Unfolding WAM Code

Michel Ferreira and Luís Damas

LIACC, Universidade do Porto,
Rua do Campo Alegre 823,
4150 Porto, Portugal

E-mail: {michel,luis}@ncc.up.pt

September 9, 1996

We present a Prolog compiler that can unfold WAM code towards obtaining efficient code for Prolog programs. The unfolder implements the principles of efficiency as stated by Van Roy: reduce instruction granularity, exploit determinism, specialize unification. It can operate without requiring abstract interpretation.

1. Introduction

Since the development of the WAM [7] in 1983 by D.Warren, Prolog compilation has been the subject of a great amount of investigation, deriving new optimizations and solutions [6], aiming at leveling the execution time of Prolog programs and the execution time of programs written in imperative languages such as C or PASCAL. However, and in spite of this effort in the compilation of Prolog and of other logic programming languages, there is still a considerable gap between execution time for these languages and for traditional imperative languages.

The major reason for this gap has not to do with the way compilation is done, because we now know how to compile Prolog efficiently. In fact, Prolog execution is a generalisation of the execution of imperative languages, as the WAM has made clear. Then, why such a difference in the execution time for a program written in C or PASCAL and for a program written in Prolog? That reason has to do with what is compiled in Prolog programs, i.e., what is written in Prolog programs, and not in how that compilation is done.

The major advantage that logic programming languages have over imperative languages is that they allow the programmer to say what he wants without having to say how to get it. In an imperative language, the essence of programming is the description, step by step, of the operations to do in order to obtain what he or she wants. In Prolog we program concepts and definitions, leaving the details of its execution to the compiler. The problem of Prolog programs is that the simplicity of writing such concepts and definitions makes the programmer write more general code than what is needed.

Consider the following example: suppose that we wanted a program that concatenates two lists, for instance [1,2] and [3,4], giving the result [1,2,3,4]. A program in an imperative language to do that would be something like that: "take the first element of the first list and copy it to the first element of the output list; continue copying from the first list to the output list until we reach the end of the first list; make the continuation of the output list be the second list".

In Prolog we simply write the definition of concatenating two lists, and write the intended query:

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

:- append([1,2],[3,4],R).
```

The difference between both programs should be clear. Firstly, because of the simplicity of Prolog, the programmer does not need to concern herself or himself on how the program executes. Secondly, because the Prolog definition is more general than the procedural definition. While the program in the imperative language can only concatenate two known lists, the Prolog program can do a lot more, because it has the definition of concatenating two lists. It can do, for instance:

```
:- append([1,2],L,[1,2,3,4]).
```

asking what is needed to concatenate to [1,2] to obtain [1,2,3,4].

However, the imperative language program will be much more efficient, because it has a much more simpler task to perform. And if what we wanted was a program that only concatenates two given lists, why generate a code that does more than that?

The gap between the execution time of imperative languages and Prolog will disapear when we generate *specialised* code for Prolog programs that executes only what the programmer needs to answer the query, instead of having general code that is only used in a certain way.

The traditional approach to this problem in logic programming is to use abstract interpretation [2] to obtain information about execution of program for standard queries. This information then allows program specialisation. Abstract interpretation can take a high percentage of compilation time, and the complexity of logic programs makes obtaining precise data rather difficult, hence sometimes abstract interpretation may not give the precise results one would need.

The solution we propose is based in the following idea: information implicitly present in each WAM instruction, translated in different cases of instructions like `xxxx_var`, `xxxx_value`, `xxxx_list`, `xxxx_const`, etc, allows not only the simplification of execution instruction itself (the reason why it was implemented in WAM), but also the simplification of the next instructions that encode the predicate. Therefore, as go through predicate code we collect the information present in WAM instructions, propagating it from instruction to instruction, and allow the simplification of the execution of the next instructions.

The only information needed is the WAM code of the clauses of the program. Figure 1 shows the WAM code for the program presented above.

Suppose that we were executing the WAM code of the predicate `$exec_1`. From the `put_list(2)` instruction we know that register `X[2]` has type *list*, from instruction `put_list(1)` we deduce the same for register `X[1]`, and from instruction `put_x_variable(3,3)` we know that register `X[3]` is a new variable. When we find the `execute(append/3)` instruction, we can use the data we collected to specialise the call to the predicate, creating a predicate that will directly unify with the types of the arguments of the goal being executed, that we know are of type *list*, *list*, *var*, respectively. This idea has some similarities with what is done in the VAM (Vienna Abstract Machine) [3], where goals and heads were directly unified, with the major advantage that we do not need to create a new abstract machine. We create a new predicate, `append_list_list_var`, whose WAM code is a subset

```
Prolog Code:                                 WAM Code:

append([],L,L).                              append/3: [get_nil(1),
append([H|T],L,[H|R]) :- append(T,L,R).                 get_x_value(2,3),
                                                        procceed],
:- append([1,2],[3,4],R).                              [get_list(1),
                                                        unify_x_variable(4),
                                                        unify_x_variable(1),
Output types:                                           get_list(3),
append(struct,any,any)                                  unify_x_value(4),
                                                        unify_x_variable(3),
                                                        execute(append/3)]

                                             $exec_1:   put_list(2),
                                                        unify_integer(2),
                                                        unify_nil,
                                                        put_list(1),
                                                        unify_integer(1),
                                                        unify_x_value(2),
                                                        put_list(3),
                                                        unify_integer(4),
                                                        unify_nil,
                                                        put_list(2),
                                                        unify_integer(3),
                                                        unify_x_value(3),
                                                        put_x_variable(3,3),
                                                        execute(append/3)
```

Figure 1: WAM code for the `append/3`

of the code of the clauses of predicate `append/3`, in this case being the code of the second clause, as this is the only clause that has a first argument of type *list*. We now unfold this code, `get_list(1)`, `unify_x_variable(4)`, `unify_x_variable(1)`, `get_list(3)`, `unify_x_value(4)`, `unify_x_variable(3)`, `execute(append/3)`, making use of knowing that register `X[1]` has type *list*, register `X[2]` has also type *list*, and register `X[3]` is a new variable, that is, a variable without any references to it.

In various other systems the way to make use of the information deduced by a phase of abstract interpretation was by creating a new abstract machine, more low level than the WAM, dividing a single WAM instruction into several simpler instructions, and then eliminating some of this instructions in special cases detected by the information derived by abstract interpretation.

We do not have a phase of abstract interpretation, but by just propagating information about types of local and global variables, and unification modes, we can derive almost the same information.

Instead of creating a new abstract machine, the way we make use of such information is by complementing WAM instructions with information about the type of the arguments, like `get_list(1,list)`. The execution of such instruction will be much simplified, as we do not have to dereference arg `1`, switch on the case of being *var* or *nonvar*, in the *nonvar* case check if its *tag* is *list*, and finally set `S` to the value of the cell. In `get_list(1,list)` we simply set `S` to the value of register `1`. In an implementation in C of the WAM instruction `get_list(1)`, while the general code would be:

```
if (IsVar(X[1]) d1 = deref(X[1]);
else d1=X[1];
if (IsVar(d1))
{
   trailbind(d1,MakeList((int) H)); H++;
   mode = write;
}
else if (IsList(d1))
     {
         S = (int *) Value(d1);
         mode = read;
     }
     else fail = 1;
if (fail) backtrak();
```

the code for the complemented instruction `get_list(1,list)` would be:

```
S = (int *) Value(X[1]);
```

Not even the mode flag would have to be set, has it will be propagated while unfolding the code, and it will complement the instructions were mode information is relevant.

The `unify` instructions are also simplified by the propagation of information. After the execution of the `get_list(1,list)` we conclude that the unification mode is *read*, and we propagate this information to the next instructions. Then `unify_x_variable(4)` and `unify_x_variable(1)` become `unify_x_variable(4,read)` and `unify_x_variable(1,read)`, and in its execution we don't have to test the current *mode*. Also in an implementation in C, while the general code for `unify_integer(1)` would be:

```
if (mode == read)
{
   d1 = deref(S);
   if (IsVar(d1))
   {
      *((int *) d1) = MakeInteger(1);
      trail(d1);
   }
   else if (d1 != MakeInteger(1)) fail=1;
}
else
{
   *H = MakeInteger(1); H++;
}
if (fail) backtrak();
```

the code for `unify_integer(1,write)` would be:

```
*H = MakeInteger(1); H++;
```

Another important simplification happens in the next instruction, `get_list(3)`, that becomes `get_list(3,var)` because we know register 3 has type *var*, which means it is an uninitialized variable. In this case, we simply copy to the address register 3 points to a new *list* cell:

```
*((int *) X[3]) = MakeList(H);
```

The `unify_x_variable(3)` that appears next is known to be executed in write mode, and so register `X[3]` will have type *var*. The propagation of such information will cause the `execute(append/3)` instruction to be executed with arguments *any*, *list*, *var*. The reason why register `X[0]` has type *any* is because it was read from the heap. This leads to the creation of a new predicate `append_any_list_var`, with a specialised code that makes a `switch_on_term` on register `1`, and then has all the advantages described above, by knowing that argument `2` has type *list* and argument `3` has type *var*, that is, it is to be used as output.

This paper is organised as follows. We first present three state of the art Prolog compilers. We use the `wamcc` compiler as the system to experiment with our ideas. We then present our unfolding algorithm in detail, and present conclusions and future work.

2. Efficient Prolog compilers

2.1. Aquarius

The Aquarius compiler [5] is currently the most efficient complete system in Prolog execution. Its performance is based in four principles:

1. Reduce instruction granularity
2. Exploit determinism
3. Specialize unification
4. Dataflow analysis

The implementation of these principles is quite complex in the Aquarius compiler. A new intermediate language, the BAM was developed to implement the reduce instruction granularity principle. The other three principles were based on a complex dataflow analysis phase.

2.2. Parma

The Parma [4] system was developed by A.Taylor in Australia in parallel with Aquarius. Many of its ideas are similar to the principles used in Aquarius. It compiles into MIPS assembly code.

Parma compilation proceeds in 10 steps. The most important of these steps is abstract interpretation, which is implemented in about 4500 lines of Prolog. The abstract domain is more complex than the one used by Aquarius.

The way the system was developed allowed Taylor to know what information should analysis derive, in order to support the optimizations of the remaining steps.

Both Parma and Aquarius depend by much on their performance on the global analysis phase.

2.3. wamcc

The `wamcc` [1] compiler differs substantially from the two compilers described above. A first difference is the fact that this compiler does not compile Prolog to native code, as do both Aquarius and Parma. It compiles into C code, which can then be compiled using gcc, to generate a executable file.

Another important difference is that it is not as complex. In fact, one of the requirements of this system was extensibility, which implies simplicity in the design of the Prolog engine, without complex optimizations that would excessively enlarge the size of the system. As a consequence, performance is not impressive, as compared with the two systems above.

Because of its simplicity and extensibility, we decided to use the `wamcc` to for our first implementation of the ideas we intended to test. In our usage, `wamcc` was essentally a generic Prolog to WAM compiler.

3. Unfolding WAM code

We now present our approach to obtain an efficient code to Prolog programs, showing that the use of abstract interpretation can be reduced to a minimum, and that only by collecting and propagating information present in WAM instructions we can implement the principles enumerated in the Aquarius compiler.

3.1. Needed information

Before we start unfolding the WAM code, there are three types of information needed:

1. The WAM code for the clauses of the program
2. A list with the output types of the arguments for the predicates (e.g., fib(int,int))
3. A list with the indication of which predicates to unfold

The use of the first information is quite obvious, as it is the subject of our unfolding.

Almost all WAM instructions can give information about their arguments. For instance, after executing an instruction `get_list(1)`, we will know that register `1` will have type *list*, or else execution will have failed. The `call` instruction however can not give any information about influenced variables by its execution, because we don't know what will come out of the execution of a goal. That is why the second information is important, allowing to improve the types of the variables influenced, instead of having to promote all of them to any possible value.

The third and last information is needed because unfolding predicates causes code to enlarge quite importantly, and therefore, to large programs only certain predicates must be unfolded.

The way the second information is derived is by using abstract interpretation on the source program. The algorithm used is quite simple when compared with the ones used in Aquarius or Parma, as it only has to derive information for the outputs. It follows a bottom-up strategy, starting with the facts, and closing the goals in the body of a clause. When a body of a clause has been completely closed, information is propagate to the head, deriving its argument types. This new information will be joined to older information and used in the next passes. The algorithm proceeds until a fix-point is reached.

Third information can be automatically derived, by applying a certain criterium to the predicates. Our criterium was to choose those predicates whose clauses defined cycles. To determine such predicates a graph was created, with the heads of clauses representing points and its goals representing the branches of the graph.

As those clauses define cycles it is more likely that the program will spent more time in its execution, paying off its unfolding. It is sure that this may not happen, but a deterministic criterium could only be based on the execution of the program and in its profiling.

Figure 2 represents the structure of our compiler, and its phases, some of them provided by `wamcc`.

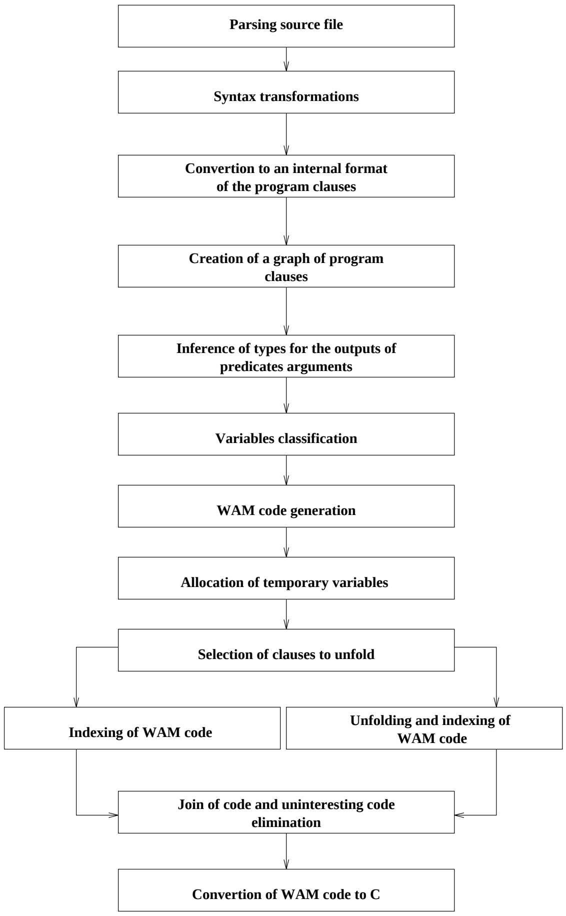

Figure 2: Structure of the compiler

3.2. How unfolding works

When we start to unfold a predicate we do not know anything about its input arguments. It is when we cross the WAM code of a clause that we collect information. We simply look to the WAM instructions that encode a clause, like `get_list(ArgNo)`, or `put_integer(ArgNo)`, or `get_constant(ArgNo)`, etc. In each case we would conclude that the argument `ArgNo` had type *list*, *int* or *const*, respectively.

As we cross the WAM code of clauses, and if those clauses have body, we will find the instructions `call(P/N)` or `execute(P/N)`. If `P/N` represents a predicate that is also on the list of predicates to unfold, it makes sense to use the information collected so far about global and temporary variables an mode of structures. This information allows the simplification of the execution of the goal, because instead of executing it in its general form, i.e., when we not know nothing about its arguments, we can execute a specialised form to the type of arguments that we know this clause is to be executed with. This is the main idea of unfolding: to expand the WAM code of clauses from a predicate into WAM code of clauses specialised to a certain type of input arguments.

As we know the types of arguments of the calling predicates we can use this information together with the types of arguments of the clauses to decide in which argument to index the code. We select this argument by choosing the one that will create less choice-points.

Not only calling instructions are complemented with the refered information. Every other WAM instruction is complemented with this information, allowing it to be crucially simplified. For instance, if when we were crossing the WAM code for a clause we concluded that `ArgNo` had type *nil*, and if we then find the instruction `get_nil(ArgNo)`, complementing this instruction with information about the argument would result in the instruction `get_nil(ArgNo,nil)`, allowing a lot of simplifications (we don't have to dereference `ArgNo`, switch on tag mode, and compare the constant), because this instruction reduces itself to increment `P`. Similar optimizations are possible in other WAM instructions.

Figure 3 gives the Prolog code to the unfolding of a WAM instruction `get_list(I)`.

```
unfold([get_list(I)|CCode],Status0,Code,UPCode,ECode0,ECode) :-
    get_a_mode(I,ModeI,Status0),
    ( ModeI == list ->
        set_s_mode(read,Status0,Status1),
        unfold(CCode,Status1,Code,UPCode1,ECode0,ECode),
        combine(get_list(I,list),UPCode1,UPCode);
    ( ModeI == var; ModeI == ovar) ->
        set_s_mode(write,Status0,Status1),
        unfold(CCode,Status1,Code,UPCode1,ECode0,ECode),
        combine(get_list(I,ModeI),UPCode1,UPCode);
    ModeI == ?  ->
        set_s_mode(read,Status0,Status1),
        unfold(CCode,Status1,Code,UPCode1,ECode0,ECode1),
        set_s_mode(write,Status0,Status2),
        unfold(CCode,Status2,Code,UPCode2,ECode1,ECode),
        UPCode = [switch_on_mode(I,?,UPCode1,UPCode2)]
    ).
```

Figure 3: Prolog code to unfold the instruction `get_list`.

It is clear that unfolding WAM code will give rise to a major code growth, specially if we can't cut a lot of possible values to the arguments like we can when execution of a programa starts by a directive. It is exponential in the number of arguments and the number of values of our domain. To control this code growth we can do three things:

- To limit the number of predicates to unfold
- To use a small domain of types
- To replace uninteresting specialised forms by the general form of the predicate

4. Early results

Table 1 shows the speed-up and code growth of doing unfolding over four simple Prolog programs. The code growth is an important aspect to evaluate, as it can be very large, because a predicate can be unfolded into a large number of specialised predicates. On the other hand, the C code to implement WAM instructions complemented with the information explained, is considerably smaller than the general version. This is why code growth factor in this examples is not very large, and can even be less than 1, as happens with fib, because this program is only specialised into one predicate, `fib(int,var)`.

Program	non-unfolded	unfolded	C Code growth	Description
append	1	5.1	2.1	append two lists of 30 elements
nreverse	1	4.8	2.8	nreverse a list of 30 elements
qsort	1	4.3	5.1	qsort a list of 50 elements
fib	1	2.1	0.7	calculate fib of 100

Table 1: Performance of unfolding: Speed-up and Code growth

5. Conclusions and further work

Unfolding WAM code is quite a nice way of getting efficient code for Prolog programs. It implements the fundamental principles of efficiency described in Van Roy, without having to perform any complex tasks. Reducing instruction granularity is achieved not by creating a new abstract machine but by complementing WAM instructions with information that is generated.

The exploitation of determinism is achieved by replacing general calls to predicates by specialised calls, and by a more intelligent indexing system.

The fact of having information about variables and structures modes allows the simplification of unification.

The use of abstract interpretation to support these optimizations has been reduced to a minimum, resulting in a quite simple algorithm.

In the future we plan to develop our system so that it will be able to run real applications, evaluating the results of unfolding in the execution of such applications.

Acknowledgments

This work has been partly supported by JNICT under the PROLOPPE project (contract Praxis/-3/3.1/TIT/24/94). Michel Ferreira thanks PRAXIS (Grant Praxis/2093/94) for supporting his research.

References

[1] Philippe Codognet and Daniel Diaz. wamcc: Compiling Prolog to C. In *12th International Conference on Logic Programming*. MIT Press, 1995.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[3] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *International Workshop in Programming Language Implementation and Logic Programming (PLIP'90)*, number 456 in Lecture Notes in Computer Science, pages 121–135. Sweeden, Springer-Verlag, 1990.

[4] A. Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, June 1991.

[5] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[6] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.

[7] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.