Annales UMCS Informatica AI 5 (2006) 79-86

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Prolog, Mercury and the termination problem

Anna Sasak*

Maria Curie-Skłodowska University, Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Abstract

This paper shortly introduces the two logic programming languages Prolog and Mercury. On this background we introduce the problem of analysing termination of programs. Then we present Mercury's termination analyser, that the authors of the language incorporated into its compiler. We will also discuss the proposition based on the same method analyser for Prolog's predicates.

1. Prolog

Prolog is a logic programming language with its roots in automated theorem proving. Its terminology and concept are based on the first-order predicate logic. There are a few definitions required to understand the rules of programming in Prolog.

Facts are used to describe some relations between the objects and they are always true. They consist of relation name, **predicate** and the list of coma-separated arguments enclosed with brackets. Each fact ends with full stop. For example, the fact expressing the relation of admiration and referring to two objects could have a form *like(kate, flowers).* Names of used object and predicates must start with lower case letters.

The second type of statement in Prolog are **rules.** Rules comprise two parts, a head and a body connected with ':-' (neck symbol also read as 'if') and end with a full stop. Head is a name of the defined rule with the list of its arguments. Body is a set of goals separated with commas which state for logic 'and'. That conditional form means that to satisfy the rule it is required for all its goals to succeed. While defining a rule it is allowed to use **variables** both in its head and body. A variable in Prolog is a string of letters, digits, and underscores beginning either with a capital letter or with an underscore. They are used to replace objects that can not be named at the moment. For example, if we want to describe a person who is a brother of somebody, in Prolog we would use the following form: *brother(X,Y) :- man(X), parents(O,M,X), parents(O,M,Y).*

* *E-mail address*: asasak@poczta.fm

Collection of facts and rules creates a **database.** With this database, we can ask Prolog questions. To answer a question about the database, or in other words, to satisfy all the goals that build that query, Prolog has to search the database. Binding variables always go from left to right, backtracking if necessary. When Prolog finishes satisfying the goals, the answer is given. Extending the previous rule with a few facts the following database is created:

```
man(kriss).
man(mathew).
parents(mathew,anne,kriss).
parents(mathew,anne,maggie).
brother(X,Y) :- man(X), parents(O,M,X), parents(O,M,Y).
```

Now after putting the question

```
?- brother (kriss , melanie).
```

the answer 'No' will be obtained as there are no possible variables binding that would satisfy every goal of '*brother*' relation.

The basic method used in Prolog's programs and data structures is **recursion**. As the definition says: recursion or recurrency in programming and mathematics is calling function on itself. Each recursive definition requires at least one ending, nonrecursive state that would make the recursive calls stop. For example, a predicate that computes factorial in Prolog could have the following form:

```
factorial(1,1):-!.
factorial(X,Y):-X1 is X-1, factorial(X1,S), Y is S*X.
```

First clause is an ending state terminating the recursive calls. Goal written as '!' is called **cut.** It is a goal that always succeeds, but cannot be backtracked. It is used to prevent unwanted backtracking. The second clause is a recursive call with its input arguments properly transformed.

2. Mercury

Mercury is a new, pure logic-functional programming language. Such as other existing logic programming languages, it is a high level programming language which allows programmers to concentrate on solving the problem rather than thinking about low-level dependencies like memory management. The project called Mercury was created at Melbourn Univeristy in 1995. For the past years there has been done a lot of research and development work. As a result, since December 2002, there has been the version 0.11.0 available that supports most of system platforms.

Main features of Mercury:

1. It is purely declarative which means that predicates and functions do not have non-logical side effects.
2. Strong type system.
3. Strong mode system.
4. Strong determinism system which catches many program errors at compile time.
5. Module system.
6. Support for higher-order programming, with closures, currying and lambda expressions.
7. In comparison with the existing logic programming languages Mercury is very efficient.

The syntax of Mercury is based on the syntax of Prolog, although the semantics differs a little. Program in Mercury is a set of modules. Each module is a file that contains sequence of elements – declarations and clauses. Element is a term ended with a full stop. Term is a set of tokens and token is a set of characters. Each module starts with a ':-**module** *Module_name'* declaration which specifies its name. Next, there should be interface section introduced by ':-**interface**' declaration. This section specifies the entities that are exported by this module. An ':-**implemenation**' definition indicates the start of next section. This is the place that must contain definitions for all declarations from the interface section. The module may end with '**:- end_module** *Module_name*' declaration. To make use of entities exported by other modules it is required to explicitly import those modules using ':- **import_module** *Module_list*' or ':- **use_module** *Module_list*' declaration. It is important is that one module must export a predicate '**main**/2' declared as either:

```
:-pred main ( io__state::di, io__state::uo) is det. or
:-pred  main (io__state::di, io_state::uo) is cc_multi.
```

Module arranging some list of operations could look like this:

```
:-module list_operations.
:-interface.
   :-type list ( El ).
   :-pred append( El , list( El ), list(El) ).
:-implementation.
   :-type list ( El ) - - -> []   ;  [ El | list(El) ].
   append( El , [] , [El] ).
   append(El , L1 , [El|L1].
:-end_module list_operations.
```

It is also worth mentioning that there are new kinds of clauses in Mercury, called **functions** or more precisely: function rules and function facts. If the top-level functor of the head of a rule or a predicate is '=/2', that clause is suitable

function rule (having the form: *head=return_value:-body*) or the function fact (having form: *head=return_value*). In both cases head can not be a variable and its arguments must be valid terms.

The type system of Mercury, is based on many-sorted, polymorphic logic. Next to built-in primitive types there are predicate types - **pred**, function types – **func** and universal type **univ**. Apart from those types new types can be introduced with '**:-type**' declaration. There are several categories of derived types, such as: **discriminated unions**, **equivalence types** and **abstract types**.

Discriminated unions are similar to records or enumeration types known from other programming languages. Their declarations have the form: :-**type** *name* [(T1,...,TN)] - - -> body where body is a comma separated the sequence of constructors. For example, the type designed to contain personal data could have the following form:

```
:-type pers_data - - -> pers_data ( firstname :: string , familyname :: string).
```

Equivalence types are declared with '==' operator. They are considered as simplification of the type situated on the right side of the declaration f.e. ***:-type*** *liczba_calkowita == int.*

Types with hidden implementation are called *abstract types*. In the interface section there is only type's name and arguments whereas its definition (list of constructors) is situated in the implementation section.

For each predicate or function it is necessary to explicitly determine which arguments are input and which are output. These two primary modes are called 'in' and 'out' respectively. Mode declaration in the form *:-* ***mode*** *term_name(mode , ... , mode) [= mode]* should be situated in the module interface section, usually just after the corresponding predicate or the function declaration.

For each declared mode there should be appropriate determinism declaration. Term, which call ends not throwing exception, can have one of the following determinism kinds: **det, semidet, multi, nondet, failure, erroneous.**

In Mercury full declaration and definition of predicate appending two lists could look as follows:

```
:- pred append ( list (T) , list (T) , list (T) ).
:- mode append ( in , out , out ) is det.
:- mode append ( out , out , in ) is multi.
:- mode append ( in , in , in ) is semidet.
append ( L1 , L2 , L3) :- (L1 = [] , L3 = L2;
L1 = [G|O] , append(O , L2 ,L33 ),L3=[G|L33].
```

3. Termination problem

Termination problem or **halting problem** in computability theory there is a decision problem that could be expressed as follows: for a program's description and its initial input, determine whether the program, when executed on this

input, ever completes. In 1936 Alan Turing proved that general algorithm to solve the halting problem for all possible inputs cannot exist. It is said that the halting problem is undecidable over Turing machines. There are many methods that are used to analyse termination of programs. I would like to present the Gröger-Plümer approach to the problem.

That method has two stages:

1. Assigning an integer to each clause of analysed predicate that estimates maximum difference between the total size of the output and input arguments.
2. Using the solution from the first stage to check if in each cycle in the predicate's call graph, input arguments decrease in size

The subject of this article is only first stage analysis which involves producing and solving a set of linear inequalities in the form:

$$\Sigma i \epsilon Ipi + \varphi p \geq \Sigma j \epsilon Jpj$$

where p_i stands for the size of argument i predicate p, I is a subset of the set of the input arguments and J is a subset of the set of its output arguments. The aim is to solve for the minimum integer value of φ_p satisfying the inequalities inferred for the predicate.

Let us consider the predicate *append(H1,H2,H3),* which attaches a list to the back of another, and its arguments modes are in, in, out respectively. Starting with the recursive clause, the first stage analysis would have the following course:

```
append (H1, H2, H3) :-
    H1 = [ X | Xs ],
    append (Xs, H2, Zs),
    H3 = [ X | Zs ].            %h3
```

Starting from the right side of the inequality for this clause is output variable H3. On the strength of the last goal it can be replaced by the sum of variables X and Zs and 1 representing the size of cons cell.

```
append (H1, H2, H3) :-
    H1 = [ X | Xs ],
    append (Xs, H2, Zs),        %x + zs + 1
    H3 = [ X | Zs ].            %h3
```

The second goal is a recursive call that also defines one of the active variables Zs. We cannot estimate the exact size of Zs precisely but we know that its upper bound is: $xs+h2+\varphi_a$

```
append (H1, H2, H3) :-
    H1 = [ X | Xs ],            %x + xs + h2 + φa + 1
    append (Xs, H2, Zs),        %x + zs + 1
```

```
    H3 = [ X | Zs ].            %h3
```

First goal states that h1=x+xs+1 so x+xs on the right side can be replaced by h1-1

```
append (H1, H2, H3) :-          %h1 + h2 + φa + 1 - 1
    H1 = [ X | Xs ],            %x + xs + h2 + φa + 1
    append (Xs, H2, Zs),        %x + zs + 1
    H3 = [ X | Zs ].            %h3
```

As a result inequality of the form $h1+h2+\varphi_a \geq h1+h2+\varphi_a+1-1$ is gained which is always true.

The annotation for the other, nonrecursive clause would be:

```
append ( H1, H2, H3 ):-         %h2
    H1 = [],                    %h2
    H3 = H2.                    %h3
```

which gives the inequality of the form $h1+h2+\varphi_a \geq h2$ also always true which finally allows us to estimate φ_a equals 0.

4. Algorithms

A few of Mercury's creators: Chris Speirs, Zoltan Somogyi and Harald Søndergaard have implemented a termination analyzer and incorporated it into the Mercury compiler. Their algorithm is mainly based on the previously described method, however with small modifications. The first stage analysis is based around the inequalities

$$\sum output\_supplier\_variables + \varphi \geq \sum output\_variables,$$

where *output_supplier_variables* represent the set of these input variables whose value contribute to the size of the output variables. The set of output variables from the right side of inequality is calculated by using the fixed point analysis. At first it is assumed that none of the procedures use none of its input arguments to produce output. Then in each of the following steps the output_suppliers set is being completed by M set which is subset of input arguments of p_j procedure. This process is repeated until reaching a fixed point. The code of Stage 1 algorithm is as follows:

For each procedure p€S set new_output_suppliers(p) to []
Do
    old_output_suppliers:=new_output_suppliers
    Step1(*old_output_suppliers*, *new_output_suppliers*)
while *new_output_suppliers* ≠ *old_output_suppliers*
output_suppliers:=new_output_suppliers
If *I* is unsatisfable then for each p€S set φp to ∞

else set (φ1,…, φn) to (d1,...,dn)
where(d1,...,dn) is the least solution satisfying I

Proc step1(in old_output_suppliers, out new_output_suppliers) is
Set *I* to true
For j:=1 to n:
Let B0:-B1,...Bm be the clause defining pj (m≥0)
M:=*outvars*(pj)
δ:=0
For i:=*m* to 1:
If M∩*outvars*(Bi)≠∅ then
M:=(M*outvars*(Bi))U*old_output_suppliers*(Bi)
δ := δ +change(B_{i})
If M⊈*invars*(pj) then for each p∈S set φp to ∞ and exit
new_output_suppliers(pj):=M
If δ contains ∞ then *I*:=false
else I := I Λ φpj ≥ δ

Proposition of algorithm realising Stage 1 analysis in Prolog could have the following form:

```
compare(InVar, OutVar):- …
replace(VarSet, OutVar, InVar, VarSetNew) :- …

step([],Var,Var):-!.
step(Body,Var,VarX):-last(Body,Clause),
    not recursiveCall(Clause),
    replace(Var, outvar(Clause),
    invar(Clause), Var1),
    remove(Body, Clause, Body1),
    step(Body1, Var1, VarX).
step(Body,Var,VarX):-last(Body,Clause),
    recursiveCall(Clause),
    add(Var, Gamma, Var1),
    replace(Var, outvar(Clause),
    invar(Clause), Var1),
    remove(Body, Clause, Body1),
    step(Body1, Var1, VarX).

step1_check(X:-Body):-Var = outvar(H),
    step( Body, Var, VarNew),
    In = invar(H)+Gamma,
    compare(In, VarNew).
```

The presented algorithm's framework has been partially written with some informal language to improve clarity. It assumes that there is information about predicate's modes available and also that predicates are allowed to make recursive calls only to itself. Predicate *step1_check* for each predicate orders to perform stage 1 analysis. Predicate *step* for each goal of analyzed predicate's body converts set of input variables including new active variables. Predicate *compare* for variables sets it gains, builds up proper inequality and returns its solution. Predicate *replace* does some required list operations.

Conclusions

Unfortunately stage 1 analysis is not enough to get a reliable answer whether that predicate terminates or not. To prove it, let us again consider the predicate calculating factorial.

```
factorial ( 1 , 1 ).
factorial (X, Y ) :- X>1 ,
    X1 is X – 1,
    factorial ( X1 , Y1 ),
    Y is Y1 * X.
```

The inequality that is obtained is of the form $x + \varphi \geq x + \varphi$ and it is always true. Yet it is enough to change only one sign

```
factorial ( 1 , 1 ).
factorial (X, Y ) :- X>1 ,
    X1 is X + 1,
    factorial ( X1 , Y1 ),
    Y is Y1 * X.
```

to see that:

1. It changes nothing in the obtained inequality and its solution.
2. Causes that this predicate's call will never terminate.

To draw some further conclusions about termination of a given predicate it is necessary to make stage two analysis.

References

[1] Małuszynski J., Nilsson U., *Logic, Programming and Prolog*, JohnWiley & Sons, (1990).
[2] Colmerauer A., Roussel P., *The birth of Prolog*, (l992).
[3] Apt K.R., *The logic programming Paradigm and Prolog*, (2001).
[4] Clocksin W.F., Mellish C.S., *Prolog programowanie*, Helion, (2003).
[5] The Mercury group. Available from http://www.cs.mu.oz.au/research/mercury.
[6] Speirs C., Somogyi Z., Søndergaard H., *Termination Analysis for Mercury*, Proceedings of the Fourth Static Analysis Symposium, (1997) 157.