A Piggy-back Compiler For Prolog

J. L. Weiner
S. Ramakrishnan
Computer Science Department
University of New Hampshire

March 18, 1988

1 Introduction

In this paper, we describe[1] the design and implementation of an optimizing Prolog compiler. Our approach is novel both for the breadth of techniques employed and the performance we achieve as a result of using those techniques. Furthermore, we know of no other compiler for Prolog that utilizes the same techniques and achieves comparable performance. In addition to efficiency, the design goals were *portability* and *simplicity*. Our compiler takes advantage of knowledge about what it is compiling to reduce the mechanisms needed to support its runtime evaluation. The types of knowledge it uses are:

- syntactic types and the directionality of the procedure arguments (mode information), allowing assignment and efficient pattern matching to replace unification, and
- whether procedures are deterministic, since non-deterministic procedures require extensive overhead.

In a break with tradition, we have chosen to compile into the C language, rather than some abstract machine language. The choice of the C language as our target language satisfies both the goals of portability and simplicity. Although compiling into an abstract machine language, provides a degree of portability, the back-end still has to be written for a new machine. In contrast, for all practical purposes, the back-end for our compiler (a C language compiler) already exists for most machines (satisfying our goal of portability). Our goal of simplicity is also satisfied by the choice of the C language because the C compiler will take care of the most difficult part of writing a compiler which is trying to take advantage of the underlying architecture of the target machine. Furthermore, most of the optimizations that are done are, in fact, performed at the source level, i.e., at the Prolog source code level. This means that there is nothing inherent in a given intermediate representation that makes it superior to ours.

2 Related Work

Much research has been done in the area of Prolog compilation. The first compiler (the DEC 10 Prolog compiler) was written at the University of Edinburgh by Warren and his colleagues [BOWE 82]. Written entirely in Prolog, the DEC 10 Prolog compiler introduced many optimization techniques for Prolog. Among the techniques used are: compiling unification into machine language, fast access to clauses within a procedure and reductions in space requirements.

[1]The assistance of Profs. Robert Russell and Philip Hatcher throughout this project is gratefully acknowledged.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

The next generation in the Prolog compilation area was ushered in when Warren compiled Prolog into a specialized intermediate language [WARR 83]. This intermediate language is so designed that both Prolog's control and data structures map in a straight forward manner to a sequence of instructions in this intermediate language and forms the basis of what we call the Warren Abstract Machine (WAM, for short). WAM's success lay in the fact that it was not only well-designed but also very portable and formed the basis for most subsequent work.

Following an approach closely related to the WAM, Mellish translates Prolog into a low-level intermediate language called POPLOG [MELL 84]. Mellish's approach differs in one significant aspect from Warren's. Mellish argues that instead of building a machine onto which Prolog could be easily mapped, it might be more fruitful to look at a subset of Prolog procedures that do not require all the machinery of a Prolog machine, such as the WAM, and that can be directly mapped onto a conventional machine. Mellish observes that most Prolog programs are both directional and deterministic and suggests techniques for inferring these properties by performing a static analysis of the Prolog programs. He also shows how knowledge of these properties can lead to generation of efficient code for a conventional machine.

Nilsson [NILS 83] first suggested that one could use the machinery provided by a higher-level language to compile Prolog and suggested a translation to Pascal. Nilsson implements non-determinism by continuation-passing and looks for the same features of the Prolog program on which to perform some optimizations, i.e., direction (or, mode) and determinacy. Nilsson addresses the three problems in compiling domain-based Prolog: Compilation of non-determinism (clauses), compilation of terms into appropriate data structures and the compilation of unification.

Bruynooghe [BRUY 86] also suggests a translation to Pascal. However, his suggestions are in the context of efficient garbage-collection. He discusses how mapping Prolog constructs to Pascal would prove that garbage-collection in Prolog could be made as efficient as in conventional programming languages.

Our approach is very similar to that followed by Nilsson and Mellish. However, even though Nilsson proposed a translation to Pascal, he has not implemented it. Furthermore, Nilsson dismisses backtracking (and the related issues) as trivial problems; we have found that the implementation of backtracking to be a very important issue, and it is only a careful design and implementation of backtracking that enables the performance improvements outlined later. We differ from Mellish more in form than in content by departing from the WAM model of compilation. The performance improvements we obtain justify the soundness of our approach.

3 Efficiencies exploited by compilation

Our compiler is aimed at improving the efficiency of a Prolog program by providing an efficient implementation of backtracking, by processing deterministic clauses efficiently, by exploiting parameter directionality and by introducing typed arguments into Prolog.

Backtracking: Backtracking is the process by which Prolog searches for all solutions to a query. This implies that a Prolog procedure can return many values. Such a procedure is said to be non-deterministic. We implement non-determinism using a continuation-passing method following Mellish [MELL 85]. In this approach, a procedure indicates success by invoking the next procedure to be executed; it returns only on failure. While the continuation-passing approach is very easy to implement, it can be very slow (we suspect that this is, perhaps, the reason for its limited use). We have a particularly efficient implementation of this backtracking process (using some assembly-coded routines) that exploits the nature of the *calling sequence* of the underlying machine.

Deterministic Procedures: Procedures that are deterministic (that is, those that are uni-valued) can be compiled into a conventional C procedure. Accordingly, such procedures use less space and execute faster since they do not generate multiple solutions. Our compiler performs this optimization for procedures that the user explicitly specifies as being deterministic.

Directional Parameters: Another contributing factor to Prolog's inefficiency is a result of the notion of "multi-directional" arguments to procedures; that is, unlike conventional languages, there is no notion of input and output arguments. To handle multi-directional arguments requires calls to expensive unification routines. If we know which arguments are input and which are output, we can generate more efficient code that uses simple assignments and comparisons, instead of calls to unification routines.

Data Typing: Another important feature of our approach is the introduction of *data types* in Prolog. We call these *domains* after Nilsson [NILS 83] partly in order to distinguish them from data types in the target language, and partly, to emphasize the abstract nature of the type concept. Many of the optimizations that we perform are due to Luca Cardelli [CARD 84]. Cardelli exploits the strongly-typed nature of ML in order to compile unification efficiently and to enable quick clause selection. We follow a similar approach by augmenting our Prolog programs with `type` annotations.

4 Language Supported

The compiler supports standard Prolog augmented with annotations. These annotations are used by the compiler in order to optimize performance. The annotations fall into four classes:

1. *Domain declarations.* These are declarations that allow the user to specify the types of the objects used in the program. Thus, the user can declare a `list` data type, which consists of a list of integers (say), with the following declaration:

```
type((list :- l(int,list),
        empty)).
```

   Here `l` is the functor of the `list` data type. If the `list` is empty, it is represented by the distinguished functor `empty`. Otherwise, the `list` must have a functor `l`, and it must have two arguments: the first must be of type `integer` and the second is of `list` type again. Notice that the functors must be unique; that is, given a functor we must uniquely be able to identify the type defined by that functor. Furthermore, types can be defined in terms of other types. Thus, the above domain declaration is equivalent to the following declarations:

```
type((w :- empty)).
type((list :- l(int,list), w)).
```

   The only difference now is that the distinguished functor `empty` is of type `w`. These domain declarations are converted mechanically into equivalent C structures that are accessed by the C program generated by our compiler.

2. *Predicate type definitions.* These type definitions specify for each predicate what the types of the arguments are. For example, given the above definition of the `list` data type, suppose we write a procedure called **append** that takes two lists and produces a third list. Then the predicate type definition for the predicate **append** will be:

```
type(append(list,list,list)).
```

   There must be one such declaration for each procedure that the user wishes to compile.

3. *Mode declarations.* These declarations specify the directionality of the arguments of a predicate. The mode value can be one of ⟨free, ground, part-ground⟩. A mode value of "free" indicates that the corresponding argument is an *output* parameter; the "ground" parameter indicates that it is an *input* parameter. A mode with a value of "part-ground" indicates that the mode is unknown. Optimizations can only be performed in the first two cases. If the predicate does not have a mode declaration, all its arguments are assumed to be "part-ground." For example, given the procedure **append** to concatenate two lists (that appear in the first two argument positions) to produce the list (in the third argument), mode declaration looks like the following:

```
mode(append(ground,ground,free)).
```

4. *Determinacy declarations.* These declarations provide information about the determinacy of

the predicates. The value for the determinacy of a predicate is either "true" or "false." If the predicate has no determinacy declaration, a value of "false" is assumed; that is, in the absence of better information, the predicate is assumed to be non-deterministic. Assuming that the predicate **append** is deterministic, the following declaration would be made:

```
det(append,3,true).
```

5 Clause Indexing

An important use of the predicate type definitions is related to the notion of *quick clause selection* or *clause indexing*. Given a set of clauses, Prolog performs a unification with the head of each clause starting with the first, and going on to the next clause only if the unification fails. If we know that a certain argument position is always "ground," we could use the functors that constitute the terms for that position as clause selectors. To illustrate this idea with an example, consider the following definition of **append**.

```
type(append(list,list,list)).

append(empty,L,L).
append(l(X,Y),L2,l(X,L3)) :-
        append(L1,L2,L3).
```

Assume that the `list` data type has been defined earlier; further assume that the mode declaration for **append** is `mode(append(ground, ground, free))`. Then, since we know that the first argument position is "ground," we can generate code that looks like the following:

```
L1:
      if (arg1->tag != empty)
          go to L2;
      <code for clause 1>
L2:
      if (arg1->tag != l)
          go to L3;
      <code for clause 2>
L3:
      return (FALSE);
```

Notice that this code consists of simple conditionals instead of calls to a general unification routine. If the tag field is neither **empty** nor **l** we simply report failure. The only catch here is that we need to know the modes. If the modes were unknown, we could not make this optimization, and would have to resort to unification. If two or more columns are "ground," we have to choose one. There are well-known techniques that enable us to make this choice [CARD 84] and they will not be discussed here.

6 Code Generation

The code generation phase can be split up into 2 parts:

1. Head Processing, and
2. Body Processing.

Head Processing Head processing is the main step in the code generation process and involves unifying the head and selecting the clause. We process each argument of the head individually. Each argument in the head can be either a variable, a constant or a structure and can have one of three modes. Thus there are nine cases that must be handled when processing an argument. When the argument is free, the term corresponding to that argument position is *created*; that is, no unification needs to be performed. Instead, we merely generate assignment statements, whenever necessary. In the case of a "ground" argument, we know that the argument exists. Therefore, we generate "if-then" conditional statements, since we know that the argument must exist. It is only in the case of the "part-ground" position that we need to generate unification code. Even in this case, since we know the syntactic types, we generate calls to unification routines for a *specific type*; this is more efficient than a general unification of two arguments of arbitrary types.

Processing the Body Processing the body is much simpler than head processing, since no unification needs to be done. All that the body processor does is to call the subgoals with the appropriate arguments. The only complexity arises as a result of

the way we implement non-determinism using the technique of *continuation passing*.

We have a particularly efficient implementation of the continuation passing mechanism. Our implementation uses two primitives called **freeze** and **melt**. **Freeze** is a procedure (coded in assembly language) that takes as its arguments the name of a procedure, the number of its arguments, and its continuation pointer, followed by a list of the arguments, and returns a pointer to a location in memory at which the procedure and its arguments are stored. This pointer is then passed along. When **melt** is invoked with that pointer as its argument, it executes the procedure at that location with those arguments.

Thus, the way we process the body is to start from the last subgoal and work our way up to the first subgoal. Each subgoal from the last to the third (counting the head as the first subgoal) is frozen. Each subgoal is passed a continuation pointer to the next subgoal, so that we thread the goals together. We then call the second subgoal with the appropriate continuation. Let us illustrate this with an example. Suppose we have the clause:

```
a(X,Y) :- b(X,Z),c(Z,W),d(W,Y).
```

The code we generate is

```
<code for the head>
c1 = freeze(d,cont,2,W,Y);
c2 = freeze(c,c1,2,Z,W);
b(c2,X,Z);
```

"cont" in the first **freeze** call is the continuation pointer associated with the procedure **a**. When (and if) **b** succeeds the procedure at its continuation is invoked. This is **c**. Now **c** gets as its continuation **c1**, which is the pointer to the function **d** and its arguments. When (and if) **c** succeeds, it invokes the procedure at its continuation (which is **d**). Finally, when (and if) **d** succeeds, it invokes the procedure at its continuation **cont**, which is the procedure that is executed after **a** succeeds. If (for example) **c** fails, an alternative clause for **b** is tried (assuming that there is one), since a procedure returns on failure.

7 Optimizing Deterministic Predicates

A deterministic procedure in Prolog returns only once. If it is ever reinvoked (as a result of backtracking) it will always fail. Thus, it is as if a deterministic procedure has an "implied cut" as the last subgoal of each of its defining clauses. A deterministic procedure, then, conforms more to our notion of a standard procedure, since there is no backtracking. Within the context of our approach, this means that deterministic procedures do not contain the **freeze** and **melt** primitives.

The compiler currently assumes that if the predicate being compiled is deterministic, all its subgoals are. It also assumes that if any one of the subgoals in the body is deterministic, each and every subgoal is deterministic. Although this rules out compiling clauses in which the body consists of both deterministic and non-deterministic subgoals, the same effect can be easily achieved. This is done by combining sequences of consecutive deterministic goals into one non-deterministic clause, whose head consists of a non-deterministic predicate, and whose body consists of only deterministic goals. A simple preprocessing step prior to start of compilation can perform this conversion. Let us illustrate this with an example. In this example predicate names that begin with "d" are deterministic; similarly, predicate names that begin with "n" are non-deterministic. Consider the clause:

```
na(X,Y) :-
nb(X,Z),dc(Z,Y),dd(Y,W),ne(Y).
```

This clause can be rewritten as the following two clauses:

```
na(X,Y) :-
nb(X,Z),ncd(Z,Y,W),ne(Y).
ncd(Z,Y,W) :- dc(Z,Y), dd(Y,W).
```

Thus, the method consists of building up a dummy clause, with a non-deterministic head and an entirely deterministic body (consisting of the sequence of deterministic subgoals). This clause acts as a non-deterministic interface to an essentially deterministic clause. This clause indicates its success by **melting** its continuation, but there is no necessity

of **freezeing** any of the subgoals in the body. Now both clauses obey the restrictions imposed by our compiler. Even though this simple transformation of combining deterministic goals is not performed by our compiler, it is a relatively simple process and can be easily implemented.

A deterministic procedure is compiled as a sequence of calls. Since each procedure returns a truth value, the failure of any of the subgoals means that the predicate fails. Thus, the clause

```
a(X) :- b(X,Y), c(Y,Z), Z is Y + 1.
```

is compiled into the following code:

```
a(X)
...
{
    <code for the head>
    if (! b(X,Y))
        return (FALSE);
    if (! c(Y,Z))
        return (FALSE);
    if (! eval_proc1(Z,Y))
        return (FALSE);
    return (TRUE);
}
```

Notice that none of the calls to the various procedures has any continuations passed to it. This shows that the class of deterministic procedures can be mapped onto a conventional control structure.

8 Results on Benchmarks

The performance of the compiler was tested on eight benchmarks [WARR 77], and the results are summarized in the tables below. For a detailed description of the tasks that each of these benchmarks achieve and the reasons for including them see [WARR 77]. All timings are in milliseconds, and represent the average execution times over a large number of iterations (100,000) of the program using the UNIX[2] utility **getrusage** running on a VAX-11/780.

[2]UNIX is a trademark of Bell Laboratories.

Task	A	B	C	MI	MDI
nrev30	69.43	63.83	56.71	8.07	18.32
qsort50	130.02	126.23	126.23	2.92	2.92
serialize	97.27	93.96	93.96	3.41	3.41
query	460.10	446.38	446.38	2.99	2.99
times10	9.70	9.58	8.07	1.24	16.80
divide10	11.80	11.16	9.76	5.42	17.29
log10	3.50	3.35	2.55	4.29	27.14
ops8	6.02	5.67	5.15	5.81	14.45

The versions are:

A Neither mode nor determinacy annotations are supplied. This means that unification routines have to be invoked and none of the optimizations outlined can be performed.

B Only mode declarations are supplied. In this case, calls to unification routines are reduced; however, since there are no determinacy declarations, all predicates are assumed to be non-deterministic, and hence require calls to **freeze** and **melt** routines.

C Both mode and determinacy declarations are supplied. In this all possible optimizations are performed.

MI This represents the percentage improvement in performance as a result of supplying the mode declarations. Simply put, it represents the improvement that results from saving calls to the unification routines.

MDI This represents the percent improvement when we know both the modes and the determinacy of the predicates.

Column MI represents the *percent improvement* in performance due to the addition of mode declarations. We can see that the improvement in performance ranges from 1.24 % to 8.07 %. Column MDI represents the *percent improvement* in performance due to the addition of *both* mode and determinacy declarations. This column therefore represents the maximum possible improvement in performance. Accordingly, we see that the gains in performance are, in fact, very impressive. The improvement in performance ranges from 14.45 % to 27.14 %. These are substantial gains.

The above times are conservative; they include calls to **malloc** as part our memory management

component (which is notoriously slow) and represent timings on programs compiled with the UNIX portable C compiler, not known for producing quality code.

Another interesting test is to see the effect of the C compiler on the C code generated. It is our claim that given a better C compiler the quality of the executable image can be substantially improved. The following table summarizes the results of running the same tests on a VAX 8650 with two different C compilers. Column CC represents the times for execution using the UNIX C compiler with the `-O` switch turned on. Column VCC represents the times using the VAX C compiler (again with the `-O` switch on). Again the timings are in milliseconds. Column CI represents the *percent improvement* in performance as a result of using a different C compiler.

Task	CC	VCC	CI
nrev30	16.00	12.90	19.375
qsort50	28.60	25.04	12.45
serialize	20.80	18.49	11.11
query	82.40	74.67	9.38
times10	2.20	1.71	22.27
divide10	3.00	2.00	33.33
log10	1.00	0.33	67.00
ops8	1.6	0.96	40.00

Looking at the table we can see that the performance improvements as a result of having a better C compiler are in fact fairly substantial (ranging from 9 % to 67 %).

9 Conclusions

One of the principal reasons for undertaking this project was to prove that Prolog could be mapped onto a conventional architecture. We have realized this goal by mapping Prolog onto an abstract C machine.

Prolog with the annotations (especially when the predicates are deterministic, directed and well-typed) has a control flow that is similar to that of conventional programming languages. In this special case, the performance gains are very impressive (upto 27.14 %). This can be thought of as representing the front-end improvement.

We have seen in the previous section that we can obtain fairly significant performance improvements as a result of using a better C compiler. The back-end improvement as a result of having a better C compiler ranges from 9 % to 67 %. This vindicates our original design decision to compile to C and let the C compiler handle the back-end. The average total improvement in performance as a result of optimizations on the front-end and the back-end (obtained by adding the **MDI** and **CI** columns and averaging) is 39.7 %, which represents a fairly substantial improvement in performance.

We have not found the writing of the annotations to be overly restrictive on the Prolog programmer. In fact, the user is usually aware of the directionality of his predicates; likewise, the user is usually aware of whether his predicates are expected to be multi-valued. More importantly, the user is *not* constrained to using "cuts" to specify the determinacy of his predicates. Our experience indicates that "cut" usage is unclear at best. It seems that an explicit declaration to specify whether a certain predicate is deterministic is a better method. These annotations can, in fact, be inferred [MELL 85]; this would remove the burden of providing these declarations from the user. However, as mentioned before, providing these annotations is useful both for redundancy and when these inference techniques fail. Furthermore, the inference techniques are very costly; therefore, one would like to use them as sparingly as possible. The ideal scenario may, in fact, be a combination of user-provided annotations, together with some that need to be inferred; the number of annotations that need to be inferred will then be small.

Compiling to C has the disadvantage of imposing a two-step compilation process: first, compile Prolog to C, and then compile the C code. We do not feel this to be a major disadvantage since Prolog programs can be interpreted. This means that the user can test his programs using the interpreter, and only when he is convinced of their correctness does he need to compile his program. Notice that the annotations that are supplied with the program have no effect as far as the interpreter is concerned.

The compiler has been written entirely in Prolog. The power of Prolog as a language that enables the rapid prototyping of large software sys-

tems is unquestioned. The very power and brevity that Prolog programs provide are (simultaneously) its strength and weakness. What our research has shown us is a way of maintaining the strengths of Prolog while minimizing its weaknesses. If we could combine the ease of programming (that Prolog provides) with the efficiency of a procedural language, we would have created a sophisticated *multi-language* programming environment.

10 Postscript

Although the preceding results show that we compare very favorably with at least one WAM-based compiler, the Mellish compiler [MELL 84], we decided to compare the quality of code generated by our compiler against another well-known compiler, the Quintus [3] compiler. This time the results did not prove quite so favorable. In fact, naïve reverse, compiled with the Quintus compiler, ran almost four times as fast as it did with ours.

In trying to understand this discrepancy, we first replaced our memory manager with another more suitable one. This cut our time in half, but we were still twice as slow as the Quintus version. After trying to speed up our compiler a bit more, with not much success, we took another approach. Rather than speed up our compiler blindly, we decided to understand the factors that made the Quintus compiler produce more efficient code. Apart from pointing out to us the areas where our compiler could be improved, this would also enable us to identify issues that need to be addressed when designing optimizing compilers for Prolog.

The only optimizations that we do not currently perform are: 1) no list optimization and 2) no tail recursion optimization. Could the effect of performing these optimizations result in the object code executing twice as fast? To test this hypothesis, we took the naïve reverse and altered it so that lists were now represented as general Prolog terms and the predicates were no longer tail recursive. This resulted in the Quintus compiler producing object code that ran slower than the object code we produced.

What do these experiments suggest? They verify the correctness of our approach and reinforce our belief that the best optimizing Prolog compilers perform all possible optimizations on the Prolog source code, and therefore the choice of the intermediate representation is arbitrary. These experiments also suggest areas for improvement in the performance of our compiler.

While it is certainly pleasing to arrive at favorable conclusions by performing "empirical" timing experiments, it is also important to remember that performing experiments using benchmarks is an art, not a science, and that any results proven/disproven as a result of their use, should be viewed with caution.

[3] Quintus is a trademark of Quintus Computer Systems, Inc.

We are grateful to Bob Keller and Quintus Computer Systems for loaning us a copy of their compiler.

References

[BOWE 82] Bowen, D.L., Byrd, L., Pereira, F.C.N., Pereira, L.M., Warren, D.H.D, DEC System-10 Prolog User's Manual, Department of Artificial Intelligence, University of Edinburgh, 1982.

[BRUY 86] Bruynooghe,M.,"Compile Time Garbage Collection," Report # 43, Dept. Computerwetenschappen, Katholieke Universiteit Leuven, Apr. 1986.

[CARD 84] Cardelli,L.,"Compiling a Functional Language," *Proceedings of the ACM*, March, 1984, pp. 208–217.

[MELL 84] Mellish, C., & Hardy,S.,"Integrating Prolog into the POPLOG environment," *Implementations of PROLOG*, J.A. Campbell (ed.), 1984, pp. 147–162.

[MELL 85] Mellish, C.,"Some Global Optimizations for a Prolog Compiler," *The Journal of Logic Programming*, **2**:1, 1985, pp. 43–66.

[NILS 83] Nilsson, Jørgen Fischer,"On the Compilation of a Domain-Based Prolog," *Information Processing*, R.E.A. Mason (ed.), Elsevier Science Publishers

B.V. (North-Holland), 1983, pp. 293–299.

[WARR 77] Warren, D.H.D.,"Implementing Prolog: Compiling Predicate Logic Programs," Research Report # 39, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

[WARR 83] Warren,D.H.D.,"An Abstract Prolog Instruction Set," Technical Note # 309, AI Center, SRI International, October 1983.