SOME GLOBAL OPTIMIZATIONS FOR A PROLOG COMPILER

C. S. MELLISH

1. INTRODUCTION

This paper puts forward the suggestion that many PROLOG programs are not radically different in kind from programs written in conventional languages. For these programs, it should be possible for a PROLOG compiler to produce code of similar efficiency to that of other compilers. Moreover, there is no reason why reasonable efficiency should not be obtained without special-purpose hardware. Therefore, at the same time as pursuing the goal of special hardware for running PROLOG programs, we should be looking at how to maximize the use of conventional machines and to capitalize on developments in conventional hardware. It seems unlikely that conventional machines can be efficiently used by PROLOG programs without the use of sophisticated compilers. A number of possible optimizations that can be made on the basis of a static, *global* analysis of programs are presented, together with techniques for obtaining such analyses. These have been embodied in working programs. Timing figures for experimental extensions to the POPLOG PROLOG compiler are presented to make it plausible that such optimizations can indeed make a difference to program efficiency.

2. WHAT ARE REAL PROLOG PROGRAMS LIKE?

It is unfortunate that very little time has been spent on studying what kinds of programs PROLOG programmers *actually write*. Such a study would seem to be an important prerequisite for trying to design a PROLOG optimising compiler. This section gives some arguments suggesting that a great many PROLOG programs do not make full use of the flexibility that a logic programming language might offer. These arguments are reinforced by the extent to which we have been able to automatically detect parts of programs which use a restricted set of the possible facilities. We suggest that large parts of PROLOG programs are *deterministic* and *directional*. Partly this is because PROLOG itself is limited as an implementation of logic.

Address correspondence to C. S. Mellish, Cognitive Studies Programme, University of Sussex, Falmer, Brighton, United Kingdom.

52 Vanderbilt Ave., New York, NY 10017 0743-1066/85/$03.30

In logic, it is frequently the case that there are alternative ways of attempting to show that some theorem follows from a set of axioms. This corresponds to the notion of nondeterministic procedure in a logic programming language. PROLOG implements nondeterminism by a very simple depth-first search strategy with chronological backtracking, and yet it is well known [4] how inefficient this strategy can be. It would hardly be a good strategy for a PROLOG programmer to use PROLOG's search mechanism to attack large search problems. Indeed, many projects have tried deliberately to avoid using the PROLOG search mechanism, in favour of more intelligent strategies (see e.g. [10],[1],[6]). These strategies have in general been expressed as PROLOG programs which (in terms of the underlying depth-first search) are largely deterministic. In addition, there are many computational problems that are not necessarily most naturally conceived of in terms of nondeterministic specifications. Thus, although nondeterminism may be used as a local control structure (for small "generate and test" loops, for instance), it is unlikely that it is used widely in many programs.

In logic, axioms involving a predicate can be used equally well to prove ground sentences and existentially quantified sentences involving that predicate. In a logic programming language, this corresponds to the notion of a multidirectional procedure—a procedure where there is no distinction between inputs and outputs, the unknown values being computed from the known values in whatever way is required. Although there are well-known examples of PROLOG programs that are multidirectional, there are a number of reasons why in general it is hard to write such programs in PROLOG. Firstly, PROLOG's simple selection function and depth-first search strategy mean that it is easy for programs to get into infinite loops if they are used in unexpected directions. For example, the naive reverse program shown below (with cuts removed) will successfully compute in the second argument the reversed form of a list already given in its first argument. However, it will get into an infinite loop if asked to answer the same query but with the list given in the second, rather than the first, argument. The second reason why it is hard to write multidirectional procedures in PROLOG is that most of the system's built-in predicates (such as those for carrying out arithmetic and arithmetic comparisons) are directional, and this directionality is inherited by those procedures which call them, directly or indirectly. The myth of multidirectionality for PROLOG programs has been discussed by McDermott [5]. It would seem to be a good strategy for the PROLOG programmer to have a small repertoire of multidirectional procedures, but to build programs largely with some specific direction in mind.

If these arguments stand up, the control structure of many parts of PROLOG programs will be not dissimilar to that of conventional programs. Of course, the benefits from the absence of side effects, the use of the logical variable, the declarative semantics, and so on still favor the use of PROLOG for these tasks. It would be pleasant if our PROLOG compilers could produce code of comparable efficiency to conventional languages for those parts of our programs that have conventional structure. This should not, of course, prevent us from using the other features of PROLOG where we feel them to be appropriate.

There are independent reasons for arguing that many programs of the future will have to be multilanguage programs [9]. Another reason for attempting to find parts of PROLOG programs that are like conventional language programs is that we can

as a result have efficient PROLOG systems running on machines capable of supporting other languages.

3. AN EXAMPLE OF STATIC ANALYSIS—DETECTING DETERMINACY

A PROLOG program can be viewed as a set of equations specifying for each predicate what instances can be inferred. One strategy for proving things about PROLOG programs is to examine what the corresponding equations look like if we restrict our attention to a more abstract domain than this, such as a domain where the only significance of a predicate is whether *any* instance can be proved or whether any instance can be proved such that the first two arguments are the same. Examining what can be "computed" in this domain will in general be simpler than examining what can be computed in the original domain. However, it enables us to draw conclusions about some aspect of how the actual program will operate and the situations it will be faced with. This notion of using *abstract interpretations* to prove properties of programs has been used successfully with other languages [3, 11, 13].

One useful property that is worth determining to enable compiler optimizations to be made in PROLOG is *determinacy*. We will use the term "determinate" to describe a predicate when its PROLOG definition in a program and the ways it is called in that program mean that it is never possible for a goal involving that predicate to return more than one possible solution. Goals may succeed or fail, but will never be able to backtrack to find alternative solutions. The presence of cuts in a PROLOG program is a clue to possible determinacy; other clues may be available if nontrivial mode declarations have been made (see below). A simple rule to determine whether a predicate is determinate is as follows:

Predicate P is determinate if:

each clause apart from the last includes a "cut" as a conjunct,

each predicate which occurs not before a "cut" in one of the clauses is itself determinate.

For system predicates it is in general known in advance whether they are determinate or not, and this will apply whatever program they appear in. The rule given is weak—anything that passes the test will indeed be determinate, and yet there may be predicates in a program which are determinate for other reasons. However, in practice it allows us to determine that a number of predicates in programs are determinate. Consider the following version of "naive reverse":

```
nrev([X|Y],Z) :- !, nrev(Y,Z1), append(Z1,[X],Z).
nrev([],[]).

append([X|Y],Z,[X|Y1]) :- !, append(Y,Z,Y1).
append([],X,X).
```

If we use the notation <X> to stand for the determinacy of the predicate X (a truth value) then, using the above rule as the only criterion for determinacy, we obtain the

following equations:

```
<nrev>   = <nrev> and <append>
<append> = <append>
```

where **and** is logical conjunction. We require to find values for **<nrev>** and **<append>** which satisfy this equation. In the absence of conflicting evidence, we would like to assume that each predicate is determinate (for if a predicate is not, we will know for certain). That is, we would like to derive a solution to the equations which is minimal with respect to the following (trivial) lattice:

```
FALSE
  |
TRUE
```

One way to derive the least solution is to perform an iterative process, where each unknown starts off with the least value **(TRUE)** and at each stage rewrites itself to the result of evaluating the right hand side of its equation using the values of the previous stage. This is bound to converge, since conjunction is monotonic and the lattice is finite. For the above program, the process has already converged at the first iteration (all predicates are determinate). For the program

```
human(X) :- mother(X,Mother), human(Mother).

animal(X) :- human(X).

mother(fred,jane).
mother(abel,eve).
```

the equations are

```
<human>  = <mother> and <human>
<animal> = <human>
<mother> = FALSE
```

and it takes three iterations for the nondeterminacy of **mother** to propagate through the program. In the initial state, the values of the unknowns are as follows:

```
<human>  - TRUE
<animal> - TRUE
<mother> - TRUE
```

After successive application of the equations the values are

```
<human>  - TRUE
<animal> - TRUE
<mother> - FALSE
```

```
<human>  - FALSE
<animal> - TRUE
<mother> - FALSE

<human>  - FALSE
<animal> - FALSE
<mother> - FALSE
```

and at this point the system has converged (this time, all predicates are inferred to be nondeterminate).

The rule for determining determinacy can be improved by taking into account mode declarations (see Section 6). For instance, if in this program it is known that the mode of `mother` is `mother(+,-)`, then `mother` is clearly determinate because the first arguments `fred` and `abel` in the heads of the clauses are incompatible. We have a working program for inferring determinacy which works as described and with this extension. If, as we have claimed, large portions of PROLOG programs are actually deterministic, one might hope that these automatic methods will uncover a large number of determinate predicates in practice. Indeed, when presented with Warren's benchmark programs, our system infers that 11 out of 16 predicates are determinate (using automatically generated mode declarations). Of the remaining five, three predicates will indeed generate multiple solutions. The other two are in fact determinate. The system is unable to infer this because of one inadequate mode declaration. One extra cut or a user-supplied mode declaration would suffice for the program to infer all the determinate predicates. For a simple natural-language generation program [8], the system infers that 89 out of 139 predicates are determinate, and for a version of the Chat80 program [17] it can detect determinacy for 132 out of 446 predicates.

We have spent some time on the relatively simple problem of showing determinacy because the way that we prove other properties of PROLOG programs has the same basic pattern. First, an abstract domain is chosen and the program is converted into equations with unknowns ranging over this domain. The aim is to find the least fixed point of these equations. This is achieved by iterating over the equations, with each unknown starting off at the least possible value. The process is guaranteed to converge, because our lattices are finite and the operations are monotonic. In general, the equations need to take into account not only the clauses for each predicate (as in this example), but also the situations in which each predicate is called. It is necessary for the user to provide information about the kinds of goals (s)he will present directly to the system, unless these kinds of goals are already well represented in the clauses of the program.

4. COMPILING PROLOG FOR THE POPLOG VM

In order to give the optimizations to be discussed a concrete flavor, they will be exemplified in terms of the instructions generated for the POPLOG Virtual Machine. The POPLOG VM was originally designed to support conventional programming

languages, and its instructions map in a straightforward way into machine instructions for most conventional machines. The original VM was subsequently augmented in order to better support PROLOG. The point of showing examples in terms of the POPLOG VM is not to make a strong claim for the advantages of this particular formulation, but is to show that, whereas a straightforward compiler may emit code that makes great use of PROLOG-specific machine features, an optimizing compiler can produce code that is not dissimilar to that produced for other languages.

In POPLOG, a PROLOG procedure is implemented as a normal procedure (subroutine), with one difference. This procedure performs a normal procedure exit only to indicate failure. To indicate success, it pops the top item from a *continuation stack* (which represents a procedure and its arguments) and calls that. Basically, the continuation stack represents the sequence of goals still to be satisfied. The use of this "continuation-passing" method of implementing depth-first search in PROLOG is explored further in [9]. That paper represents a theoretical view of the technique, without going into implementation details. A PROLOG clause with multiple subgoals is normally compiled in POPLOG into code to push all the subgoals but the first onto the continuation stack and then call the first one. If there are subsequent clauses to be tried on backtracking, a normal procedure call to the first subgoal is then generated. The code following this call is then the code for these subsequent clauses (it will be reached if the subgoal returns—i.e. fails). If there are no subsequent clauses (or a cut has been encountered), then it is possible to *chain* to the first subgoal, reclaiming the current control stack frame. For example, the simple PROLOG procedure

```
a :- b, c.
a.
```

compiles into something like the following VM code (for a more detailed description of the VM, see Appendix B):

```
procedure a       ;;; start of code for procedure "a"
save              ;;; save current state
pushq   c         ;;; )
pushq   1         ;;; ) push c onto continuation stack
pushc             ;;; )
call    b         ;;; call b (returns only on failure)
restore           ;;; reset variables etc.
chainc            ;;; chain to goal on top of contn stack
endprocedure      ;;; end of code for procedure "a"
```

The `save` and `restore` instructions implement the state saving necessary for backtracking. The three instructions to push `c` as a continuation in fact involve unnecessary work (this is an aspect of the VM that can be easily improved), inasmuch as items have first to be pushed onto the *user stack*, where procedure arguments are passed). The `chainc` instruction causes the current control frame to be exited and control to be transferred to the goal at the top of the continuation stack.

In general, the code for each procedure will start by popping items from the user stack into fixed locations in the current control stack frame. These locations are given names in the code. In addition, the code for each clause will start with unification code. So the general format of a PROLOG procedure with n clauses will be something like

```
    procedure <name>
    ...                 ;;; pop arguments into ``loc1'',``loc2''...
    save
    ...                 ;;; unification code for 1st clause
    ...                 ;;; continuation pushing for 1st clause
    ...                 ;;; call of first subgoal in 1st clause
label1:                 ;;; come here if unification fails
    restore
    save
    ...                 ;;; similar for 2nd,3rd...clauses
    restore
    ...                 ;;; unification code for nth clause
    ...                 ;;; continuation pushing for nth clause
    ...                 ;;; chain to first subgoal in nth clause
labeln:                 ;;; come here if no clause succeeds
    endprocedure
```

Some examples of actual code for some small PROLOG programs are given in Appendix C.

5. OPTIMIZING DETERMINATE PROCEDURES

When determinate built-in predicates written in the implementation language (such as the input/output and arithmetic predicates) are called immediately after the `:-` of a clause (or a cut), it is possible for the POPLOG PROLOG compiler to emit code to simply "call" the relevant implementation-language procedures. These procedures always return (whether or not they "succeed") and hence do not behave in the same way as PROLOG procedures. This does not matter, because the compiler knows how to handle them specially. In fact, determinate PROLOG procedures written by the user can be treated in the same way as long as the compiler knows in advance that they are determinate. A determinate PROLOG procedure can be compiled into a "standard" procedure that always returns. We can have it return a truth value on the top of the user stack to indicate whether this return indicates success (`TRUE`) or failure (`FALSE`).

If predicates `a`, `b`, and `c` are known to be determinate, then the program

```
a :-b, !, c.
a.
```

can be compiled into the following:

```
procedure a         ;;; start of code for procedure "a"
save                ;;; save current state
call    b           ;;; call b
ifnot   lab1
chain c
```

```
lab1
    restore          ;;; reset variables etc.
    pushq   TRUE     ;;;
    endprocedure     ;;; end of code for procedure "a"
```

Notice that this formulation makes less use (in fact, no use) of the PROLOG-oriented continuation stack. The code looks much more like what one would expect from a program in a conventional language.

It is interesting to examine this kind of optimization in the context of more standard PROLOG implementations. In a standard implementation, the variables used by a procedure are kept in a *local stack*, together with information about the current position in the code of a clause and the current position in the sequence of clauses for a predicate. This stack is only popped when failure occurs or when tail recursion optimization [16] takes place. Hence it is not usually possible to use the conventional subroutine call-return mechanism. In normal POPLOG, the continuation stack subsumes much of the role of keeping the variable values needed for a use of a clause. The control stack is used only for keeping information about choice points—if there is no choice in a procedure, then the code swiftly causes a "chain" to some other procedure ("tail recursion" on the first, rather than the last, subgoal). This seems similar to Warren's proposal [14] for keeping environments (almost our continuations) and choice points (almost our control stack frames) separate. Normal POPLOG uses the conventional subroutine call mechanism for implementing backtracking, but the penalty is that it needs to manipulate a continuation stack as well.

With determinate procedures optimised in the above way, POPLOG can minimize use of the continuation stack and maximise use of the standard procedure call mechanism. There is a penalty, inasmuch as the control stack grows not only when there are choices, but in fact when most procedures are called. On the other hand, this growth is only temporary, as determinate procedures will always return (reclaim their control stack), and tail recursion optimization is still possible on the last call in a clause. With these optimizations, POPLOG actually becomes closer to standard PROLOG implementations because it keeps its variable values almost entirely in the control stack (corresponding to local stack). However, it is able to use to a significant extent the procedure call mechanism which is standard on most machines.

Many determinate predicates (such as input/output predicates) are actually expected always to succeed, and for such cases it is wasteful to return a truth value to indicate success. If the user is able to indicate that a procedure should always succeed, as well as being determinate, even better code can be produced. We have not investigated how such "success annotations" might be automatically generated.

In many ways, the advantage of knowing determinacy is only fully exploited when this information is combined with mode declarations (see Section 7).

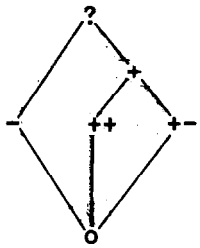
6. DERIVING MODE DECLARATIONS

We have already discussed the automatic generation of mode declarations for PROLOG programs elsewhere [7]. This section provides a brief summary of the method, described in the same terms that we used above. The notion of *modes* was introduced by Warren [15] as a way of talking about the ways in which a predicate is used in a PROLOG program. If goals involving a predicate always have a particular argument uninstantiated, then one talks of the argument of the predicate having

mode −; if that argument is always instantiated, then the mode is +; otherwise the mode is ?. We have found it necessary for automatically generating the modes for predicates in programs to enrich Warren's possible modes by expressing slightly more subtle distinctions than this (although most of the compiler optimizations presented here only need the degree of detail found in Warren's set). We have also found it necessary in addition to consider the degree of instantiation of a goal's arguments *after* the goal has succeeded. Our augmented set of possible modes is as follows:

- o Nothing known about the mode
- − Always uninstantiated
- +− Always instantiated to a structure whose arguments are all uninstantiated
- ++ Always instantiated to a ground term
- \+ Always instantiated, with all possibilities for its internal structure
- ? Sometimes instantiated, and sometimes not

The o mode is a temporary value used by the system initially. It will only arise as a finally inferred mode for certain trivial programs or if there is a predicate that is never defined or called in the program. These possible modes can be seen as forming a lattice as follows:



We wish to assign a mode to each argument place of each predicate in the program, both at time `in` (before goals are attempted) and at time `out` (after goals succeed). Using the notation `<P,N,S>` to represent the mode of argument position `N` of predicate `P` at time `S`, we can derive the following (again conservative) equations for the `append` modes in the above program:

```
<append,1,out> = ++ or
                 combine(sim(in(<append,1,in>),in(<append,3,in>)),
                         seq(in(<append,1,in>),<append,1,out>))
<append,2,out> = sim(<append,2,in>,<append,3,in>) or
                 seq(<append,2,in>,<append,2,out>)
<append,3,out> = sim(<append,2,in>,<append,3,in>) or
                 combine(sim(in(<append,1,in>),in(<append,3,in>)),
                         seq(in(<append,3,in>),<append,3,out>))
<append,1,in> =  in(<append,1,in>) or
                 <nrev,2,out>
<append,2,in> =  combine(++,in(<nrev,1,in>)) or
                 <append,2,in>
<append,3,in> =  <nrev,2,in> or
                 in(<append,3,in>)
```

where `combine`, `in`, `sim`, `seq`, and `or` are monotonic functions on modes or pairs

of modes (the last three embody the "simultaneous effects rule", the "subsequent effects rule", and the "alternatives rule" of [7]). We can apply the same iterative process as before, starting with each mode at o and iterating until a fixed point of the equations is found.

In order to infer mode declarations about a program, it is necessary to know about every possible occasion that a predicate might be invoked. Most occasions can be discovered by looking at the program source, but it is still necessary for the user to indicate what kinds of goals are to be presented directly from the terminal and what kinds of goals will arise from using the PROLOG " variable as a goal" facility.

If, as we have suggested, large parts of PROLOG programs are written with a fixed direction in mind, we might hope that these directions can be inferred automatically in the form of mode declarations. Indeed, when presented with Warren's benchmark programs, our system infers a non-? mode for 29 out of the 38 possible argument positions of predicates. For the natural-language generation program, the system infers non-? modes for 133 out of 302 argument positions, and for the version of Chat80 it infers interesting modes for 932 out of 1736 positions.

7. OPTIMIZATIONS USING MODE DECLARATIONS

Obviously if it is known in advance that a particular argument to a predicate will always be instantiated or will always be uninstantiated, then the unification code that we produce for that argument position need test for fewer possibilities. This will in general lead to faster and more compact code (Actually, in some cases in POPLOG, it can lead to slower and more bulky code because a call to a general, tightly optimized, out-of-line procedure can be preferable to a less optimized, but more specific, sequence of in-line instructions). An example where POPLOG gains from mode declarations is in the code for matching a term of the form

```
[X | Y]
```

in the head of a clause. If the mode of this argument position is ? (or unknown), the POPLOG PROLOG compiler makes use of the **pair** instruction to carry out most of the tests on the argument. This instruction pops one item off the user stack and leaves three new items there. The first two are the dereferenced form of the item and a truth value (indicating whether it is an uninstantiated variable). If it is uninstantiated, the third item returned is the dereferenced form again. Otherwide it is a truth value (indicating whether the item is a list pair or not). The instructions following the "pair" consume these truth-value results and either construct a list pair or look inside the list pair that is already there, as follows:

```
push    arg
pair
pop     temp                ;;; holds the dereferenced form
ifnot   instantiated
newvar                      ;;; ) create a new list pair
pop     X                   ;;; )
push    X                   ;;; )
newvar                      ;;; )
pop     Y                   ;;; )
push    Y                   ;;; )
cons                        ;;; )
```

```
    assign                      ;;; assign to the variable in the
                                    argument
    goto   success
instantiated:                   ;;; come here if the arg is instantiated
    ifnot  failure              ;;; see whether it is a list pair
    push   temp                 ;;; )\
    front                       ;;; ) extract the first component
    pop    X                    ;;; )/
    push   temp                 ;;; )\
    back                        ;;; ) extract the second component
    pop    Y                    ;;; )/
success:
```

If it is known that the argument is always instantiated (+), then much simpler code is possible (the instruction `ispair` just tests whether something is a list pair or not):

```
    push    arg
    ispair
    ifnot   failure
    push    arg
    front
    pop     X
    push    arg
    back
    pop     Y
success:
```

Similar optimizations can be performed for arguments which have mode −. Some care is, however, needed, as it is possible that after some unification steps have been performed an argument that was originally uninstantiated no longer is. We show in Section 8 how the places where this might happen (where sharing structures are passed into different arguments) can be determined automatically.

Another place where optimizations can be made as a result of mode declarations involves the dereferencing of PROLOG terms. Dereferencing is needed when several variables have been made to "share" [2] (this is represented by some of the bindings being represented by pointers to other variables). Finding out what binding is associated with a given variable involves following the chain of pointers until a nonpointer is reached. This is dereferencing. In general, the unification of one argument of a predicate can cause other arguments to become further instantiated. It is therefore inadequate to dereference all the arguments in advance (at the start of code for the predicate); instead it is necessary to dereference each argument each time it is subjected to unification tests in a clause. Fortunately, this extra work can be avoided in many cases. The first argument to a prediate (assuming that this is the first one to be unified) can always be dereferenced in advance. Likewise, any argument guaranteed to be + can be (although dereferencing may still be needed when accessing its subcomponents). Finally, the results of an analysis to determine which pairs of arguments might share structure (see below) can be used to detect the earliest point at which the value of an argument could have been changed by previous unification code. This can then suggest extra arguments which can be dereferenced only once, at the start of the procedure.

State saving (in POPLOG, the **save** instruction) must take place in the code for a clause when a variable is about to be bound and yet this binding may need to be undone when an alternative clause for the same predicate is later tried. The more specialized code that can be generated for unification when non-? modes are known makes it easier to postpone state saving. For instance, the POPLOG there is a VM instruction **const** to match a term against a given constant. If the term is an uninstantiated variable, **const** will bind that variable, and so it is in general necessary to save the current state at some point before this instruction is executed:

```
push    loc1
pushq   []
save
const
ifnot   label_51
```

If the argument is known to always be +, the instruction can be replaced by an **ident** instruction (which simply tests for two objects being the same pointer). With this, no state saving is needed:

```
push    loc1
pushq   []
ident
ifnot   label_51
```

This optimization is very important when there are a number of clauses for the same predicate which include different constants for a + argument. The code produced is now like that for a conventional "if-then-else" construction, rather than including state saving and restoring between successive comparisons. Warren [15] takes this further and produces special indexing code when the + argument happens to be the first. This will only be worthwhile on conventional machines if there are a reasonably large number of clauses. We have restricted our attention to predicates defined by a small number of clauses here.

Sometimes it is possible with this postponement of state saving to notice that in fact the state does not need to be saved in a given procedure. This means that the control stack frame for the procedure can be smaller.

The specialized unification code generated with the use of mode declarations makes it easier for the compiler to keep track of what values are being kept where. For instance, in the above example of matching an argument against **[X|Y]** with a + mode, the compiler can skip the code to assign the components of **loc1** to local variables associated with **X** and **Y**, simply remembering where these values can be obtained. The components of **loc1** can then be extracted when **X** and **Y** are next used. If **X** and **Y** are used exactly twice in the code, this means that their corresponding control frame locations, together with various **push** and **pop** instructions, can be dispensed with. It would be less useful for the compiler to make this optimization if the argument mode was ?. In this case, it would have to arrange that the components could be extracted later in exactly the same way, regardless of whether the argument was given as a list pair or whether a list pair had to be constructed. This would inevitably introduce an extra overhead for one of these possibilities.

When a predicate has been inferred to be determinate, a look at other programming languages suggests an alternative way of handling one of its arguments with mode -. In a conventional language, such an argument would usually be treated as a *result* to be conveyed back to the procedure caller. We can develop our existing scheme for determinate predicates to allow other values to be returned (in POPLOG, on the user stack) apart from the truth value (indicating success or failure). When a determinate procedure is called now, arguments with mode - are not passed down. Instead, the caller provides code for these arguments on the assumption that the called procedure will pass them up (if it succeeds). In effect, the procedure is being treated as a function producing values for the mode - arguments.

What advantages and disadvantages does this scheme offer? Firstly, the code of the calling procedure can often avoid initializing a PROLOG variable and then sending it as an argument, in favor of simply assigning the result returned to that variable (in fact, POPLOG cannot make best use of this). So, for example for the clause

```
nrev([X|Y],Z) :-!, nrev(Y,Z1), append(Z1,[X],Z).
```

instead of generating

```
push    Y
newvar
pop     Z1
push    Z1
call    nrev/2
ifnot   failure
push    Z1
push    X
pushq   []
cons
push    Z
call    append/3
ifnot   failure
```

for the "body", if `nrev` is known to be determinate with modes (+,-) and `append` is known to be determinate with modes (+,+,-), the compiler can avoid initializing `Z1` and can produce

```
push    Y
call    nrev/2
ifnot   failure
pop     Z1
push    Z1
push    X
pushq   []
cons
call    append/3
ifnot   failure
pop     Z
push    Z
```

For the called procedure, the advantage is that no unification code need be built for the given argument. The second version of this code finishes by pushing the value of `Z` on the stack. This is the second argument being passed back as the `result` of the `nrev` procedure. This simple pushing onto the stack replaces any unification code for the second argument.

A disadvantage of having determinate procedures return results in this way is that it may be less easy to make tail recursion optimizations. In this example, it happened that the "result" of the last subgoal was the same as that of the main procedure, and so it would have been possible simply to `chain` to `append`. In general, the result of the last subgoal and that of the main procedure will be related in more complex ways, and tail recursion may not be possible. An advantage of having less unification code in the called procedure, however, is that that procedure may be able to postpone or even avoid, state saving as a result. For instance, `append` (with or without cuts and whatever the order of clauses) can be compiled in this way into a procedure that performs no `save/restore` operations. What is happening here is that we are simply postponing instantiating a variable to a tentative binding until it is guaranteed that the subgoals introduced by the clause can be satisfied. It is therefore unnecessary to locally handle the possibility that this binding might be wrong.

It is important to appreciate that there is a class of cases where this last optimization cannot be made. Although an inferred mode of `-` for an argument position means that the argument in this position will always be uninstantiated *when the goal is invoked*, it does not follow that it still will be so after unification with clause head. The places where this will not be so are where the predicate is invoked with the same uninstantiated variable appearing in more than one argument. In fact, it is possible to automatically determine the places where this may occur (erring on the pessimistic side). They are a subclass of the places where the same structure is *shared* in two different arguments.

8. DETECTING POSSIBLE SHARED STRUCTURES

We have mentioned above several reasons why an optimizing compiler might need to know which pairs of arguments to a predicate might share structure. This notion of shared structure only makes sense at the implementation level, where one can distinguish between, for instance, two lists which happen to have identical elements and two pointers to the same list. In this context, we are interested in the sharing of uninstantiated variables and complex terms. We can apply the same techniques as before to make inferences about possible shared structures. As with mode declarations, we need to keep information about argument pairs both `in` (before a goal is attempted) and `out` (after a goal has succeeded). Let us use the notation

```
<Pred,A1,A2,Time>
```

to refer to whether or not the two arguments numbered `A1` and `A2` for predicate `Pred` might share structure at a given time `Time` (`in` or `out`). This will be a truth value. We will call a pair of argument positions "dangerous" if the corresponding value is `true`. Equations can be automatically derived from a program, relating the

"danger" of various argument positions for different predicates. For example, from the program

```
nrev([X|Y],Z) :- !, nrev(Y,Z1), append(Z1,[X],Z).
nrev([],[]).
```

one can obtain equations such as the following:

```
<nrev,1,2,out> = <nrev,1,2,in> or
                 (<nrev,1,2,out> and <append,1,3,out>) or
                 <append,2,3,out>
```

where **and** and **or** are the usual logical connectives. In this example, the three possibilities for the two arguments to be sharing structure at time **out** are:

(1) The two arguments already share structure at time **in**.

(2) The **nrev** subgoal causes **Y** and **Z1** to share, and the **append** subgoal causes **Z1** and **Z** to share.

(3) The **append** subgoal causes [X] and Z to share.

The set of equations of this kind for any reasonable-sized program will be very large, as is the number of unknowns. We can apply the same iterative technique as before, each unknown starting off with the value **FALSE**, the bottom element of the following trivial lattice:

```
TRUE
  |
FALSE
```

To make the computation more efficient, however, we can assume that by default each unknown will have the value **FALSE** and only actually apply an equation when one of the unknowns on its right-hand side is **TRUE**. Some unknowns will start off with the value **TRUE** because of an explicit repetition of a PROLOG variable in the head of a clause or in one of the goals of the clause. For instance, in the program

```
append([X|Y],Z,[X|Y1]) :- !, append(Y,Z,Y1)
append([],X,X).
```

the unknowns **<append,2,3,out>** and **<append,1,3,out>** will start off **TRUE**.

Given this way of working out which pairs of arguments of predicates may share structure when a predicate is invoked, it is possible to automatically decide where to include "occur checks" in a program, if it is desired to use unification in its pure form and yet not slow down program execution unduly. An occur check potentially needs to be included wherever a variable appears in two arguments in the head of a clause and those arguments have been inferred to be a dangerous pair. Plaisted [12] presents an alternative approach to automatically deciding where occur checks may be needed. His method seems to be working in a less abstract domain than ours, which should mean that it produces fewer spurious places but is less efficient.

Conservative conclusions about possible "dangerous" predicates can be tempered to some extent by looking at the mode declarations inferred for those predicates. If either of the arguments marked as potentially "dangerous" has mode ++, then in fact it is impossible for the unification of one to affect the value of the other. Moreover, no occur check will be needed, since a cycle can only result from unification of two structures when both contain uninstantiated variables. In fact, when presented with Warren's [15] benchmark programs, our program infers that no predicates are potentially "dangerous". In this case, it does not need mode declarations as extra help. With the natural-language generation program, the system infers that 39 out of 139 predicates are potentially dangerous. However, it infers that occur checks would only be necessary with a maximum of 9 of these.

Given information about the places in a program where structures might be shared, one can make inferences about the last place in a program where a given structure could possibly be used. It is hoped that this may provide a way of helping a PROLOG compiler to introduce destructive operations where it is safe.

9. PROGRAM DEVELOPMENT AND INCREMENTAL COMPILATION

In discussing the automatic derivation of various properties of PROLOG programs, we have neglected to mention various pragmatic factors, for instance:

(1) How quickly can these various properties be derived? In fact, our programs are quite slow (the derivation of mode declarations taking several times as long as normal compilation).

(2) What happens when a program is changed? All the compiler optimizations depend on the whole program being available in advance. In real life, a program evolves gradually and changes. Can this be accommodated?

There are a number of ways in which one might actually use the techniques described here, and one should consider how the various possibilities would cope with these problems. Firstly there is the issue of whether the inference of the program properties (modes, determinacy, etc.) should be performed by the compiler itself, or by a separate program. There seem to be advantages in keeping the two separate, in order to keep normal (nonoptimized) compilation fast and to emphasize that optimal declarations for the compiler may actually be best generated by a human being aided by automatic methods (rather than being totally automatic). Secondly, there is the issue of whether the system should be geared for use with a finished program, to provide a final higher-performance version, or whether it should be geared more fundamentally towards the idea of a developing program. There seems to be no reason why the systems for inferring program properties could not be integrated with an incremental compiler. The resulting system would keep track of the current knowledge about predicates that have been defined (including guesses about predicates that have been referenced but not yet defined) and would cause parts of the program to be automatically recompiled if changes elsewhere violated the assumptions under which the previous compilation was carried out. Indeed, many of the necessary structures to be kept by such a compiler (indicating dependencies between different parts of the program) are already built up by our (currently standalone) programs for inferring mode declarations etc. The inference

of program properties would proceed in an incremental way, as the program was developed, and so the speed element would not be too crucial. One factor about a Prolog program which is impossible to predict in advance is the kinds of goals that will be presented at the terminal and as the values of variables standing for goals. Here the compiler could again play a useful role, checking at runtime that such goals are according to the existing specifications and if necessary recompiling parts of the program so that the questions asked can indeed be correctly answered.

It seems not unreasonable to expect the compilers of the future to keep track of all sorts of facts about a program in its various stages of development and to help the user make local changes without unforseen global effects. Keeping track of the kinds of program properties discussed here might well become an important function of PROLOG compilers for conventional machines.

This research would not have been possible without John Gibson's implementation of the POPLOG VM and his design and implementation of extensions to the VM for improved PROLOG performance. I am also grateful to many people within the Cognitive Studies Programme at Sussex for useful discussions.

APPENDIX A: EXAMPLE TIMINGS

The following give times for various experimental versions of the POPLOG PROLOG system running on example tasks on a VAX-11/780 machine. Times are all in milliseconds. Note that figures that Warren gives for timings of PROLOG systems do not include garbage collection and assume user-provided mode declarations. Our timings include garbage collection (which explains some fluctuations), and only make use of *automatically generated* declarations (except in the case of "success annotations"). We have mentioned in the text various places where POPLOG cannot fully exploit all the optimizations discussed. Moreover, it is inconceivable that *all* the relevant declarations have been inferred by our programs. Thus the following represents a lower bound on the speed advantages that might be obtained in future systems.

Version	Task:	A	B	C	D	E
POPLOG V9		162	155	17	130	3358
POPLOG opt		144	147	13	96	1783
POPLOG modes		134	130	11	98	1038
POPLOG det		83	72	9	68	812
POPLOG succ		81	72	9	78	736

The versions:

POPLOG V9	POPLOG Version 9 (as currently distributed)
POPLOG opt	As POPLOG V9, with lexically scoped variables and non-compact code option
POPLOG modes	As POPLOG opt, with automatic mode declarations
POPLOG det	As POPLOG modes, with automatic determinacy declarations
POPLOG succ	As POPLOG succ, with manually added success declarations

The tasks:

A Naive reverse of 30-element list (from Warren)
B Quicksort of 50-element list (from Warren)
C Differentiation of `times10` (from Warren)
D Serialise of 25-element list (from Warren)
E "Database query" (from Warren)

APPENDIX B: THE POPLOG VM

The POPLOG VM instructions access data in five areas of the memory. The *control stack* (like the control stack in a conventional language) is used to store return addresses and local variables (mainly for choice points). The *user stack* is used for passing arguments to and results from procedures. The *continuation stack* is used for holding a representation of the remaining goals. The *trail* is used to keep references to variables that may need resetting on backtracking. The *heap* is used to store the program code itself as well as dynamically created datastructures (just lists here).

Here is a brief and simplified summary of the VM instructions that have been used here. These will be translated into subroutine calls or inline code, depending on the underlying machine. In general, each one pops a number of items off the user stack and returns any results to the user stack. The number of items taken from the user stack and the number of items pushed on the user stack are put in brackets. For instance, `assign` takes two items from the user stack and puts no items on the user stack.

`assign`		`(2,0)`	Updates a PROLOG variable to have a specific value, trailing the variable.
`back`		`(1,1)`	Extracts the `cdr` of a list pair.
`call`	`<name>`	`(?,?)`	Calls the procedure with name `<name>`.
`chain`	`<name>`	`(?,?)`	As `call`, except that the current control stack frame is reclaimed.
`chainc`		`(0,0)`	Exits the current control frame and calls a goal popped from the continuation stack (popped).
`cons`		`(2,1)`	Creates a list pair.
`const`		`(2,1)`	Sees whether an item matches a given constant, instantiating it if necessary. Returns a truth value.
`deref`		`(1,1)`	Dereferences a PROLOG term.
`endprocedure`		`(0,1)`	Returns.
`front`		`(1,1)`	Extracts the `car` of a list pair
`goto`	`<label>`	`(0,0)`	Transfers control to `<label>`.
`ident`		`(2,1)`	Tests whether two items are identical (returning a truth value).

`ifnot`	`<label>`	`(1,0)`	Transfers control to `<label>` if the top item on the user stack is non-`FALSE`.
`ispair`		`(1,1)`	Tests whether an item is a list pair (returning a truth value).
`newvar`		`(0,1)`	Creates a value for a new PROLOG uninstantiated variable. In fact, this is a pointer to a heap-allocated record at present.
`pair`		`(1,3)`	Tests whether an item is a pair or an uninstantiated PROLOG variable. First two items returned are the dereferenced form of the item and a truth value (whether it is an uninstantiated variable). If it is uninstantiated, the third item is the dereferenced form again. Otherwise it is a truth value (whether the item is a pair).
`pop`	`<name>`	`(1,0)`	Pops the top of the user stack into the variable `<name>`.
`procedure`	`<name>`	`(0,0)`	Marks the start of code for a procedure with a given name.
`push`	`<name>`	`(0,1)`	Pushes the value of variable `<name>` onto the user stack.
`pushc`		`(n,0)`	Pushes the top `n` items on the user stack (the number `n` being on the top of the user stack) onto the continuation stack.
`pushq`	`<constant>`	`(0,1)`	Pushes a constant onto the user stack.
`restore`		`(0,0)`	Restores stack pointers to the saved values associated with the current control frame, resetting variables instantiated since the corresponding "save" was executed.
`save`		`(0,0)`	Saves the values of the main stack pointers in the current control frame (ready to be reset if backtracking occurs).
`unify`		`(2,1)`	Tests whether two PROLOG terms unify, returning a truth value.

APPENDIX C: EXAMPLE POPLOG VM CODE

The following shows the POPLOG VM code generated for the first example program given above (`append` and `nrev`) without cuts. The variable names `loc1`, `loc2`, etc. refer to local variables (locations in the current control stack frame).

Without optimizations:

```
            procedure nrev/2
            pop     loc2
            pop     loc1
            push    loc1
            save
            pair
            pop     loc6
            ifnot   label_49
            newvar
            pop     loc5
            push    loc5
            newvar
            pop     loc4
            push    loc4
            cons
            assign
            goto    label_50
label_49:
            ifnot   label_46
            push    loc6
            front
            pop     loc5
            push    loc6
            back
            pop     loc4
label_50:
            push    loc4
            newvar
            pop     loc3
            push    loc3
            push    loc3
            push    loc5
            pushq   []
            cons
            push    loc2
            pushq   append/3
            pushq   4
            pushc
            call    nrev/2
            restore
label_46:
            push    loc1
            pushq   []
            const
            ifnot   label_51
            push    loc2
```

```
            pushq   []
            const
            ifnot   label_51
            chainc
label_51:
            endprocedure

            procedure append/3
            pop     loc3
            pop     loc2
            pop     loc1
            push    loc1
            save
            pair
            pop     loc7
            ifnot   label_59
            newvar
            pop     loc6
            push    loc6
            newvar
            pop     loc5
            push    loc5
            cons
            assign
            goto    label_60
Label_59:
            ifnot   label_56
            push    loc7
            front
            pop     loc6
            push    loc7
            back
            pop     loc5
label_60:
            push    loc3
            pair
            pop     loc8
            ifnot   label_61
            push    loc6
            newvar
            pop     loc4
            push    loc4
            cons
            assign
            goto    label_62
label_61:
            ifnot   label_55
            push    loc6
```

```
            push    loc8
            front
            unify
            ifnot   label_55
            push    loc8
            back
            pop     loc4
label_62:
            push    loc5
            push    loc2
            push    loc4
            call    append/3
label_55:
            restore
label_56:
            push    loc1
            pushq   []
            const
            ifnot   label_63
            push    loc2
            push    loc3
            unify
            ifnot   label_63
            chainc
label_63:
            endprocedure
```

With optimizations:

```
            procedure nrev/2
label_27:
            deref
            pop     loc1
            push    loc1
            ispair
            ifnot   label_24
            push    loc1
            back
            call    nrev/2
            pop     loc2
            push    loc2
            push    loc1
            front
            pushq   []
            cons
            call    append/3
            pop     loc3
            push    loc3
            goto    label_29
```

```
label_24:
          push      loc1
          pushq     []
          ident
          ifnot     label_29
          pushq     []
          goto      label_29
label_29:

          endprocedure
          procedure append/3
          deref
          pop       loc2
          deref
          pop       loc1
          push      loc1
          ispair
          ifnot     label_33
          push      loc1
          back
          push      loc2
          call      append/3
          pop       loc3
          push      loc1
          front
          push      loc3
          cons
          goto      label_38
label_33:
          push      loc1
          pushq     []
          ident
          ifnot     label_38
          push      loc2
          goto      label_38
label_38:
          endprocedure
```

REFERENCES

1. Bundy, A., Byrd, L., Luger, G., Mellish, C. and Palmer, M., Solving Mechanics Problems Using Meta-Level Inference, *Proc. IJCAI-79*, Tokyo, 1979.
2. Clocksin, W. and Mellish, C., *Programming in Prolog*, Springer, 1981.
3. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in *Principles of Programming Languages*, 1977.
4. Mackworth, A., Consistency in Networks of Relations, *Artificial Intelligence* 8:99–118 (1977).
5. McDermott, D., The Prolog Phenomenon, *Sigart Newsletter* 72, July 1980.

6. Mellish, C., Coping with Uncertainty: Noun Phrase Interpretation and Early Semantic Analysis, Ph.D. Thesis, Univ. of Edinburgh, 1981.
7. Mellish, C., The Automatic Generation of Mode Declarations for Prolog Programs, Research Report 163, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1981.
8. Mellish, C., Natural Language Generation from Plans, Cognitive Studies Research Report, Univ. of Sussex, 1984.
9. Mellish, C. and Hardy, S., Integrating Prolog into the POPLOG Environment, *Proc. IJCAI-83*, Karlsruhe, Germany, 1983.
10. Milne, R., Resolving Lexical Ambiguity in a Determinisitic Parser, Ph.D. Thesis, Univ. of Edinburgh, 1983.
11. Mycroft, A., The Theory and Practice of Transforming Call-by-need into Call-by-value, in *Springer Lecture Notes in Computer Science*, No. 83, Springer, 1980.
12. Plaisted, D., The Occur-Check Problem in Prolog, *Proc. International Symposium on Logic Programming*, Atlantic City, 1984.
13. Sintzoff, M., Calculating Properties of Programs by Valuations on Specific Models, *Sigplan Notices* 7, No. 1 (1972).
14. Tick, E. and Warren, D., Towards a Pipelined Prolog Processor, *Proc. International Symposium on Logic Programming*, Atlantic City, 1984.
15. Warren, D., Implementing Prolog—Compiling Predicate Logic Programs, Research Report 39, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.
16. Warren, D., An Improved Prolog Implementation which Optimises Tail Recursion, *Proc. Logic Programming Workshop*, Debrecen, Hungary, 1980.
17. Warren, D. H. D. and Pereira, F. C. N., *An Efficient Easily Adaptable System for Interpreting Natural Language Queries*, American Journal of Computational Linguistics 8:110–122 (1982).