution of a call. For example, to behave efficiently, it suffices that Append knows its first argument. We have to introduce an annotation nio standing for 'neither input nor output' and we can declare a pattern (i,nio,nio) for Append.

It is reasonable to state that the input part of a pattern uniquely determines the remainder of the pattern. Either an argument is computed (o) or it is not (nio). Thus, the choice of a matching pattern is deterministic. However, the availability of different patterns can obscure the effect of typografical errors. One of the other patterns can match an erroneous call. But, because different patterns create different output, it is unlikely that the output of the clause, as derived from a data flow analysis is in agreement with what has been declared in the pattern of the call.

Although our simple annotations i,o and nio can cover a large amount of simple Prolog procedures, they are clearly insufficient to cover everything. In fact, Prolog is so flexibel, it is possible that a call consumes a complete component of an incomplete argument or produces a partially complete argument or ... that it is unclear to us how to design a simple compact annotation to cover all possible uses. In the above example of Quicksort, it is possible to switch the last two calls. We can execute Append before s2 is known. This results in an incomplete sort. The call to Quicksort will fill the hole in sort. However, our dataflow analysis will fail to notice this and will not validate the procedure. At the end of the paper, we discuss an extension which is able to cover such a frequently occurring situation.

The algorithm to validate the dataflow through a clause for a given input-output pattern is quite simple. The data flow enters the clause in the input arguments (i) of the heading. All variables occurring in such a term are completely instantiated. Then we handle the calls from left to right. We have to look for a matching input-output pattern. Arguments with all variables completely instantiated have to match i, the others either o or nio. All variables occurring in an output argument (o) become completely instantiated and we process the next call. Once all calls are handled, we have to validate the output part of the heading. All variables occurring in an output (o) argument have to be completely instantiated; a nio argument has to contain at least one incomplete variable.

3. Types

In the previous section, we have introduced input-output patterns to enhance the readability of programs and to perform some validation. This section discusses another aspect contributing to the same goal : the association of sets of meaningfull values ('types') with the arguments of predicates and with the variables.

To avoid confusion and errors, it is usefull to clasify atomic objects into several separated types, especially in database oriented applications. For example, we can distinguish between students, courses and rooms. This improves the readability of clauses and can be used to detect errors in calls like the switching of two arguments. In such cases, the use of types will create an incompatibility between the actual types in the call and the formal types in the heading, e.g. using a room where a course is expected and a course where a room is expected.

It is also possible to introduce a type which is a subset of another type, e.g. a type person with subtypes student, man and woman. To judge the usefulness of this, we have to look what happens with calls involving such types. We have two important cases :

- The formal parameter (e.g. person) is more general than the actual parameter (e.g. student). Because a student is also a person, this is completely legal.

- The actual parameter (e.g. person) is more general than the formal parameter (e.g. student). In Prolog, there is no reason to state this as an error, it only means that the procedure will fail in some cases, e.g. for the persons which are not students, but will succeed in other cases.

We can only recognize an error when the procedure will always fails, thus when the types of the actual and formal parameters are disjoint. Thus, for validation purposes, the only important question is whether two types are disjoint or not. In favor of subtypes is their contribution to readability of program. Their disadvantage is the complexity of their handling : it is lengthy to describe which types are disjoint and which are not. Moreover, it is unclear how they can contribute to the validation process. We have decided to consider all types as disjoint.

Atomic types can be defined by a set of values e.g. :

```
Atomic Color = [red,blue,yellow]
       Month = [1..12] (integers in the interval 1 to 12)
       Room = [...]
         '...' means an undefined range of values.
```

Besides atomic objects, we also need structured objects like lists, trees,... . Actually, we would like to distinguish between lists of rooms, lists of courses, lists of trees of students,... . Because all kinds of lists have the same global structure, it would be very inconvenient to define each kind of list separately. For this purpose, we have introduced typeschemas.
An example :
List (any) = [nil] ¦ . (any,List (any)).
This is a typeschema for lists. It has one parameter any. To obtain the definition of a particular type of lists, we replace the parameter any by that type. For example
List (Tree (student)) = [nil] ¦ . (Tree (Student),List (Tree (Student))).
The righthandside hase one component (between square brackets) giving the atomic objects belonging to the type. The remainder consists of alternative rules to obtain elements of the types. Each rule consists of a n-ary functor followed by n types. To facilitate the derivation of the types of the components (variables) from the type of a term, we require that all functors in a type schema are different.

Schemas are also very useful to declare general purpose procedures. A procedure like Append can be used to concatenate any

kind of lists. We declare append with the following schema :
Append (List (any), List (any), List (any))
A particular instance is obtained by replacing the parameter by a particular type, e.g.
Append (List(Room),List(Room),List(Room));
which expresses that appending two lists of rooms results in a list of rooms. However, other procedures are less general, a sorting procedure does not work for all lists but for example only for lists with atomic elements. We found it convenient to clasify atomic types in three classes :
Int : all elements are integers.
String : none of the elements are integers.
Atom : some elements are integers, others are not. (Int, String and Atom are never used as types, they are type-classes). Now we can express that a parameter is restricted to the subtypes of one of these classes, e.g. any-Atom, any-Int,... and we can declare :
Sort (List(any-Atom),List(any-Atom)) or
Sorttrees (List(Tree(any-Atom)),List(Tree(any-Atom))).
A particular instance is obtained by replacing the parameter by a subtype of Atom.

In general, a schema can require more than one parameter, to distinguish between them, we can use a set of prefixes like any1, any2,... .

We can recognize different parts in the validation of the type information about a clause :

- verification that we cannot derive two disjoint types for the same variable.

- verification that the types of the actual and formal parameters in a call are not disjoint.

- given the types of the input arguments of a clause, verification that the types of the results, as obtained by executing the calls in the body are in agreement with the declared types of the output arguments. This makes it necessary to perform the validation for each input output pattern.

The use of declaration schemas poses special problems to the validation :

1. The clause being validated has a declaration schema. In fact, the clause has to be validated for each instance which will ever be used. We restrict ourselves to validation of the clause for a random instance replacing the parameter by a completely new type (with the appropriate subtype restriction).

2. A call has a declaration schema : it is necessary to select the appropriate instance using the available information about the actual parameters.

To discuss the validation more in detail, we give an example. The relevant facts to validate the types in a Quicksort clause

are:
```
   Structure List(any) = [nil]¦.(any,List(any)) .
   Decl Quicksort(List(any-Atom),List(any-Atom)), use (i,o) .
   Quicksort(nil,nil) <--
   Quicksort(.(x,list),sort) <-- Partition(x,list,l1,l2),
          Quicksort(l1,s1), Quicksort(l2,s2),
          Append(s1,.(x,s2),sort).
   Decl Partition(any-Atom,List(any-Atom),
                 List(any-Atom),List(any-Atom).
   Decl Append(List(any),List(any),List(any)).
```

Because Quicksort has a declaration schema, with parameter any-Atom, we introduce a new type A which is a subtype of Atom and we have to show that, given a first argument of type List(A), Quicksort succeeds and its second argument obtains the type List(A).

We start with the input part of the heading. (For this validation, we also consider nio arguments as input). We have to compare the argument with the type. In case of Quicksort we have the term .(x,list) as input. By consulting the type definition of List(A), we obtain A and List(A) for the type of the components. We conclude with x having type A and list having type List(A).

Next we handle the calls from left to right. To handle a call, we have to distinguish between calls having a normal declaration and those having a declaration schema. With normal declaration, the treatment is simple : compare the argument with the type. Arriving at a variable with undefined type, we assign the type to the variable. Finding an already typed variable, we verify that the type of the variable is not disjoint (thus equal) with the type of the argument. Confronted with a constant we look whether the constant has the desired type.

Example : compare
   .(John.(x,l)) with x of type Student and l of undefined type with List(Student). The result is : a message stating that John should be of type Student, the confirmation of x having type Student and the assignment of type List(Student) to l.

Having a call with a declaration schema, we have to select an instance of the schema. This is obtained by comparing the type-information of the actual arguments with the schema. This results in assignments of types to the parameters. Taking the call to Partition in Quicksort we compare x having type A with any-Atom. We assign type A to any-Atom (after verifying that A is a subtype of Atom). Comparing list of type List(A) with List(any-Atom) results in comparing type A with the parameter which already has a type. We verify that both types are equal. The variables in the other arguments do not have a type and we conclude that our instance is derived by replacing any-Atom with A. Having eliminated the schema, we compare the arguments with the type. In case of Partition this results in a type List(A) for l1 and l2. In general, it is possible that a parameter has not received a type. In such a case, we introduce a completely new type with the proper subtype restriction (see next example).

The treatment of the other calls of Quicksort is similar. The instances of the Quicksort declaration are obtained by replacing any-Atom with A, the instance of Append by replacing any with A. The handling of the body ends with assigning List(A) to sort.

Finally, we have to verify that the output argument of the clause has the desired type, i.e. that the computed type is equal to the desired type. In our example, both types are identical. In general, the output argument is a term and we have to derive the desired types for the variables occurring in it and to compare the desired types with the types obtained by processing the body.

Another example :

```
Atomic Month[1..12] subset of Int;
Decl Nextmonth(Month,Month), use i,o
Nextmonth(m,nm) <- Lt(m,12), Plus(m,1,nm).
Decl Lt(any1-Int,any2-Int);
Decl Plus(any1-Int,any2-Int,any3-Int);
```

Processing the heading results in m having type Month. For the call Lt, we select an instance Lt(Month,A) of the type schema with A a new subtype of Int. A message that 12 should be of type A, a subtype of Int is given. For the next call, we arrive at an instance plus(Month,B,C) with B and C new subtypes of Int. A message that 1 should have type B is given and nm receives type C. Looking at the output part of the heading, we conclude that nm does not have the type Month and another message results. We feel this treatment is appropriate because the correctness is based on the semantics (e.g. using Lt(m,13) or dropping lt makes the procedure invalid). Moreover, to avoid messages all over his program, the programmer is encouraged to use the good programming style of localising the use of procedures as Plus in a few procedures. Remark that declaring Plus as Plus(any-Int,any-Int,any-Int) will result in the selection of the instance Plus(Month,Month,Month) and that the procedure will be validated. We consider this as undesirable.

4. Final Remarks

We have presented a method to improve the quality of Prolog programs. It consists of the introduction of formal declarations and a methodology to analyse the consistency between declarations and prolog procedures. We have extended the programs with two kinds of information :

1. Type information : the user can define atomic or structured types which are mutually disjoint. Atom types are classified as subtypes of Atom, Int or String. For each procedure, the user declares the types of the arguments. We have introduced schemas to obtain a compact notation.

2. Information about the intended patterns of use. We have introduced the annotation i (input = completely instantiated before execution), o (output = completely instantiated after execution but not before) and nio (neither input nor output) to describe allowed patterns of use. These patterns can be used to verify the flow of data through a clause. These patterns are insufficient to describe all possible uses of prolog-procedures. An important case we cannot handle is the use of incomplete data structures (trees with free variables in the leaves, lists with free variables in the tail,...). In fact the correctness of using such data structures often depends on the semantics of "cut" and the ordering between clauses, where our method considers a clause in isolation of other clauses and does not take in account the side-effects of build-in predicates. Another important class we cannot handle is exemplified by Append. Append can be executed with an incomplete second argument. In this case, the third argument is incomplete after execution of the call, but is completed as soon as the second argument becomes complete. To cover this, we could introduce a notation (i,nio,o(2)). o(2) means the argument becomes complete as soon as the second argument is completely known. With such annotations, the validation of the data flow becomes more complex. Upon failure of the validation process, it becomes necessary to backtrack and to try other patterns for the different calls.

We have to await experience with an implementation to judge to value of our current proposal and to obtain insight in the desirable extensions.

5. Acknowledgements

We are indebted to the Belgian "National Fonds voor Wetenschappelijk Onderzoek" for funding a visit to Luis Pereira, and to Luis Pereira for the useful discussions about the subject of the paper.

6. References

[1] Clocksin, W.F., Mellish, C.
Programming in Prolog.
Springer Verlag, 1981.

$