Towards Banishing the Cut from Prolog†

Saumya K. Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

David S. Warren
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract: There has been a great deal of interest in logic programming languages in recent years. This is due, in great part, to the advantages offered by these languages: declarative readings of programs and separation of program logic from control. However, logic programs can often be dismayingly inefficient. The usual solution to this problem has been to return some control to the user in the form of impure language features like *cut*. Unfortunately, this often results in the loss of precisely those features that made logic programming attractive in the first place.

We argue that it is not necessary to resort to such impure features for efficiency. This point is illustrated by considering how most of the common uses of *cut* can be eliminated from Prolog source programs, relying on static analysis to generate them at compile time. Three common situations where the cut is used are considered. Static analysis techniques are given to detect such situations, and applicable program transformations are described. We also suggest two language constructs, *firstof* and *oneof*, for situations involving ‘‘don't-care’’ nondeterminism. These constructs have better declarative readings than the cut, and extend better to parallel evaluation strategies. Together, these proposals result in a system where users need rely much less on cuts for efficiency, thereby promoting a purer programming style without sacrificing efficiency.

---

† This research was performed while the first author was at the Department of Computer Science, SUNY at Stony Brook, NY 11794; it was supported, in part, by the National Science Foundation under grant no. DCR-8407688.

1. Introduction

There has been a significant interest, in recent years, in logic programming languages like Prolog. This is due in great part to the advantages offered by these languages: clear, declarative readings of programs; separation of program logic from control; and the consequent ease of understanding and maintaining programs. Unfortunately, this otherwise bright picture is marred by the pragmatic problem of efficiency. Logic programs can often be dismayingly inefficient. The very mechanism that lets users write programs without having to worry about control aspects of problems can, in many cases, lead to programs that are unduly expensive to run.

The usual solution to this problem has been to return some control to the programmer, in the form of impure language constructs such as *cut*, failure-driven loops with side effects, etc. While these do alleviate the efficiency problem somewhat, they often do so at the cost of precisely those features that made logic programming attractive in the first place. As a result, they seriously undermine the credibility of these languages as declarative languages based on logic.

The purpose of this paper is to argue that static analysis can, in very many cases, be used to produce code of good quality, without having to resort to these impure language features at the source level. As an example, we could eliminate cuts from Prolog programs, relying on compile-time analysis to generate them instead. The status of *cut* would thus be reduced to a low level implementation feature of sequential Prolog. The advantage of such an approach is that the desirable features of logic programming languages, such as declarative semantics, are retained, without surrendering efficiency.

We assume some acquaintance with logic programming languages, in particular Prolog. The remainder of this paper is organized as follows: Section 2 discusses some basic notions. Section 3 considers the *cut* construct, and identifies three common uses of cut. Sections 4 and 5 describe static analysis and optimization techniques that can be used to render most of these cuts unnecessary in the source program: Section 4 considers the problem of inferring mutual exclusion of clauses and functionality of predicates and literals, and constructs to express ‘‘don’t-care’’ nondeterminism; Section 5 considers source-to-source transformations for converting tail recursive predicates to failure-driven loops with cuts. Section 6 gives some experimental results from an implementation, and Section 7 concludes with a summary.

2. Preliminaries

This section briefly outlines the basics of Prolog, and may be skipped by the reader familiar with the language. First, the syntax and semantics of the language are discussed. This is followed by a brief outline of the notions of *modes* and *functional dependencies*.

2.1. Prolog: Syntax

There are three kinds of symbols in Prolog: variables, function symbols and predicate symbols. The entities comprising a Prolog program can be divided into two broad categories: data objects called *terms*, and program objects called *clauses*. A term is defined to be either a variable, a constant, or a structure $f(t_1, ..., t_n)$, where f is a function symbol of arity n, i.e. taking n arguments, and the t_i, $1 \le i \le n$, are terms. A structured term $f(t_1, ..., t_n)$ is conceptually analogous to a Pascal record whose type is f and whose fields are $t_1, ..., t_n$. Examples of terms are

X 23 $np(det(a), noun(dog))$ $1{\circ}2{\circ}3{\circ}[]$

where X is a variable; 23 is a constant; $np(det(a), noun(dog))$ is a structured term consisting of the function symbol np of arity 2, and the subterms $det(a)$ and $noun(dog)$; the last term is a list of three elements, the list constructor being the binary (infix) function symbol $\circ$, with the empty list denoted by []. For notational convenience in the case of lists, we usually omit explicit mention of the list constructor $\circ$, e.g. write the list above as [1, 2, 3]. A list $Hd{\circ}Tl$ will also be written $[Hd\,|\,Tl]$.

A clause consists of two parts, the *head* and the *body*. The head of a clause is an atom $p(t_1, ..., t_n)$, where p is an n-ary predicate symbol and the t_i, $1 \le i \le n$, are terms. In this case, p is said to be the predicate symbol of the clause. The body of a clause is a sequence of literals separated by commas, where each literal is either an atomic goal like the head, or the negation of an atom *not*(*Atom*). A clause with head *Head* and body *Body* will be written

Head :– *Body*.

In addition, a literal may be the *cut* construct '!', which is discussed in more detail later. If the body of a clause is empty, the clause written simply as

Head.

Following the convention of Edinburgh Prolog (see [3]), variable names will begin with upper case letters, while function and predicate symbol names will begin with lower case letters; in addition, ''void'' or ''anonymous'' variables (variables that appear only once in a clause), which function as placeholders, will be written as underscores '_'.

A Prolog program consists of a set of predicate definitions. A predicate definition consists of a sequence of clauses having the same predicate symbol. Conceptually, a clause corresponds to a procedure definition, where the head gives the formal parameters and the literals in the body correspond to the sequence of procedure calls defining the procedure body. A predicate definition corresponds to a procedure definition, each clause for the predicate corresponding to an alternative body for the procedure. Thus, a predicate definition in Prolog can be thought of as a simple generalization of procedure definitions in traditional languages, in that multiple alternative procedure bodies, not necessarily mutually exclusive, are permitted. An example of a predicate definition is given by the following clauses for quicksort:

```
qsort([], []).
qsort([M|L], R) :–
        split(M, L, U1, U2),
        qsort(U1, V1),
        qsort(U2, V2),
        append(U1, [M|V1], R).
```

2.2. Prolog: Semantics

The meaning of a Prolog program is usually given in terms of the model theory of first order logic. In this view, each clause in a program is interpreted as a logical formula; the meaning of a predicate is then the conjunction of the meanings of the clauses defining it. The meaning of a clause is given as follows: consider a clause

$$p(\bar{X}) :- q_1(\bar{X}, \bar{Y}), ..., q_n(\bar{X}, \bar{Y}).$$

where $\bar{X}$ consists of the variables appearing in the head, and $\bar{Y}$ consists of the variables appearing only in the body. Then, the logical meaning of the clause is the formula

$$(\$forall\ \bar{X})(\$exists\ \bar{Y})[q_1(\bar{X}, \bar{Y})\ \$and\ ...\ \$and\ q_n(\bar{X}, \bar{Y}) \Rightarrow p(\bar{X})].$$

Thus, a clause ''p :− q_1, ..., q_n'' is read as ''p *if* q_1 *and* ... *and* q_n''. This is the *declarative* reading of a Prolog program, and allows the meaning of a program to be understood without necessarily having to work out complicated control flow details.

Programs can also be understood procedurally. In this view, each predicate is a procedure defined by its clauses. Each clause provides an alternate definition of the procedure body. The terms in the head of the clause correspond to the formal parameters, and each literal in the body of the clause corresponds to a procedure call. Thus, the definition of the *qsort* predicate above can be understood as two alternative procedure bodies: one for the case where the list to be sorted is empty, and the other, for the case involving a nonempty list, consisting of a call to the procedure *split*, followed by two recursive calls to the procedure *qsort*, and finally a call to the procedure *append*.

Parameter passing in such procedure calls is via a generalized pattern matching procedure called *unification*. Briefly, two terms t_1 and t_2 are unifiable if there is some substitution of terms (called the *unifier*) for the variables occurring in t_1 and t_2 which make t_1 and t_2 identical. For example, the terms $f(X, g(X, Y))$ and $f(a, Z)$ are unifiable with the unifier $\{X \rightarrow a, Z \rightarrow g(a, Y)\}$. Usually the most general unifier, i.e. one that does not make any unnecessary substitutions, is used: e.g. the two terms above could also have been unified via the unifier $\{X \rightarrow a, Y \rightarrow b, Z \rightarrow g(a, b)\}$, but this unnecessarily substitutes for the variable Y, and hence is not considered. The result of unifying two terms is the term produced by applying their most general unifier to them, e.g. the result of unifying the two terms above is $f(a, g(a, Y))$. If the terms under consideration are not unifiable, then unification is said to *fail*. A fundamental theorem of logic programming states that whenever two terms are unifiable, they have a most general unifier that is unique upto variable renaming; and furthermore, there is an algorithm that finds the most general unifier whenever it exists, and reports failure otherwise [18]. Application of unifiers is global in the sense that when a variable undergoes substitution as a result of a unification, all literals that share that variable undergo the appropriate substitution.

Operationally, the execution of a Prolog program follows the textual order of clauses and literals. Execution begins with a query from the user, which is a sequence of literals processed from left to right. The processing of a literal proceeds as follows: the clauses for its predicate are tried, in order, until one unifies with it. If there are any remaining clauses for that predicate whose heads might unify with that literal, a backtrack point is created to remember this fact. After unification with the head, the literals in the body are processed from left to right. If unification fails at any point, execution backtracks to the most recent backtrack point, and the next clause is tried, and so on. Execution terminates when all the literals have been processed completely.

Strictly speaking, a literal is a static component of a clause. In an execution of the program, the corresponding dynamic entity is a call, which is a substitution instance of an alphabetic variant of the literal. A call to an n-ary predicate p can therefore be considered to be a pair $\$\mathit{langle}\, p/n, \bar{T}\$\mathit{rangle}$ where $\bar{T}$ is an n-tuple of terms which are arguments to the call. When the predicate being referred to in a call is clear from the context, we will omit the predicate name and refer to the call by its tuple of arguments. We refer to calls corresponding to a literal as *arising from* that literal. If the predicate symbol of a literal L is p, then L will be said to *refer to* the predicate p.

If we consider the input-output behavior of a Prolog program, a call to a predicate, with a particular tuple of input arguments, can return multiple output values via backtracking. The same is true of literals (which amount to a call to the corresponding predicate) and clauses (which consist of a sequence of literals). The denotation of a predicate, clause or literal can therefore be thought of as a relation over pairs of tuples of terms. Thus, if D is the set of terms (possibly containing variables) in a program, then an n-ary predicate (clause, literal, call) denotes a subset of $D^n \times D^n$. The first element of the pair represents the ''input'', or calling, values of its arguments, and the second, the ''output'', or returned values. We will refer to such relations as *input-output relations*. If a pair $\$\mathit{langle}\, t_1, t_2\$\mathit{rangle}$ is in the input-output relation of a predicate p, and θ is the most general unifier of t_1 and t_2, then we will say that a call t_1 to p can *succeed* with the substitution θ. Notice that for any pair $\$\mathit{langle}\, t_1, t_2\$\mathit{rangle}$ in an input-output relation, t_2 must be a substitution instance of t_1.

Recall that an ''anonymous'', or ''void'', variable is one that appears only once in a clause, and hence functions as a placeholder. Values returned for void variables occurring as top-level arguments do not affect computations in any way. We will therefore sometimes wish to ignore values returned for void variables. Given an n-tuple T and a set of argument positions $A = \{m_1, \ldots, m_k\}$, $1 \le m_1 < \ldots < m_k \le n$, let the *projection* of T on A denote the k-tuple obtained by considering only the values of the argument positions $m_1 \ldots m_k$ of T. Then, the projection of an input-output relation on a set of argument positions can be defined as follows:

Definition: Given an input-output relation R and a set of argument positions A, the *projection* of R on A, written $\pi_A(R)$, is the set of pairs $\$\mathit{langle}\, S_I, S_O\$\mathit{rangle}$ such that for some pair $\$\mathit{langle}\, T_I, T_O\$\mathit{rangle}$ in R, S_I is the projection of T_I on A, and S_O is the projection of T_O on A.

Moreover, given an n-ary predicate in a program, not all possible n-tuples of terms are encountered in calls to it. For program analysis purposes, therefore, it suffices to restrict our attention to a horizontal slice of the input-output relation of a predicate. To this end, we define the notion of input restriction:

Definition: Let R be the input-output relation for an n-ary predicate (clause, literal) in a program, and let T be a set of n-tuples of terms. The *restriction* of R to T, written $\sigma_T(R)$, is the set of pairs *$langle* S_I, S_O*$rangle* in R such that S_I is in T.

The application of projections and restrictions to the static analysis of Prolog programs is discussed in Section 4.

2.3. Negative Goals

It is possible to have negative literals in the body of a clause. The semantics of negated goals is given in terms of unprovability by finite failure, i.e. a goal *not*(P) succeeds if every attempt to solve the goal P fails in a finite number of steps; it coincides with logical negation (with respect to the ''completed'' predicate) under certain conditions − the set of program clauses must have a minimal model and all negated goals to be proved must be ground [2]. It turns out that if a variable occurs within a negated goal, then *as long as this variable does not appear anywhere outside the negation*, it behaves logically as a universally quantified variable. This is evident if we consider the clauses

```
p(X) :− not( q1 ), r(X).
q1 :− q(Z).
```

where the negation has the expected semantics. Unfolding the negated literal in the clause for p after appropriately renaming variables gives

```
p(X) :− not( q(Y) ), r(X).
```

where the variable renaming guarantees that variables in the negated goal do not appear elsewhere in the clause outside the negation. The negated goal here can be thought of as representing the statement ''for no instance of Y is $q(Y)$ provable''.

2.4. Modes and Functional Dependencies

In general, Prolog programs are undirected, i.e. can be run either ''forwards'' or ''backwards'', and do not distinguish between ''input'' and ''output'' parameters for a predicate. However, in most programs, individual predicates tend to be used with some arguments as input arguments and others as output arguments. Knowledge of such directionality, expressed using *modes* [21], enables various compile-time optimizations to be performed [7, 14, 21]. Mode information can either be supplied by the user, in the form of mode declarations, or be inferred from a global analysis of the program [7, 13]. Mode information also plays an important role in the inference of functionality.

It is convenient to think of the mode for an n-ary predicate as representing a set of n-tuples of terms. The modes we consider are quite simple: **c** represents the set of ground terms, **f** the set of variables and **d** the set of all terms. Thus, if a predicate *p/3* has mode *$langle* **c**,**f**,**d***$rangle* in a program, then it will always be called with its first argument ground and its second argument uninstantiated in that program; however, nothing definite can be said about the instantiation of its third argument. In general, a mode for an n-ary predicate will be an n-tuple over {**c**,**d**,**f**}. A call to a predicate with arguments $\bar{X}$ is *consistent* with the mode of that predicate if $\bar{X}$ is in the set of tuples of terms represented by that mode.

If we assume an ‘‘input’’ mode for a predicate or clause for a predicate, it is possible to propagate it from left to right to literals in the body and obtain modes for these literals [7]. The modes so inferred will be said to be *induced* by the input mode. For example, consider the program

$p(X, Y)$:– $q(X, Y)$, $r(X, Y)$.
$q(a, b)$.
$q(b, c)$.
$r(b, d)$.

Suppose we know that p is always called with its first argument bound to a ground term, and its second argument free, i.e. has the mode *$langle* **c**, **f***$rangle*. This information can be propagated across the body of the clause, e.g. we can infer that q has the induced mode *$langle* **c**, **f***$rangle*; an examination of the clauses for q shows that q succeeds binding its arguments to ground terms, so that the induced mode for r is *$langle* **c**, **c***$rangle*.

The notion of *functional dependencies* is well known in relational database theory. Given a predicate $p(\bar{X})$, where $\bar{X}$ is a sequence of distinct variables, if there exist subsets of its arguments $\bar{U}, \bar{V} \subseteq \bar{X}$ such that a ground instantiation of the arguments $\bar{U}$ uniquely determines the instantiation of the arguments $\bar{V}$, then $\bar{U}$ is said to *functionally determine* $\bar{V}$ in p, and $\bar{V}$ is said to *depend functionally* on $\bar{U}$. More formally, if $\bar{U}$ functionally determines $\bar{V}$ in $p(\bar{U}, \bar{V})$, and $\bar{u}$ is a ground instantiation of $\bar{U}$, then for all $\bar{v}_1$ and $\bar{v}_2$, whenever $p(\bar{u}, \bar{v}_1)$ and $p(\bar{u}, \bar{v}_2)$ are true, it must be the case that $\bar{v}_1 = \bar{v}_2$. We will use the notation ‘‘$L : S_1 \rightarrow S_2$’’, where L is a literal and S_1 and S_2 are sets of variables occurring in L, to indicate that there is a functional dependency between S_1 and S_2 in L, i.e. that if L is executed with ground instantiations for the variables in S_1, then the instantiations of the variables in S_2 will be uniquely determined if the call returns successfully. Alternatively, we will say that S_1 functionally determines S_2 in L. The following axioms, known as Armstrong’s axioms, are sound and complete for functional dependencies:

Reflexivity: If $S_2 \subseteq S_1$, then $L : S_1 \rightarrow S_2$.

Transitivity: If $L : S_1 \rightarrow S_2$ and $L : S_2 \rightarrow S_3$ then $L : S_1 \rightarrow S_3$.

Augmentativity: If $L : S_1 \rightarrow S_2$, and $S = S_0 \cup S_1$ for some S_0, then $L : S \rightarrow S_2$.

This extends in a natural way to conjunctions of literals, and to clauses: if L is a member of a conjunct C and $L : S_1 \rightarrow S_2$, then $C : S_1 \rightarrow S_2$; if Cl is a clause ‘‘H :– B’’ and $B : S_1 \rightarrow S_2$, then $Cl : S_1 \rightarrow S_2$.

Let S be a set of variables in a clause C, and F a set of functional dependencies that hold in the clause. The set of all variables in that clause that can be inferred to be functionally determined by S under F, using the axioms above, is called the *closure* of S under F. If S_2 is the closure of S_1 under a set of functional dependencies F, we will write $C : S_1 \xrightarrow[F]{*} S_2$. Given a set of functional dependencies F and a set of variables S, the closure of S under F can be determined in time linear in the size of F (see e.g. [12]). We will anticipate what follows by mentioning that knowledge of functional dependencies and modes can be used to determine when a predicate can return at most one answer to any call to it. Before considering this analysis and its applications, however, we discuss the *cut* construct of Prolog.

3. The ''Cut'' Construct

This section considers the behavior of Prolog's *cut* construct, gives a brief description of how it is implemented on our system, considers the ways in which cut is commonly used, and describes some of the problems with it.

3.1. The Behavior of Cut

As mentioned in the previous section, execution backtracks to the most recent backtrack point when unification fails. One problem that can arise with this simple control strategy is that execution may backtrack exhaustively through clauses that cannot contribute to a solution (in extreme cases, this can lead to logically correct programs going into an infinite loop). The *cut* construct returns some control over this backtracking behavior to the user.

Operationally, the effect of a cut is to discard certain backtrack points, so that execution can never backtrack into them. While the implementation of cut in a sequential interpreter is not difficult, it complicates the semantics of the language substantially (see, for example, [6, 9]). It also becomes difficult to read the program declaratively, so that one of the principal advantages claimed for logic programming languages is lost. To make things worse, the behavior of cut is not universally agreed upon in all contexts [15]. Indeed, the cut has the dubious distinction of being one of the few features of Prolog that, while reviled by purists [11, 19], is used extensively by programmers for efficiency reasons.

In practice, cuts are usually encountered in one of two static contexts: as part of the top-level conjunction in a clause, or within a disjunction in a clause, i.e. either in a context

```
p :- . . .
p :- . . ., !, . . .
p :- . . .
```

or in a context

```
p :- . . .
p :- . . ., ( ( . . . !, . . . ) ;
              ( . . . )
            ), . . .
p :- . . .
```

Most current implementations of Prolog behave similarly in their treatment of cut in these contexts. The expected behavior here is that the backtrack points discarded by a cut will consist of: all those set up by literals to the left of the cut all the way to the beginning of the clause; and the backtrack point for the parent predicate whose definition includes the clause containing the cut, i.e. all remaining alternative clauses for this predicate. Cuts exhibiting this behavior are sometimes referred to as *hard* cuts; this is to distinguish them from cuts which discard the backtrack points set up by literals to the left of the cut in the clause but not the alternative clauses for the predicate, and are referred to as *soft* cuts. We will restrict our attention to cuts that occur statically in the above contexts, and assume them to be hard unless explicitly mentioned.

3.2. Implementation of Cut

Prolog systems are implemented in a stack-oriented manner. A call to a predicate corresponds to a procedure call, and results in an activation record for the called predicate being pushed on the runtime stack.[1] If there is more than one clause whose head might match the arguments in the call, a backtrack point is created. This consists of a special frame, called a *choice point* or *backtrack point*, that is pushed on the stack. A choice point contains the address of the next clause to try on backtracking; and values of various registers and environment variables, e.g. the top of the stack, the top of the heap, the previous choice point, etc., to enable the current state to be restored upon backtracking. When execution backtracks, it picks up at the most recent backtrack point, i.e. the one nearest the top of the stack. In this case the portion of the stack above the backtrack point is popped off, registers and environment variables are reset appropriately, and execution resumes in the forward direction with the next clause, with the ‘‘next clause to try’’ pointer in the choice point appropriately updated.

When a cut is encountered, a number of backtrack points at the top of the stack have to be discarded. In any implementation of cut, it is necessary to know how far to cut back to in the chain of choice points, i.e. how many choice points to discard. One way of doing this is to note the current choice point at an appropriate point in execution, and cut back (i.e. reset the backtrack point) to this point when a cut is encountered. In our system, this is done via two primitives, *savecp*/1 and *cutto*/1. These are internal primitives that are introduced by the compiler, and are not available to the user. The call *savecp*(X) saves the current choice point in the variable X, while the call *cutto*(X) sets the current choice point to that saved in the variable X, effectively discarding any backtrack point more recent than that in X. Thus, a predicate with a cut in it,

$p(\bar{X}) \text{ :- } q(\bar{Y}), !, r(\bar{Z}).$
$p(\bar{X}) \text{ :- } s(\bar{U}).$

is transformed by the compiler to

$p(\bar{X}) \text{ :- } savecp(W), p1(\bar{X}, W).$

$p1(\bar{X}, W) \text{ :- } q(\bar{Y}), cutto(W), r(\bar{Z}).$
$p1(\bar{X}, \_) \text{ :- } s(\bar{U}).$

where W is a new variable not occurring in the original definition of p, and *p1* is a new predicate not appearing in the original program. In this case, in any call to p, the choice point before entering p is

[1] Details vary from implementation to implementation. Usually more than one stack is involved, but these can be thought of as optimizations of a single runtime stack.

noted in the variable *W* and passed into *p1*. At the point where the cut occurs, after the literal $q(\bar{Y})$, execution of *cutto* causes all choice points created since *p* was entered to be discarded. As the reader may verify, this amounts to the discarding of all the choice points for literals to the left of the cut, and the remaining clauses for *p*.

In general, *savecp* and *cutto* can be used to bracket literals whose choice points are to be cut. Thus, depending on the relative locations of *savecp* and *cutto*, both hard and soft cuts can be implemented in a uniform fashion.

3.3. Common Uses of Cut

It is possible, in principle, for cuts to be used in a wide variety of contexts. In practice, however, most uses of cut tend to fall into one of three categories:

(1) Cuts to commit to a clause: here the cut tends to be towards the beginning of the clause, possibly after some tests.

(2) Cuts to force computations to be functions (i.e. deterministic): here the cut tends to appear towards the end of the clause. Similar usage is encountered in contexts involving ‘‘don’t-care’’ nondeterminism.

(3) Cuts to break out of failure-driven loops: while the natural specification for a loop in Prolog is a tail-recursive predicate, tail recursion optimization does not reclaim space from the heap. Implementations lacking garbage collectors, therefore, may have to resort to using failure to reclaim space on the heap, e.g. by coding loops using *repeat/fail* constructs. Cuts are then used to break out of these loops.

In each of these cases, it is possible to have the compiler generate most of the cuts via static analysis, and perform the appropriate transformations (e.g. from tail recursion to failure-driven loop), leading to source code that is much purer and easier to understand. Section 4 considers static analysis and transformation techniques for cases (1) and (2), while Section 5 discusses automatic program transformation techniques applicable to case (3).

3.4. Problems with Cut

There are several reasons one might wish to avoid using cuts. The foremost is that cuts often make programs considerably harder to understand declaratively. As an example, consider the following predicate to find the larger of two numbers:

```
max(X,Y,X) :– X > Y, !.
max(X,Y,Y).
```

Because of the cut, the two clauses cannot be understood declaratively as alternative definitions for *max*/3: the second clause is to be tried if, and only if, the first fails. The clauses do not make sense unless this operational effect of the cut is taken into account explicitly.

Another problem with the use of cuts is that because of their non-local nature, they do not interact gracefully with program transformation systems [16]. For example, unfolding a predicate without taking the presence of cuts into account can result in the loss of some solutions. This, in turn, can lead to the failure or nontermination of queries that would otherwise have terminated successfully.

A third problem with cut is its inherently sequential nature. In order to give the expected behavior, it is necessary to assume an ordering on the clauses and literals of predicates containing cuts. This means that if a predicate contains cuts, its evaluation will effectively be forced to be sequential in order to respect the cut, even if the underlying implementation allows parallel evaluation. Under such circumstances, cuts may actually cause programs to run slower. This can be a significant problem if large Prolog applications are to be ported from sequential systems to ones that allow parallel evaluation.

4. Mutual Exclusion, Functionality and ‘‘Don’t-care’’ Nondeterminism

Two of the most common uses of cut, referred to in the previous section, are (*i*) to commit to a clause, and (*ii*) to compute a function, i.e. discard all solutions returned by a predicate except the first. In the former case, the cut tends to appear towards the beginning of the clause, while in the latter it tends to appear towards the end.

If we use a cut to commit to one of the clauses of a predicate, the effect is to create two partitions of the clauses for that predicate such that for any call to that predicate, a clause from at most one of the partitions can succeed. In other words, the clauses in the two partitions are *mutually exclusive*. If the purpose of the cut is to discard all solutions of a predicate but the first, then the predicate is being constrained to compute a function, i.e. the predicate is *functional*. A special case of this involves ‘‘don’t-care’’ nondeterminism. Here, any one solution returned by a predicate is computationally adequate, so that all other solutions for it can be discarded.

4.1. Definitions

Two clauses for a predicate are said to be mutually exclusive relative to a call to that predicate at runtime if at most one of them can succeed for that call. Two clauses are mutually exclusive relative to a mode M if they are mutually exclusive relative to every call consistent with M. If two clauses are mutually exclusive relative to all calls to a predicate that can arise in a program at runtime, they will be referred to simply as *mutually exclusive*.

A predicate is said to be functional if any call to it can return at most one distinct answer. It generalizes the notion of determinacy, which refers to the ability to succeed at most once [14, 20]. The difference is illustrated by the following example:

Example 1: The predicate defined as

$p(a).$
$p(X) :- p(X).$

is functional, since its solution set is the singleton $\{p(a)\}$. However, it is not determinate, since it can succeed infinitely many times when backtracked into. ●

Functionality can be considered at the level of literals, clauses and predicates. We define these notions as follows:

Definition: Let R be the input-output relation of a literal L, in a program, and A the set of its non-void argument positions. Then, L is *functional relative to a call C* in the program iff $\pi_A(\sigma_{\{C\}}(R))$ is a function.

A literal is *functional relative to a mode M* iff it is functional relative to every call consistent with M.

In other words, a literal is functional if and only if any call that can arise from it is functional on its non-void arguments, i.e. if the bindings for the void argument positions are projected away, then, in the resulting input-output relation, each input value is mapped to a unique output value.

Definition: Let C be a set of calls to a predicate (clause) with input-output relation R. The predicate (clause) is *functional relative to C* iff $\sigma_C(R)$ is a function.

A predicate (clause) is *functional relative to a mode* iff it is functional relative to every call consistent with that mode.

If a literal (clause, predicate) is not functional, it will be said to be *relational*. The problem of static inference of functionality is considered later in this section.

4.2. Mutual Exclusion of Clauses

4.2.1. Inference of Mutual Exclusion

The most obvious condition under which two clauses are mutually exclusive is when there is a cut in the body of the textually antecedent clause. However, it is possible to detect mutual exclusion statically even without cuts, as the following propositions show:

Proposition 1: Two clauses are mutually exclusive relative to a mode M if the arguments corresponding to some subset $\overline{U}$ of the argument positions in their heads are not unifiable, and each argument in $\overline{U}$ is ground in any call to the corresponding predicate consistent with M.

Proof: It follows from the properties of unification that for any call to the predicate consistent with M, unification will fail for one clause or the other on the argument positions $\overline{U}$, i.e. no call to the predicate will unify with the heads of both clauses. Thus, no call consistent with M will succeed through both clauses, so the clauses are mutually exclusive relative to any call consistent with M. *$box*

Example 2: The clauses

$p(a, f(X), Y) :- q1(X, Y).$
$p(b, f(g(Y)), Z) :- q2(Y, Z).$

are mutually exclusive given the mode *$langle* **c**,**d**,**d***$rangle* for *p*, since the first arguments in the heads of the clauses are not unifiable; however, they may not be mutually exclusive given the modes *$langle* **d**,**c**,**d***$rangle* or *$langle* **d**,**d**,**c***$rangle* for *p*. ●

It is possible to weaken this condition somewhat, so that the relevant terms in the calls are not required to be ground, as long as they are ‘‘sufficiently instantiated’’ to discriminate between the clauses.

Proposition 2: Two clauses for a predicate *p* of the form

$p(\bar{X}) :- p_1, ..., p_k, r(\bar{Y}_0), s_1, ..., s_m.$
$p(\bar{X}) :- q_1, ..., q_l, not(\ r(\bar{Y}_1)\ ), t_1, ..., t_n.$

are mutually exclusive relative to a mode *M* if $p_1, ..., p_k$ and $q_1, ..., q_l$ are functional relative to their modes induced by *M*, and for any call *c* to *p* that is consistent with *M*, the call arising from the literal $r(\bar{Y}_0)$ in the first clause is subsumed by the call arising from the literal $not(r(\bar{Y}_1))$ in the second.

Proof: If the literals p_i and q_j are functional relative to their modes induced by *M*, then for any call consistent with *M*, there will be only one call arising from each of the literals $r(\bar{Y}_0)$ and $not(r(\bar{Y}_1))$. Assume that the subsumption condition of the proposition is satisfied for these. Then, at runtime, if the goal $r(\bar{Y}_0)$ is called with substitution σ and succeeds, then the goal $r(\sigma(\bar{Y}_1))$ will also succeed. Therefore, the goal $not(r(\sigma(\bar{Y}_1)))$ will fail. Conversely, the goal $not(r(\bar{Y}_1))$ can succeed only if no instance of $r(\bar{Y}_1)$ succeeds, which means that $r(\bar{Y}_0)$ must fail. Thus, the two clauses can never both succeed for any call consistent with *M*, i.e. they are mutually exclusive relative to *M*. *$always*

For this proposition to hold, it is necessary that $r(\bar{Y}_1)$ subsume $r(\bar{Y}_0)$ in the program. This does not guarantee subsumption at runtime, of course, but sufficient conditions for runtime subsumption can be given. One such sufficient condition is that $p_1, ..., p_k$ and $q_1, ..., q_l$ are identical, and do not share any variables with $r(\bar{Y}_1)$. Another is that none of the q_j instantiate any variables, e.g. that they are negated literals.

This proposition assumes a finite failure semantics for negation. Analogous propositions can be given for other treatments of negation as well.

Example 3: Consider the clauses

$p(M, [E|L], U1, U2) :- E < M, U1 = [E|U1a], p(M, L, U1a, U2).$
$p(M, [E|L], U1, U2) :- E >= M, U2 = [E|U2a], p(M, L, U1, U2a).$

are mutually exclusive given the mode *$langle* **c**, **c**, **f**, **f***$rangle* for *p*, since the predicate ‘*E* >= *M*’ is equivalent to ‘*not*(*E* < *M*)’. ●

4.2.2. Static Optimization based on Mutual Exclusion

A compiler can optimize mutually exclusive clauses in different ways. Cuts are usually not necessary for clauses inferred to be mutually exclusive from Proposition 1 if the only purpose of the cut is to commit to the selected clause, since this can be achieved by using better indexing strategies than those commonly used in Prolog. A compiler aware of this property can, in such cases, generate better index structures and obviate the need for cuts.

Compiler-generated cuts can be inserted in clauses inferred to be mutually exclusive from Proposition 2. In general, additional analysis is necessary to determine that no solutions are being lost in the process of cut insertion. This is illustrated by the following example:

Example 4: Consider the predicate

```
p(X,Y) :– q(X), r(X,Y).
p(X,Y) :– not( q(_) ), s(Y).
```

These clauses are mutually exclusive from Proposition 2. If the predicate *q/1* can be shown to be functional in this context (e.g. assuming knowledge of modes), then this can be transformed to

```
p(X,Y) :– q(X), !, r(X,Y).
p(X,Y) :– s(Y).            •
```

Some researchers have proposed the use of the *if-then-else* construct ('–>') of Prolog instead of cuts [16]. However, the semantics of this construct is also closely tied to the sequential execution strategy of Prolog: the goal ''. . ., p –> q ; r, . . .'' is equivalent to ''. . ., p′, . . .'', where p′ is a new predicate symbol not appearing in the original program, defined by

```
p′ :– p, !, q.
p′ :– r.
```

Notice that in general, the definition

```
p(X,Y) :– q(X), r(X,Y).
p(X,Y) :– not( q(_) ), s(Y).
```

is *not* equivalent to

```
p(X,Y) :– q(X) –> r(X,Y) ; s(Y).
```

They are equivalent if and only if *q*/1 is functional and free of side effects, i.e. the cut implicit in the '–>' does not discard any solutions. On the other hand, if *q*/1 is provably functional, then the cut can be generated by the compiler, so it is not necessary for the programmer to mention it, either explicitly, or implicitly through the '–>'. We would argue that the definition using complementary literals, of the form *q*(*X*) and *not*(*q*(_)), provides a better declarative reading than the version using '–>', since the reader does not have to worry about the fact that ''q(X) –> r(X,Y) ; s(X,Y)'' could fail, even though there might be a value of *X* that satisfied both q(X) and r(X,Y), because this value did not happen to be the first one found.

We would go so far as to argue that if $q/1$ had been intended to behave as a function, then the user ought to have defined it in a manner that made this fact obvious, and left it to the compiler to insert cuts where appropriate.

4.3. Functionality

4.3.1. Inference of Functionality

It does not come as a great surprise that functionality is, in general, undecidable (Sawamura proves the undecidability of determinacy, which is a special case of functionality [20]). However, sufficient conditions can be given for functionality:

Proposition 3: A literal $p(\bar{X})$ is functional relative to mode M if either (i) the predicate p is functional relative to mode M, or (ii) there is a partition $\{\bar{U}, \bar{V}, \bar{W}\}$ of $\bar{X}$ such that (a) $\bar{U}$ functionally determines $\bar{V}$ in p; (b) in any call consistent with M, each term in $\bar{U}$ is ground; and (c) $\bar{W}$ consists only of void variables.

Proof: Case (i) is obvious. For case (ii), note that from the definition of functional dependencies, whenever the arguments $\bar{U}$ are instantiated to ground terms in a call, the arguments $\bar{V}$ are uniquely determined. Since the mode M implies that the arguments $\bar{U}$ are ground, any call consistent with M has the arguments $\bar{V}$ uniquely determined. Since $\bar{W}$ consists entirely of void variables, bindings returned for these variables can be ignored. Thus, the projection of the input-output relation for the literal on the argument positions of $\bar{U} \cup \bar{V}$ is a function for any call consistent with M, i.e. the literal is functional relative to mode M. *$box*

Example 5: Consider a predicate *emp*(*Id, Name, Dept, Sal, PhoneNo*), which is an employee relation whose arguments are the employee's identification number, name, the department he works for, his salary and phone number. Assume that the predicate has the functional dependencies $Id \rightarrow Name$ (''an employee can have only one name'') and $Id \rightarrow Sal$ (''an employee can have only one salary''). Then, the literal

. . ., *emp*(12345, *EmpName*, _, *Sal*, _), . . .

is functional. Here, the arguments {Id, Name, Dept, Sal, PhoneNo} can be partitioned into the sets {*Id*}, {*Name,Sal*} and {*Dept, PhoneNo*} where $Id \rightarrow Name$ and $Id \rightarrow Sal$, *Id* is a ground term in the literal and {*Dept*, *PhoneNo*} correspond to anonymous variables. However, the literal

. . ., *emp*(12345, *EmpName*, _, *Sal*, *PhoneNo*), . . .

may not be functional, since an employee can have more than one phone number. ●

In general, a clause is functional if each literal in its body is functional. It is possible to permit literals that are not functional in the body, as long as they occur in contexts that are functional. The notion of a functional context can be defined more formally as follows:

Definition: A literal *L* occurs in a *functional context* in a clause *C* if the body of *C* is of the form ‘‘G, !, G_1’’ or ‘‘G_1, *not*(G), G_2’’, and *L* occurs in *G*. ●

This definition applies to simple Horn clauses extended with negation. It can be extended in a straightforward way to take other kinds of connectives, such as Prolog’s *if-then-else* construct, into account.

Proposition 4: A clause *C* is functional if any literal *L* in its body that is not functional relative to its mode induced by *M* occurs in a functional context. *$box*

Notice that this proposition does not require the presence of cuts in the clause: if each literal in the body of the clause can be showed to be functional, then the clause can be inferred to be functional even if there are no cuts in the body. The proposition, as stated, suffers from one problem, however, in that it may be sensitive to the order of literals in the body of the clause. Consider, for example, the program defined by

```
q(a,b).
q(c,d).
r(b,c).
r(d,e).
p1(X,Y) :– q(X,Z), r(Z,Y).
p2(X,Y) :– r(Z,Y), q(X,Z).
```

Assume that *p1* and *p2* have mode *$langle* **c**, **d***$rangle*, and that in both *q* and *r*, the first argument functionally determines the second. Each literal in the body of the clause for *p1* is functional from Proposition 3, and hence the clause for *p1* is also functional relative to the mode *$langle* **c**, **d***$rangle*. Now consider the clause for *p2*: the induced mode for the literal ‘‘r(Z,Y)’’ is *$langle* **d**, **d***$rangle*, so the literal is not functional. Further, there is no cut to the right of this literal. Thus, the clause is not functional according to Proposition 4.

The real problem here is that the functional dependencies are being encountered in the wrong order. A simple solution that suggests itself is to try reordering the literals in the body of the clause. However, this does not always work: consider the clause

$p(X, Z)$:– $q(X, Y, U, Z)$, $r(Y, U)$.

Assume that *p* has mode *$langle* **c**, **d***$rangle*; that the functional dependencies $X \rightarrow Y$ and $U \rightarrow Z$ hold in $q(X, Y, U, Z)$, and the functional dependency $Y \rightarrow U$ holds in $r(Y, U)$. Further, assume that both *q* and *r* succeed binding their arguments to ground terms. Then, the clause is functional. However, it cannot be shown to be functional from Proposition 4 by reordering the literals in its body.

The key to the problem lies in determining what other functional dependencies are implied by those that are already known to hold in the clause:

Proposition 5: Let the functional dependencies that hold in a clause C be F, and let the set of variables appearing in the head of the clause be S. Given a mode M for C, let $gd(M) \subseteq S$ be the set of variables appearing in arguments in the head of the clause that are guaranteed to be ground in any call consistent with M, and let $C : gd(M) \overset{*}{\underset{F}{\longrightarrow}} S_0$. Then, C is functional relative to mode M if $S \subseteq S_0$.

Proof: By definition, S_0 is the closure of $gd(M)$ under F. Further, M guarantees that each variable in $gd(M)$ will be instantiated to a ground term in any call to C. It follows that in any call to C consistent with M, each variable in S_0 will be uniquely determined on success. Since $S \subseteq S_0$, it follows that any call to C consistent with M will succeed with its variables uniquely determined, i.e. that C is functional. *$always*

Returning to the example above, given the mode M = *$langle* **c**, **d***$rangle* for the clause

$p(X, Z) :- q(X, Y, U, Z), r(Y, U).$

the set $gd(M)$ is $\{X\}$, and the closure of $\{X\}$ under the functional dependencies $\{X \rightarrow Y, Y \rightarrow U, U \rightarrow Z\}$ is $\{X, Y, U, Z\}$. Since the set of variables appearing in the head of the clause is contained in this closure, the clause is functional relative to mode M. Notice that using this proposition, it may be possible to infer a clause to be functional even though it contains literals that neither have functional dependencies nor are functional, e.g. the clause

$p(X, Z) :- q(X, Y, U, Z), r(Y, U), s(U, W).$

with functional dependencies $\{X \rightarrow Y, Y \rightarrow U, U \rightarrow Z\}$ can be inferred to be functional relative to mode *$langle* **c**, **d***$rangle* irrespective of whether the literal $s(U, W)$ is functional.

The rules given above enable us to reason about the functionality of literals and clauses. The next step is to extend them to allow reasoning about the functionality of predicates. A sufficient condition for the functionality of a predicate in a program is that each clause of the predicate be functional, and further that at most one clause succeed for any call to that predicate in that program:

Proposition 6: A predicate is functional relative to a mode M if its clauses are pairwise mutually exclusive relative to mode M, and each clause is functional relative to mode M. *$box*

These propositions can be used to obtain an algorithm for the inference of functionality. The basic idea in the inference of functionality is to solve a set of simultaneous, possibly recursive, equations over a set of propositional variables. This is similar to the technique for the inference of determinacy used by Mellish [14]. As an example, consider a predicate p whose definition is of the form

(cl_1) $p :- p_{11}, p_{12}, ..., p_{1n_1}.$

(cl_2) $p :- p_{21}, p_{22}, ..., p_{2n_2}.$

. . .

(cl_m) $p :- p_{m1}, p_{m2}, ..., p_{mn_m}.$

By Proposition 8, p is functional if each of its clauses $cl_1, \ldots, cl_m$ is functional, and they are pairwise mutually exclusive. If we strengthen this condition to *if and only if* for the inference procedure,[2] we can set up a set of equations of the form

$$func\_p = MutEx_p \ \$and\ func\_cl_1 \ \$and\ \ldots \ \$and\ func\_cl_{\mathrm{m}}$$

where each of the variables is propositional, $func\_p$ being true only if p is functional, $func\_cl_1$ if clause cl_1 is functional, and so on. $MutEx_p$ is true if the clauses for p are pairwise mutually exclusive, false otherwise. The functionality of each clause depends, from Proposition 4, on the functionality of the literals in its body. This enables us to add the equations

$$func\_cl_1 = func\_p_{11} \ \$and\ func\_p_{12} \ \$and\ \ldots \ \$and\ func\_p_{1n_1}.$$

$$\ldots$$

$$\mathrm{func}\_cl_m = func\_p_{m1} \ \$and\ func\_p_{m2} \ \$and\ \ldots \ \$and\ func\_p_{mn_m}.$$

Each of the variables $func\_p$, $func\_p_{11}$, $func\_cl_1$ etc., are referred to as *functionality status flags*. We also set up equations for the propositional variable $MutEx_p$ if necessary.

The algorithm for functionality analysis takes as input a set of clauses comprising the program, a set (possibly empty) of modes for predicates in the program, and a set (possibly empty) of functional dependencies that hold for predicates in the program. The output is an annotated program, where each predicate, clause and literal is annotated with a bit which indicates whether or not it is functional.

Associated with each literal, clause and predicate A is a bit $A.fstat$, its *functionality status*, that initially has the value **true**. The essential idea behind the algorithm is to first detect predicates whose clauses are not pairwise simple mutually exclusive and set their functionality status bits to **false**, and then to propagate these values in the program call graph in a depth first manner. After this, the the functionality status bits of clauses are set using the values of the functionality status bits of literals.

Nodes in the call graph represent predicates in the program. The node corresponding to a predicate p contains the functionality status bit $p.fstat$ for that predicate (initialized to **true**), together with a bit *p.visited* that initially has the value **0**. Intuitively, if there is an edge from a predicate p to a predicate q, then a literal with predicate symbol q appears in the body of a clause for p We denote the set of edges in the call graph of the program by *CG_EDGES*: if *$langle p, q$rangle* $\in$ *CG_EDGES* then there is a (directed) edge from p to q in the call graph. The graph is represented using adjacency lists. The algorithm also maintains, as an auxiliary data structure, a queue of predicates *RELPREDS* that is initially empty. Pseudocode for the algorithm is given in Figure 1. The worst-case time complexity of this algorithm can be shown to be $O(N + max(p, E) + F)$, where N is the size of the program, E the number of edges in its call graph, p the number of predicates in the program and F the number of functional dependencies in it.

---

[2] Note that this strengthening is conservative, i.e. it may lead to a loss of precision but not of soundness.

4.3.2. Static Optimization Based on Functionality

Information regarding functionality can be used to insert cuts where appropriate in the user's program. It will usually be profitable to insert *savecp/cutto* pairs around functional literals referring to non-functional predicates, as in Example 5 above. If a predicate is itself functional, then it will generally be preferable to insert cuts in the clauses defining it rather than around literals referring to it.

Further optimization is possible if we consider sequences of functional literals. Define a *fail_back_to* relation over pairs of literals, with the following semantics: *fail_back_to*(*p1*,*p2*) is true if execution should backtrack to the goal corresponding to literal *p2* if the goal corresponding to literal *p1* fails. For functional calls, the *fail_back_to* relation is transitive. In other words, given a sequence of literals

. . . p1, . . ., p2, . . ., p3, . . .

where *p1*, *p2* and *p3* are functional literals and the *fail_back_to* relation has the tuples *$langle p3,p2$rangle* and *$langle p2,p1$rangle*, then execution can can fail back directly to *p1* on failure of the goal *p3*, i.e. *$langle p3,p1$rangle* is in the *fail_back_to* relation. This property can be used to produce more efficient code for backtracking in contiguous functional calls. Details are given in [8].

4.4. ''Don't-care'' Nondeterminism

Sometimes we may encounter a computation that is nondeterministic, but where any solution will suffice. In other words, while the computation might generate a number of different results, it is enough to compute any one of them. This kind of nondeterminism is usually referred to as ''don't-care'' nondeterminism.

Since it suffices to compute any one result of a computation in such cases, other solutions for the computation can be discarded. Usually, this is done by adding cuts to the program. As we have seen, however, the use of cuts has several drawbacks. In our system, therefore, we support two primitives that are related to don't-care nondeterminism: *firstof* and *oneof*.

The *firstof* predicate returns only the first solution computed by its argument, according to Prolog's usual evaluation order. Cuts within a *firstof* are soft. This means that the clauses defining a predicate can be understood individually, without requiring knowledge about cuts etc. in the other clauses. This is similar to a the *prove_once* construct of LM-Prolog [10]. The implementation of *firstof* on our system is as follows:

```
firstof(X) :– savecp(CP), call(X), cutto(CP).
```

The predicate *oneof*, on the other hand, returns a single arbitrary element of the set of solutions, with no guarantee on solution order. Since no order is assumed on the solutions, the compiler can perform optimizing transformations involving reordering of literals and clauses, and generate code that can better exploit parallelism in non-sequential implementations.

As an example, consider the following program to color a map consisting of five regions *A*, *B*, *C*, *D*, and *E*, so that no two adjacent regions have the same color:

```
begin
  RELPREDS := nil;                                        /*  Stage I : initialization of flags */
  CG_EDGES := ∅;
  for each literal, clause and predicate A in the program do A.fstat := ⊥;
  for each predicate p in the program do
      if p's clauses cannot be inferred to be pairwise simple mutually exclusive then begin
          p.fstat := false; p.visited := true;
          RELPREDS := push(p, RELPREDS);
      end
      else begin
          for each clause C of p do begin
              for each literal L ≡ q( ... ) in the body of C do
                  if L can be inferred functional from Proposition 3 then L.fstat := true
                  else if L is not in a functional context then CG_EDGES := CG_EDGES ∪ $langle p, q$rangle ;
              if C can be inferred to be functional from Propositions 4 or 5 then C.fstat := true;
          end;
          if C.fstat = true for each clause C of p then begin
              p.fstat := true; p.visited := true;
          end
          else p.visited := false;
      end;      /* if */
  while RELPREDS ≠ nil do begin                           /*  Stage II : iterative propagation  */
      p := head(RELPREDS);
      if ($exists  q)[$langle q, p$rangle  ∈  CG_EDGES and p.fstat = false and ¬q.visited] then begin
            q.fstat := false;
            q.visited := true;
            RELPREDS := push(q, RELPREDS);
      end
      else RELPREDS := pop(RELPREDS);
   end;      /* while */
   for each predicate p in the program do if p.fstat = ⊥ then p.fstat = true;        /*  Stage III : cleanup */
   for each clause C in the program do begin
      for each literal L ≡ q( ... ) in the body of C do if L.fstat = ⊥ then L.fstat := q.fstat;
      if C.fstat = ⊥ then
            C.fstat :=  ($exists  a literal L in the body of C)[¬L.fstat and L is not in a functional context];
   end
end.
```

Figure 1: Algorithm for Functionality Inference

$map\_color(A, B, C, D, E)$:–
    $next(A, B)$, $next(C, D)$, $next(C, E)$, $next(A, C)$,
    $next(A, D)$, $next(B, C)$, $next(B, E)$, $next(D, E)$.

$next(X, Y)$:– $color(X)$, $color(Y)$, $not(X = Y)$.

$color(red)$.
$color(blue)$.
$color(green)$.
$color(yellow)$.

As defined, the predicate *map_color* is nondeterministic, and can produce possible different four-colorings of the given map via backtracking. Now suppose that any one coloring is sufficient. The usual way of doing this would be using a cut:

$map\_color(A, B, C, D, E)$:–
    $next(A, B)$, $next(C, D)$, $next(C, E)$, $next(A, C)$,
    $next(A, D)$, $next(B, C)$, $next(B, E)$, $next(D, E)$, !.

Unfortunately, this enforces the standard execution order of Prolog: there is no way to communicate to the system that any arbitrary four-coloring of the map would be acceptable. On the other hand, if *oneof* were used, an intelligent compiler could transform the clause

$map\_color(A, B, C, D, E)$:–
    $oneof(next(A, B)$, $next(C, D)$, $next(C, E)$, $next(A, C)$,
        $next(A, D)$, $next(B, C)$, $next(B, E)$, $next(D, E))$.

to use a more efficient literal order, e.g. to

$map\_color(A, B, C, D, E)$:–
    $savecp(CP)$,
    $next(A, B)$, $next(A, C)$, $next(A, D)$, $next(B, C)$,
    $next(C, D)$, $next(B, E)$, $next(C, E)$, $next(D, E)$,
    $cutto(CP)$.

The *firstof* and *oneof* primitives have several attractive features: they have a better declarative reading than cuts; they can be manipulated freely by program transformation systems without any of the problems that might be encountered with cuts, since cuts within *firstof* and *oneof* are ‘‘soft’’; and while they can be easily implemented using *savecp/cutto* pairs on a sequential machine, the *oneof* construct does not force any sequentialization of evaluation in a parallel system.

5. Elimination of Failure-Driven Loops

The usual way of specifying an iterative computation in Prolog is via a tail recursive predicate. If the predicate is determinate, tail recursion optimization (see [1, 22]) reclaims space on the local stack.

However, this does not reclaim space on the heap. In the absence of a garbage collector, the only way to reclaim heap space is to have execution fail back. In many cases, this is exactly what the programmer is forced to do. However, the use of failure-driven loops is not unique to primitive systems lacking garbage collectors: experienced programmers sometimes resort to failure driven loops to reduce the overhead involved in garbage collection (see, for example, the AUNT system for VLSI design by Reintjes [17]). As we will see, controlling the backtracking behavior of such loops can be quite tricky, and the correct coding of such loops, in a way that preserves equivalence with the original tail recursive predicate, can be nontrivial. This section discusses how the programmer may be relieved from having to write failure-driven loops explicitly. We first consider how tail recursive predicates can be transformed to a general class of failure-driven loops, called *repeat-fail* loops. This is followed by a discussion of a specialized class of failure-driven loops, called *member-fail* loops, which apply to a very common class of loops – those that iterate through the elements of a list; *member-fail* loops are more efficient than the more general *repeat-fail* loops for such iterations. The section concludes with some general comments on failure-driven space recovery.

5.1. repeat-fail Loops

A failure-driven loop might typically be written using the Prolog predicate *repeat*/0, which is defined as

repeat.
repeat :– repeat.

This predicate succeeds arbitrarily many times when backtracked into. The loop is implemented by having the loop termination clause and the loop body as two branches of a disjunct within a repeat. In the loop body, what would have been the tail-recursive call is replaced by saving the parameters of that call and failing back into repeat. Execution then resumes forward, and the parameters are picked up. On termination, the alternative branches are discarded with a cut (notice that the cut must be ‘‘hard’’, to cut away the other alternative for *repeat* as well), to avoid an infinite loop in case execution fails back to this predicate later.

While such programs do reclaim heap space, they contain most of the ‘‘impure’’ constructs of Prolog – cuts, side effects using *assert* or *record*, etc. – that are widely condemned as making programs hard to understand declaratively. Our approach will be to effect a source-to-source transformation from the original tail-recursive specification to the failure-driven implementation, thereby eliminating the need to use such constructs in the source program. Of course, this may involve a trade-off of space for time, and therefore should be done only when it has been requested by the programmer, e.g. via pragma annotations.

In general, a loop can be written in pure Prolog in the form

p($\bar{X}_1$) :– loop_termination($\bar{X}_1$).
p($\bar{X}_2$) :– loop_body($\bar{X}_2$, $\bar{Y}$), p($\bar{Y}$).

To transform this tail recursive predicate into a failure-driven loop, it is necessary to save the arguments being passed into the tail recursive call before failing back into *repeat*. These can then be picked up going forward. There may also be variables in the call to the predicate that get instantiated in the loop. These instantiations can be returned by a back unification step after succeeding through the loop. Two sets of pointers need to be saved: one pointer to the top level structure that will ultimately be returned, and one set of pointers to the substructures that are being constructed by the next iteration of the loop. Thus, a direct transformation of the original program to a repeat/fail loop yields the program of Figure 2. Here, p1 is a new predicate which handles the actual iteration, and whose name does not appear in the original program. The predicates *save* and *restore* have the expected behavior, viz. save a value over backtracking via a side effect, and unify a term with a saved value, respectively. We do not describe them further, except to note that they are easily implemented in terms of available Prolog primitives such as *record/recorded*, *assert*, etc.

It is straightforward to improve this program to eliminate some redundant *save*s and *restore*s. However, while the failure driven loop illustrated above has the same success behavior as the original tail-recursive predicate, its failure behavior is not the same: e.g. if the execution of *loop_body* fails, then in the original program, the entire loop would fail. In the program of Figure 2, however, execution would

---

p($\bar{X}$) :– save($\bar{X}$),
        save($old($\bar{X}$,$\bar{X}$)), p1, restore($old($\bar{X}$,_)).   /* back unify */

p1 :– repeat,
        restore($\bar{X}$),
            (($\bar{X}$ = $\bar{X}_1$, loop_termination($\bar{X}_1$), !,
              restore($old(Z, $\bar{X}_1$)), save($old(Z, $\bar{X}_1$)))   /* for back unification */
        ;
        ($\bar{X}$ = $\bar{X}_2$, loop_body($\bar{X}_2$, $\bar{Y}$),
              restore($old(Z, $\bar{X}_2$)), /* unify with previous value */
              save($old(Z, $\bar{Y}$)),   /* save for back unification */
              save($\bar{Y}$),   /* save parameters of call   */
              fail)
).

Figure 2

---

backtrack into *repeat*, resume in the forward direction, fail again (either in ‘‘*restore*($\bar{X}$)’’ or in ‘‘*loop_body*($\bar{X}_2$, $\bar{Y}$)’’, depending on the implementation of *restore*), and so on, *ad infinitum*. To preserve equivalence, therefore, some minor modifications need be made, yielding the program in Figure 3. Details of the transformation are given in [4].

The basic repeat/fail scheme illustrated above can be improved substantially with more knowledge about the behavior of the program. For example, with mode information, it is necessary to save only instantiated arguments, and back unify only on free, non-void variables. In addition, read-only parameters, e.g. ground terms such as symbol tables that are passed around the loop, need not be saved and restored at each step. They can be maintained as part of the choice point for *repeat*, by extending it to

```
repeat( _ ).
repeat(X) :– repeat(X).
```

The reader is referred to [4] for further details.

---

p($\bar{X}$) :– save(($\bar{X}$,$\bar{X}$)), p1, restore(($\bar{X}$,_)).   /* back unify */

p1 :– savecp(U),
          repeat,
          restore((Z,$\bar{X}$)),
              (($\bar{X}$ = $\bar{X}_1$, loop_termination($\bar{X}_1$), !, save((Z, $\bar{X}_1$))   /* to back-unify */
)
          ;
          ($\bar{X}$ = $\bar{X}_2$, savecp(V),
              ((loop_body($\bar{X}_2$, $\bar{Y}$), save((Z, $\bar{Y}$)), cutto(V), fail) ;
               (cutto(U), fail))
)
).

Figure 3

5.2. member-fail Loops

A very commonly encountered class of loops is those that iterate over the elements of a list. Such a loop can be written as

```
loop([], []).
loop([E | L], [NewE | NewL]) :- loop_body(E, NewE), loop(L, NewL).
```

The corresponding failure driven loop can be implemented more efficiently as a *member-fail* loop than the *repeat-fail* loops discussed above. The basic idea here is to retrieve successive elements of the input list using the predicate *member*, defined as

```
member(X, [X | L]).
member(X, [Y | L]) :- member(X, L).
```

A first attempt at transforming the *loop* predicate above into a *member-fail* loop yields the following:

```
loop([], []).
loop(L, NewL) :-
        save((L, NewL, NewL)), mf_loop(L), restore((L, NewL, _)).        /* back unify */

mf_loop(L) :-
        member(E, L),
        loop_body(E, NewE),
        restore((L, NewL, [NewE | NewLTail])),
        save((L, NewL, NewLTail)),
        fail.
mf_loop(L) :-
        restore((L, NewL, [])), save((L, NewL, [])).
```

In this, we assume that in a call ''*loop(L1, L2)*'', *L1* is a given list whose elements are to be iterated over, and *L2* is a new list constructed during the processing of *L1*. In general, *L1* may not be ground, and processing its elements may cause variables in both *L1* and *L2* to become instantiated. This accounts for the fact that the input list *L1* has to be saved and restored at each step. Of the triple (*L*, *NewL*, *NewLTail*) that is saved and restored, the first element is the input list, the second is the output list *L2*, and the third is the variable ''tail'' of *L2* that is being extended at each iteration of the loop.

While the success behavior of this loop is identical to the original loop, its failure behavior is not: if, for example, the execution of *loop_body* fails for some element of the list, then the original, tail recursive formulation fails, while the failure-driven version simply picks up the next member of the list and continues. The solution is similar to the corresponding problem in the *repeat-fail* case, and the modified *member-fail* loop is as follows:

```
mf_loop(L) :-
        savecp(X),
        ((member(E, L),
```

```
        (loop_body(E, NewE) -> true ; (cutto(X), fail)),
         restore((L, NewL, [NewE | NewLTail])),
         save((L, NewL, NewLTail)),
         fail
        ) ;  /* member */
        (restore((L, NewL, [])), save((L, NewL, [])) )
       ).
```

The basic *member-fail* scheme shown here can be optimized substantially with more knowledge about the predicate being transformed. For example if, as is often the case, the input list *L1* in a call ‘‘*loop*(*L1*, *L2*)’’ is guaranteed to be ground (in general, if no variable in *L1* is instantiated by *loop*), then it need not be saved and restored repeatedly, and the failure-driven version can be written as

```
loop([], []).
loop(L, NewL) :-
        save((NewL, NewL)), mf_loop(L), restore((NewL, _)).      /* back unify */

mf_loop(L) :-
        ((member(E, L),
          (loop_body(E, NewE) -> true ; (cutto(X), fail)),
         restore((NewL, [NewE | NewLTail])),
         save((NewL, NewLTail)),
         fail
        ) ;  /* member */
        (restore((L, NewL, [])), save((L, NewL, [])) )
       ).
```

The code can be improved still further for predicates which iteratively process a list but neither instantiate variables in it, nor construct a new ‘‘output’’ list, e.g. a compiler which iterates over a list of predicate definitions, generating code for each predicate in turn, which is written to a file or into memory as a side effect. In such cases, the *save*s and *restores* can be eliminated entirely. Thus, a tail recursive predicate of the form

```
loop([]).
loop([H|L]) :- loop_body(H), loop(L).
```

can be transformed to the failure-driven version

```
loop([]).
loop(L) :- mf_loop(L).

mf_loop(L) :-
        savecp(X),
        (member(E, L),
```

$(loop\_body(E) \rightarrow fail \; ; \; (cutto(X), fail))$

).

The code produced in this case is almost exactly what a programmer would write in this case.

5.3. Comments on Failure-Driven Loops

The transformations discussed above preserve equivalence only if the loops being transformed are deterministic (or, at least, functional). Our experience has been that this is not terribly restrictive in practice, since most loops encountered in programs tend to be deterministic. However, if the predicate cannot be proved to be deterministic or functional, space optimizing transformations have to be restricted to the bodies of individual clauses. Details of some restricted transformations that apply to nondeterministic loops may be found in [4].

With this technique, it is possible to reclaim heap space in loops without having to resort to failure-driven loops in implementations lacking garbage collectors. Indeed, one can argue that this could be useful even in situations where garbage collectors are available. Garbage collection, involving scans of both the local stack and the heap followed by compaction, is not an inexpensive operation. Ideally, therefore, when a program is executing and runs out of space, it is necessary to free up not as much space as possible (as a garbage collector would do), but only as much as is necessary to let the program run to completion. Now consider a program which has a few loops that create large structures on the heap but subsequently discards them. By annotating these loops and having the system convert them to failure-driven loops, the user could reclaim a reasonable amount of space – perhaps enough to let the program run to completion – without having to go through explicit garbage collection that might additionally reclaim space that the program will not use. It is important to note that the user has control over which predicates are to be implemented via failure-driven loops: by appropriately annotating such predicates, he can control the tradeoff between saving and restoring of parameters and garbage collection, and tune it to the application. Under such conditions, the transformation could result in a program which, while it avoided impure constructs in the original source, executed more efficiently than a naive tail recursive program relying on garbage collection to reclaim heap space.

6. Implementation

Some of the ideas outlined in this paper – specifically, the inference of functionality and mutual exclusion – have been implemented as part of an experimental compiler for the SB-Prolog system [5]. The inference system, which is written in Prolog, uses a simple mode inference system to infer predicate modes [7]. The system was tested on some simple programs (quicksort, four-queens, a simple rewriting theorem prover) as well as significant modules from the SB-Prolog compiler (the parser, preprocessor, peephole optimizer and assembler). The results of these experiments indicate that the analysis is reasonably efficient (it takes about 2% to 4% of the total compilation time) and that its precision is quite acceptable, e.g. about 65% to 80% of the predicates can typically be inferred to be functional (see Table 1). A closer examination indicates that where the analysis is conservative, it is so principally because of a rather simple mode inference system that is overly conservative for programs that pass around a lot of partially instantiated data structures (this is the case, for example, with the SB-Prolog preprocessor and parser). This suggests that the precision of the inference system could be improved even further given a more sophisticated mode inference system, or via user-declared modes.

7. Conclusions

While ‘‘pure’’ Prolog programs usually have a well-defined declarative reading, programmers often use impure features to improve efficiency. Unfortunately, this may result in a loss of this attractive feature of the language. We have argued that the use of such impure features can in very many cases be rendered unnecessary by optimizing compilers.

As an example, we considered the compiler insertion of cuts into programs. Cuts are considered bad for a number of reasons: they complicate the formal semantics of the language; they make programs hard to understand declaratively; programs containing cuts do not lend themselves to manipulation by program transformation systems in a straightforward way; and the presence of cuts tends to force a sequentialization of execution even when parallel evaluation is possible. We considered three of the commonest uses of cut – for committing to a clause, for discarding alternative solutions for a predicate and for breaking out of failure-driven loops. In each case, we showed how the efficient but impure program could be obtained automatically as a result of source-to-source transformations applied by the compiler to the original, pure program. This supports our view that it is usually not necessary to sacrifice declarative clarity for operational efficiency.

Program	*No. of predicates*	*No. inferred functional*
quicksort	4	4
theorem prover	9	6
fourqueens	11	8
peephole	13	10
assembler	33	21
preprocessor	38	14
parser	42	31
func-inf	47	38

Table 1

References

1. M. Bruynooghe, The Memory Management of PROLOG Implementations, in *Logic Programming*, K. L. Clark and S. Tarnlund (ed.), Academic Press, London, 1982. A.P.I.C. Studies in Data Processing No. 16.

2. K. L. Clark, Negation as Failure, in *Logic and Data Bases*, H. Gallaire and J. Minker (ed.), Plenum Press, New York, 1978.

3. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

4. S. K. Debray, Program Transformations for Failure Driven Space Reclamation in Prolog, Technical Report No. 85/32, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, Dec. 1985.

5. S. K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Tech. Rep. 87-15, Department of Computer Science, University of Arizona, Tucson, AZ, Dec. 1987. (Revised March 1988).

6. S. K. Debray and P. Mishra, Denotational and Operational Semantics for Prolog, *J. Logic Programming 5*, 1 (Mar. 1988), pp. 61-91.

7. S. K. Debray and D. S. Warren, Automatic Mode Inference for Logic Programs, *J. Logic Programming 5*, 3 (Sep. 1988), pp. 207-229.

8. S. K. Debray and D. S. Warren, Functional Computations in Logic Programs, *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), pp. 451-481.

9. N. D. Jones and A. Mycroft, Stepwise Development of Operational and Denotational Semantics for PROLOG, in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, 289-298.

10. K. Kahn, Unique Features of LISP Machine Prolog, UPMAIL Report 14, University of Uppsala, Sweden, 1983.

11. J. W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, 1984.

12. D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.

13. C. S. Mellish, The Automatic Generation of Mode Declarations for Prolog Programs, DAI Research Paper 163, Dept. of Artificial Intelligence, University of Edinburgh, Aug. 1981.

14. C. S. Mellish, Some Global Optimizations for a Prolog Compiler, *J. Logic Programming 2*, 1 (Apr. 1985), 43-66.

15. C. Moss, Results of Cut Tests, in *Prolog Electronic Digest, Vol. 3, No. 42*, Oct 9, 1985.

16. R. A. O’Keefe, On the Treatment of Cuts in Prolog Source-Level Tools, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, 73-77.

17. P. Reintjes, AUNT – A Universal Netlist Translator, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, Aug. 1987, pp. 508-515.

18. J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *J. ACM 12*(Jan. 1965), pp. 23-41.

19. J. A. Robinson, Logic Programming – Past, Present and Future, *New Generation Computing 1*, 2 (1983), pp. 107-124, Springer-Verlag.

20. H. Sawamura and T. Takeshima, Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and Their Applications to Prolog Optimization, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, 200-207.

21. D. H. D. Warren, Implementing Prolog – Compiling Predicate Logic Programs, Research Reports 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

22. D. H. D. Warren, An improved Prolog implementation which optimises tail recursion, Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, 1980. Presented at the 1980 Logic Programming Workshop, Debrecen, Hungary.