Proc. 4th Int'l Conf. on Data Engineering, Los Angeles, CA, IEEE, Feb. 1988

EFFICIENT REORDERING OF PROLOG PROGRAMS

Markian M. Gooley and Benjamin W. Wah

Coordinated Science Laboratory, University of Illinois
1101 West Springfield Avenue, Urbana, Illinois 61801

Abstract- Prolog programs are often inefficient. Prolog execution corresponds to a depth-first traversal of an AND/OR graph; traversing subgraphs in another order can be less expensive. In this paper we show how reordering of Prolog clauses and especially goals can prevent unnecessary search. We characterize the restrictions on reordering and show how they may be detected. We design a new system of calling modes for Prolog, geared to reordering, and discuss ways to infer them automatically. We present an improved method for determining a good goal order for Prolog clauses, and use it as the basis for a reordering system, showing how it can be guided by information about modes and restrictions to generate reordered Prolog that behaves correctly.

1. Introduction

1.1. Motivation for our research

Prolog programs tend to be inefficient. In executing a program, Prolog traverses an AND/OR graph [10] depth first. It will often traverse several large subgraphs, each corresponding to a goal, only to fail on a later goal. On failure, Prolog backtracks, searching the subgraphs again to find another way to satisfy its conjunction of goals. Experienced Prolog programmers, when they can, arrange Prolog clauses so that inexpensive goals that are likely to fail (*e.g.*, tests) are near the beginning of a clause [15]: if an inexpensive goal fails, the rest of a clause need not be evaluated. Similarly, they put clauses last whose heads are unlikely to match a query to their predicate.

Even for small programs, programmers cannot always determine the best order of goals or clauses. They write clauses or goals in an easily-understood order perhaps a poor order indeed, if they seem to have about the same cost and chance of success. For large, complex programs, choosing an order may be difficult: readability wins if it does not appear to slow a program.

1.2. Defining the problem

We want to reorder Prolog clauses and goals to minimize, or at least reduce, the cost of evaluating a query. (We measure this as the number of predicate calls or unifications; CPU time is too imprecise) Reordering should require little intervention from the user. The reordered program need not be formally equivalent to the original, but must be correct in some sense, as we discuss later.

1.3. Assumptions and definitions

The reader should know as much about Prolog as he might by reading the first few chapters of Clocksin and Mellish [3] or Sterling and Shapiro [15]. Examples are in the *de facto* standard DEC-10 Prolog [14] notation.

A predicate with name `name` and number of arguments `arity` is written `name/arity`. A variable not set to an atom or functor is *uninstantiated* or *free;* else it is *instantiated* or *bound.* A tuple of instantiations corresponding to the arguments of a goal is called a *mode;* a goal calls a predicate in one mode and returns from it in (usually) another. If a predicate will not function in certain modes (*i.e.*, it produces a run-time error or an infinite recursion), it *demands* a particular instantiation or mode.

A predicate that performs an action unrelated to the logic of a program and that cannot be undone by backtracking (*i.e.* built-in predicates for I/O or modifying th program) has a *side-effect* and is called *extra-logical.* A predicate that uses no built-in predicates is *pure,* approximating theoretical logic programming under Prolog's execution mechanism [15]. Our techniques are not limited to pure Prolog, though we do assume that the program we are restructuring is free of errors.

We treat certain built-in predicates cursorily; *viz.*, `not/1` and the set-predicates `bagof/3` and `setof/3`. We forbid variable goals, whether written explicitly or using `call/1`, and we do not consider assertion and retraction of clauses: we have insufficient information about these before runtime.

1.4. An example of improvement

We express a family tree in Prolog. Three predicates give relationships: `wife/2`, `mother/2`, and `female/1`. `wife(john,jane).` means that John and Jane are married, `mother(john,joan).` means that Joan is John's mother, and `female(jan).` means that Jan is female. `female/1` is reserved for a female who is not a wife or a mother: assuming no illegitimacy,

```
female(Woman) :- wife(_,Woman).
```

accounts for married women (the underscore is the *anonymous variable*). Thus someone is female if we state that she is, or if she is a wife.

We have a large collection of such facts, and these predicates:

```
grandmother(GC,GM) :-
      grandparent(GC,GM), female(GM).

grandparent(GC,GP) :-
      parent(P,GP), parent(GC,P).

parent(C,P) :- mother(C,P).
parent(C,P) :- mother(C,M), wife(P,M).
```

The query

```
:- grandmother(X,Y).
```

finds all grandmother-grandchild pairs; however, it finds a grandparent-grandchild pair first, instantiating the variables GC and GM, about half the time promptly rejecting it. Changing the first clause to

```
grandmother(GC,GM) :-
      female(GM), grandparent(GC,GM).
```

probably reduces the cost: `female/1` takes at most two calls, whereas `grandparent/2` can take far more. Note also that `female/1` instantiates GM and thus the mode of `grandparent/2`: we consider fewer possibilities, making the goal less costly.. Unless only a tiny fraction of the females in the database are grandmothers, the reordering pays.

1.5. Previous work

Warren [18] gave a method for reordering goals of conjunctive queries. The queries were automated translations of questions in English:

This research was supported by NASA grants NAG 1-613 and NCC 2-481.

a user typed in a question on geography, and a parser generated a query with goals corresponding to the order of the words in the question. This query was often inefficient. Warren gave each mode of each predicate a number: the factor by which the goal multiplies the number of alternatives the system must consider. (A goal `country(C)`, `C` uninstantiated, multiplies the possibilities by the number of countries in the database -- about 150. With `C` instantiated, the goal is a test; it narrows the possibilities and gets a value less than 1.) Warren estimated this number crudely, by dividing the number of tuples of (answers to) a predicate by the product of the sizes of the domains of each instantiated position in the calling mode. (If `borders/2` -- one country borders another -- has 900 tuples, and each argument has a domain size of 150, the function gives 900 for an uninstantiated call, 6 for a partly-instantiated call, and 0.04 for an instantiated call.) Reordering to minimize this yielded speedups up to several hundred times.

Warren reordered only top-level conjunctive queries. We have experimented with reordering the predicates of simple, nearly pure Prolog programs (*i.e.*, a few built-in predicates, but none with side-effects) such as family trees. We tailor a version of the predicate to each mode, renaming both the new version and the goals that call it. This replaces a predicate with a set of new predicates, one for each mode: 2^{arity} if all modes appear. (Arity is usually two for a family tree.) Speedups are typically 5 to 40, rarely under 1 (slowdowns): somewhat better than Warren's, considering that our database of facts is about an order of magnitude smaller than his.

Both Warren's method and our extension have drawbacks. Finding a good heuristic is not easy: it should be easy to evaluate, yet account for both probability and cost of goals. Though it seems effective, Warren's function considers only the number of solutions, not their costs. Further, domain size for an argument is problematic even for database programs [18]. Our extended method is expensive: we call each predicate, forcing repeated backtracking, and count the solution-tuples. This is impractical even for "toy" problems like family trees, and useless for predicates with unpredictable instantiations or data structures.

Li and Wah [10] model Prolog clauses as Markov chains, allowing one to compute approximately the cost and probability of success for a clause from their values for each goal; one need provide them only for grond clauses because they propagate upwards. Goals can be reordered to minimize the cost. We describe this method in detail later because it provides a good way to evaluate different orders of execution.

1.6. Approach and goals of this paper

We describe the problem of inefficient search more precisely, and consider reordering of both clauses and goals. We show how impure features of Prolog restrict reordering, and how to collect information on restrictions. We look at restrictions due to modes, comparing the traditional mode system for Prolog compilers with a new system of *legal modes* suited to reordering and parallel processing. We review ways to infer modes automatically. We extend the Markov-chain model, add impure features and mode restrictions, and look at the problems of implementing a real reordering system. We present current results, summarize, and suggest extensions to our methods and ideas for future work.

2. Equivalence

2.1. Levels of equivalence

A reordered Prolog program will, in general, not behave like its original. By reordering, we have Prolog search a different program tree, simulating an execution mechanism that is more efficient. We willfully change the program, yet we still want it to retain enough equivalence to be useful.

"Equivalence," for the purposes of this paper, means equivalence of output. We informally define four levels of equivalence between a program and a reordered version. They are:

(1) **Reflexive:** The relation between a program and itself. For a given query, reflexively equivalent programs produce the same sets of results in the same order. They fail on the same queries.

(2) **Set:** The reordered program has the same set of results as its original, but they may occur in a different order. The new program's tree (graph, if it is recursive) will have reordered subtrees, but any pruning (as done by the "cut") wil affect the same subtrees as in the original. The programs fail on the same queries.

(3) **Tree:** Pruning of the new program's tree (graph) affects different subtrees, so that the set of results differs from that of the original, and the programs may fail on different queries.

(4) **Inequivalence:** Anything less than tree-equivalence. Ill-defined.

2.2. Applications

Most transformation systems for Prolog try to preserve reflexive equivalence (*e.g.* Debray's method for making tail recursions [7]). This may be too restrictive; for many applications, such as databases, set-equivalence suffices if the order of answers is not important. If only one answer is needed out of a set, tree-equivalence may suffice; in a tree-equivalent reordering, we try to make the Prolog system find the least costly answer and commit to it. In practice, however, the chance that a tree-reordered program will fail when its original succeeded (or vice versa) will render it unacceptable.

The permitted reorderings described in this paper preserve set-equivalence at worst. They can be used freely on predicates for which the order of answers is not vital. We will discuss in detail the role of the cut in destroying set-equivalence.

3. Reordering of clauses and goals

We want to reduce the expected cost of evaluating a Prolog query by reordering the elements of the predicates it calls. Clauses of a predicate and goals of a clause may be reordered, with certain restrictions. The two types of reordering are distinct: they do not compete but are synergistic.

3.1. Reordering of clauses:

In standard Prolog, when a predicate is called its clauses are tried in the order they are written. If the calling goal fails to unify with the head of a clause, or if it does unify but the clause fails despite backtracking amongst its goals, the system tries the next clause. Clause order thereby provides one of the programmer's few control mechanisms.

We put a clause first if it is inexpensive and likely to succeed. What we hope to do to the search tree is illustrated in Figure 3.1; the triangles labeled with probabilities represent subtrees, with size proportional to cost. Before restructuring, the subtrees that correspond to the clauses are in no particular order: we search some large trees likely to fail (*i.e.*, try costly clauses) before searching smaller ones likely to succeed (inexpensive clauses); afterwards, the order is more logical. The search tree does not shrink, however; if we are looking for all solutions to the query, we must search the entire tree anyway and have gained nothing.

Clause reordering can also avoid futile attempts at unification. For instance, many list-processing predicates handle a list recursively, repeatedly decapitating it; a clause with the empty list in its head ends the recursion. For example,

```
length([_|List],C,L):-
        C1 is C+1, length(List,C1,L).
length([],L,L).
```

is in a "good" order: though the second clause costs far less to evaluate than the first, it will match only the empty list. Were it first, the system would attempt unification on every call. *Clause indexing* [17] can have the same effect: at run-time, the system checks the type of several arguments of a call (usually just the first) and tries only the clauses whose corresponding arguments might unify with the call's. However, unless the system always indexes on the proper arguments, reordering will still be useful here.

3.2. Reordering of goals:

Standard Prolog evaluates the goals of a clause in order from left to right. On failure, it successively backtracks, first to alternative clauses of the predicate called by the failing goal, and then to those of earlier goals. (*Intelligent backtracking* [1] allows Prolog to backtrack to the first predecessor goal that might provide fresh instantiations, avoiding more-recent goals that cannot. It is only a palliative: it does not prevent expensive

goals from being tried first. Further, a full implementation requires NP-hard computations [19].) Thus if a clause fails, it is best to have the culprit goal occur as soon in the clause as possible: not only will later goals never be tried, but less backtracking will be attempted. Figure 3.2

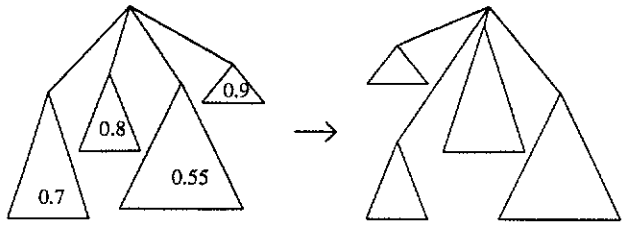

Figure 3.1: Reordering a predicate.

represents a clause as an AND/OR tree. The original version is inefficient: note that if the final goal always fails for a particular call, we needlessly search and backtrack through several large subtrees. The transformed version puts the least expensive goal first: if it fails, the larger subtrees are never tried; if it succeeds, we narrow the possibilities for the later, costlier goals.

As with the extended Warren's method we provide a different version of each predicate for each mode. In the worst case, this again replaces a predicate with 2^{arity} new predicates of the same size. In practice this tends not to occur: predicates of high arity can rarely function in more than two or three modes. Furthermore, high arity is poor programming style [4], and such predicates can usually be replaced by several new ones of lower arity. We trade increased program size for decreased execution time.

3.3. Necessities

In order to go beyond "toy" problems and reorder real Prolog, we must find restrictions on reordering and apply them to a reordering method. Recursive and built-in predicates cause the two classes of restrictions: restrictions on mobility and restrictions due to modes.

Section 4 is about restrictions on mobility. We show how even a single side-effect can make many reorderings impossible, and describe the exceptions. Control predicates and recursions have milder effects, except for the cut. Section 5 is about modes: specifically, the restrictions they cause and a special type of mode intended for reordering: the *legal mode*. Section 6 presents the basic reordering method, first for a subset of pure Prolog, then for real programs: the information gained by the methods of Sections 4 and 5 prevents incorrect reorderings. These techniques work together; Figure 3.3 shows them as modules between which various pieces of information flow: program code, programmer declarations, and information inferred automatically. Note that the system informs the programmer when it cannot infer properties of the program, and when declarations are inconsistent.

4. Restrictions on movement

4.1. Introduction

We cannot reorder carelessly. In all but the simplest programs written in pure Prolog, unconstrained reordering is risky: we cannot guarantee any sort of equivalence, or freedom from infinite recursion.

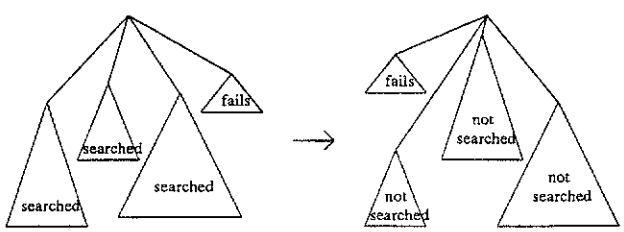

Figure 3.2: Reordering a clause.

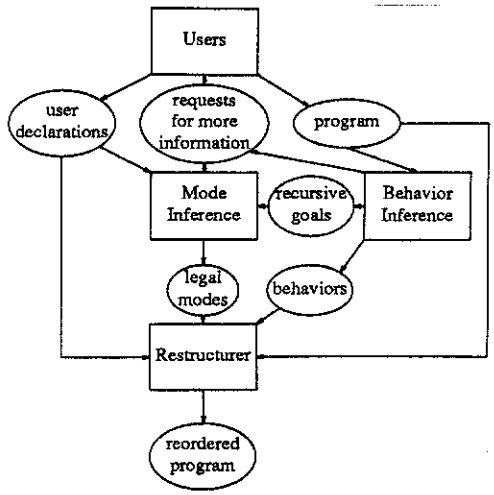

Figure 3.3: The reordering system.

Certain built-in predicates have side-effects that cannot be undone by backtracking; others provide control. Neither can be treated lightly.

4.2. Predicates with side-effects: fixity

Many predicates built into standard dialects of Prolog have *side-effects:* in particular, those that modify the program or do I/O. I/O predicates include `read/1`, `write/1`, `get/1`, `put/1`, and most of the file-handling predicates; predicates that modify a program include `asserta/1`, `retract/2`, and `abolish/2`. It happens that goals calling these predicates are *immobile* within a clause; they therefore act as barriers to permutation of other goals.

Imagine three goals `a`, `b`, and `c` within the body of a clause. `b` has a side-effect. The variables of the goals are instantiated enough that any permutation will work. Exchange `a` and `b`. Suppose that when the clause executes, `a` fails. But `b` has succeeded, and its side-effect cannot be undone by backtracking. Originally `b` would not have been reached, because `a` failed. Similarly, exchanging `b` and `c` is wrong, because if `c` fails it prevents `b` from having its side-effect as in the original program. Unless `a` or `c` is certain to succeed, we cannot move `b`. We may justly call `b` and the like *fixed* predicates, and other predicates *mobile*. Fixed goals can in a few cases be moved (neighbor goals bound to succeed, fixity mode-dependent), but this requires further analysis of the program and may prove unprofitable.

Clauses of a predicate are also fixed by fixed goals. Suppose that `a1` `a2`, and `a3` are clauses of the predicate `a`, and `a2` has a goal that writes. The three clauses have identical heads. Exchange `a1` and `a2`. `a` is called so that any of the three clauses would succeed if selected. Thus `a2` succeeds and writes. Had `a1` been first, it would have succeeded without writing. Now a goal calling `a` might be followed -- perhaps not immediately or even in the same clause -- by a goal that writes. If so, swapping `a1` and `a2` will change the output of the program: `a` will now write before the other goal, when in the original it would not have. Again, further analysis of the program (testing clauses for mutual exclusivity, global analysis of I/O) might show certain reorderings safe, but it might also prove too costly; for now, we consider a clause fixed if it calls a fixed goal.

The restriction is actually far worse than these examples show. Any predicate that has a fixed predicate as a descendant is itself fixed. If `write/1` within a clause of the predicate `w` prints something, a predicate `x` that calls `w` might print as well. A predicate `y` that calls `y` might also print, and so forth. Predicates are responsible for the actions of their descendants: in this way, a single fixed goal can contaminate most of a program.

Before we try to reorder, we must find out which predicates are fixed. We start at a predicate which is not called by any other predicates of the program (an *entry* or *top-level predicate*) and scan its descendants top-down, mimicking program execution. We maintain a list of ancestor

predicates so that when we reach a goal that calls a fixed predicate, we can declare its ancestors fixed as well. We can assert this information or, better yet, store it in a partly-instantiated binary tree (dictionary) [15]; at the end of the process, we write collected information to a file as facts.

4.3. Semi-fixity

Some predicates behave differently in different modes. For example, the built-in `var/1` succeeds only when its argument is uninstantiated. We must preserve the modes of such predicates under reordering, or at least those of culprit variables, or the reordered program will be wrong. Other built-ins and some user predicates share this behavior; usually it involves a unification or test that always succeeds or fails in some mode, and due to a cut has no alternative. For example,

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

will match only its first clause if none of its arguments are instantiated, but probably only its second if the last argument is. Any reordering that changes the latter situation to the former is wrong. We call such predicates *semi-fixed*.

The easiest way to preserve the mode of a culprit variable is to fix the semi-fixed goal with respect to other goals that might change its instantiation; for example, if `s(X,Y)` is semi-fixed in its first argument, and we have

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

What we can do depends on `X`. If `X` is uninstantiated, `a(X)` does not instantiate it, but `d(X)` does, then `a(X)` and `d(X)` may not cross `s(X,Y)`, though `b(Y)` and `c(Y,Z)` might. If we call `t/3` with `X` instantiated, `s(X,Y)` does not restrict reordering. Hence the term "semi-fixed."

Semi-fixity propagates to ancestors if culprit variables also appear in the clause head. It is usually less restrictive than fixity, but it demands far more bookkeeping, and care when aliasing of variables is possible.

4.4. Control predicates

Prolog includes control predicates that allow programmers to alter its default behavior. These predicates can force or prevent backtracking, provide an *if-then-else* construct, or serve as shorthand notation.

4.4.1. The cut

The *cut*, written `!`, is notorious for altering program behavior. When a cut is encountered as a goal during forward execution, it succeeds immediately. When it is reached during backtracking, it acts as a barrier: backtracking must stop at the cut, and no other clauses of the predicate are to be tried. Backtracking to a cut means that the call to its parent predicate has failed.

A cut-bearing clause is essentially fixed within its predicate. If several clauses in a predicate are mutually exclusive for a particular mode, (*i. e.*, for any call in that mode, at most one of their heads will match), they may be swapped even if some of them have cuts. This exception is trivial: the reordering will at most bolster an inadequate indexing system.

In the body of a clause a cut appears to behave almost like a fixed goal: it is not mobile, but it does not fix the ancestor predicates of the clause. Instead it immobilizes every *descendant* of every goal that precedes it in the clause, which is far worse.

Consider the goals that occur before the cut. They instantiate their variables in a particular way, giving a first answer for their conjunction. The cut commits the system to this instantiation. Unless we *know* that reordering the goals or their descendants won't give a different first answer, we cannot reorder any of them. Reordering preserves tree-equivalence but not set-equivalence.

4.4.2. Disjunction

A predicate is a disjunction of its clauses. Prolog, however, allows explicit disjunctions within clauses. If two clauses have identical heads, we can write them as one disjunctive clause. For example,

```
citizen(X) :- native_born(X).
citizen(X) :- naturalized(X).
```

can become

```
citizen(X) :- native_born(X) ; naturalized(X).
```

where `;` represents disjunction. This is not merely shorthand: we can use it to avoid repeating a costly unification. We can also make two clauses that share initial goals into a single disjunctive clause, so that we run the initial goals only once.

Disjunction provides a "semi-permeable" barrier to reordering: clearly we cannot move goals between halves, or between either half and the outside. We can move a goal from outside to inside by putting a copy of it in each half, if it is not fixed or a cut. Once the copy is inside, we can permute it with the rest of its half. Similarly, if we can move duplicate mobile goals in each half to the front or back of their halves, we can replace them with one goal outside the disjunction.

4.4.3. Implication

The implication is Prolog's analog to the *if-then* statements of algorithmic languages. If we can satisfy a set of goals `p`, we try to satisfy a set of goals `q`; written `p -> q`. This can be the first half of a disjunction, forming an *if-then-else*. Implication behaves as if defined by

```
P -> Q :- P, !, Q.
```

We can treat the *then* and *else* much like any other disjunction: a mobile goal common to both can be moved before the premise or after the disjunction. The goals of the premise are immobile, like goals before a cut.

4.4.4. Failure-driven loops

In pure Prolog, a failing clause does nothing, because instantiations it makes are undone on failure. In real implementations, goals call predicates with side-effects that failure cannot undo. For instance, backtracking lets a failing clause execute queries repeatedly, printing the results. Such code is often written for Prolog systems without garbage collection, because heap space is reclaimed on failure.

Several kinds of these loops are common in Prolog programs. We can force failure by using the built-in predicate *fail*; for example:

```
show_all:- t(X,Y,Z), write((X,Y,Z)), nl, fail.
show_all.
```

This prints out all tuples of `t/3`: it finds one, prints it and a newline, and then fails, forcing backtracking to find another; when all are found, the second clause makes the predicate succeed. Goals of a failure-driven loop must remain there; we may reorder them as usual.

4.4.5. Negation

The built-in `not/1`, also written `\+`, has a conjunction of goals for an argument. It behaves as if defined by

```
not(X) :- X, !, fail.
not(X).
```

When it is called as a goal, it succeeds only if its argument fails. `not/1` never instantiates the variables in its argument, but its argument may demand certain instantiations. For now we treat a negation as semi-fixed in all its variables, but reorder multiple goals within its argument.

4.4.6. Set predicates

The built-ins `bagof/3` and `setof/3` generate lists of terms that satify certain properties specified by a conjunction of goals. We reorder of the internal goals, as with `not/1` but treat calls to these predicates as semi-fixed, pending further study.

4.4.7. Recursion

Reordering the goals of a recursive predicate is often unsafe. Consider this example [15] written in pure Prolog:

```
select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Ys]) :- select(X,Xs,Ys).

permutation(Xs,[X|Ys]) :-
        select(X,Xs,Zs), permutation(Zs,Ys).
permutation([],[]).
```

`permutation/1`, given a list as its first argument, produces (on backtracking), all of its permutations. Given a variable instead, it will go into an infinite loop. If we swap the two goals of the first clause of `permutation/1`, we will get an infinite loop if the *second* argument is a variable. Thus, reversing two goals changes a safe mode for an unsafe one.

Better implementations of Prolog include at least a simple form of *tail-recursion optimization* [17]: stack space is reclaimed on each tail-recursive call, saving memory and speeding executing somewhat. It is tempting to reorder a recursion so that the recursive call is final; Debray [7] has a method that works for some cases.

Because we have no reliable way to reorder recursive clauses, we assume for now that the programmer declares a predicate recursive and provides necessary information to the reorderer. We can also detect recursion automatically, using a method similar to the one for finding fixed predicates: traverse the program top-down, keeping a list of predicates being scanned, and check if each new goal is a member of the list.

5. Legal modes

5.1. Introduction

Some Prolog compilers, notably the classic one for the DEC-10 [14], let users specify the modes in which a predicate will be called. The compiler then generates object code for those modes only, code that is more compact and will run faster.

Some Prolog predicates function in more than one mode. When we reorder goals, we usually change their calling modes; therefore, the more modes we can show *legal*, the more reorderings we can do. The usual mode declarations do not tell us what modes are legal, merely which might arise in the original program.

What makes a mode legal? Some goals demand particular instantiations; a legal mode satisfies these demands, without which the goals give run-time errors or go into infinite recursions. We assume that the programmer does not deliberately call any predicate in an illegal mode. We can have him declare certain modes legal -- worthwhile even though time-consuming and error-prone -- or we can infer them automatically.

5.2. Restrictions due to modes

Fixity, semi-fixity, and control structures already reduce the ways to permute the goals of a clause or the clauses of a predicate; modes reduce them further.

5.2.1. Built-in predicates

Many built-in predicates cannot function in some modes. They *demand* certain of their arguments to be instantiated, or demand certain modes. For example, `functor/3` is a built-in predicate. If its first argument is a term, it instantiates its other arguments to the term's name and arity; similarly, given a name and number as its last two arguments, it constructs a new term as its first. Give `functor` only a name or arity, and it will not work: in C-Prolog [13], for example, it fails without a name and gives a run-time error without an arity.

5.2.2. Recursive predicates

Recursive predicates chiefly build or modify recursive structures such as lists or trees. Given the wrong modes, they enter unproductive infinite recursions, or produce infinite sets of solutions on backtracking. Consider `delete/3`:

```
delete(X,[X|Y],Y).
delete(U,[X|Y],[X|V]):- delete(U,Y,V).
```

With an atom as its first argument and a list of atoms as its second, `delete` sets its third to the list with one instance of the atom deleted. However, if only its first argument is instantiated, it produces an infinite set of solutions. `b` appears in successive positions of the list `X` *ad infinitum*.

Unfortunately, we cannot always tell beforehand what modes yield infinite recursions. A solution sufficient for most cases will probably show that an infinite recursion in a particular mode a structure is essentially unchanged though growing larger, and nothing in the predicate can alter it [9]. For now, the programmer must declare legal modes for a recursive predicate; these may allow a goal to be moved. Otherwise we consider it semi-fixed.

5.3. Mode systems

5.3.1. Traditional

The DEC-10 Prolog compiler [14] and some successors let the programmer specify modes in which a predicate will be called. A clause of the form `:- mode <predicate>(<modes>).` assigns `<predicate>` the modes denoted by the tuple `<modes>`. `<modes>` has one element, or *mode item*, for each argument of the predicate; an element is `+` if the argument will be instantiated, `-` if it will be uninstantiated, or `?` if it might be either (so that a tuple of all `?` is superfluous [14]). The compiler generates code only for the modes declared: compilation is faster and the code more efficient and compact.

5.3.2. Our system: legal modes

The DEC-10 system won't do for reordering. We want the modes in which a predicate delivers a useful result, not those arising in the original program, DEC-10 declarations can give a superset of the actual modes; ours must give a (preferably improper) subset of the legal modes: an illegal mode makes a program wrong.

5.3.2.1. Description

Our system must be suitable both for declaring and inferring modes. Legal modes are verbose yet precise: tedious work for the programmer. To avoid confusion with DEC-10 modes, we change notation. "-" becomes `u`, "+" becomes `i`, and "?" becomes `a`. `i` is only for terms with no uninstantiated elements, and `a` is both "don't care" (like "?") and "don't know." We introduce other mode items presently.

Certain modes are safe for a given predicate: its legal *input modes*. Each predicate also has an *output mode*, at least as instantiated as its input mode. For instance, `delete/3` has the pairs `[(a,i,a), (i,i,i)]`, `[(u,u,i), (u,i,i)]`, and `[(i,a,i), (i,i,i)]`.

A three-symbol system is not expressive enough, due to *coupled variables* and *partially-instantiated structures*. *Coupled* (or *aliased*) variables have different names but have been unified together, losing their distinctness; instantiating one instantiates the other. In generating possible reorderings, we update instantiations so that we reorder goals safely. We show two variables as coupled by marking their positions in the output mode with the same symbol: `c3`, say, for the third coupled pair.

Missed couplings make instantiated variables appear uninstantiated, needlessly constraining reordering (though not creating illegal calls). We assume that the programmer is always right, and allow his original order even if it seems illegal. Thus the reordering is sound, if only, as a last resort, by not being a reordering at all. In practice, we do try to detect coupling and propagate its effects.

A recursive structure may contain uninstantiated variables. If we treat it as instantiated and feed it to a recursive predicate that demands an instantiated argument, the predicate might, after a few calls, expose a variable and go into an infinite recursion. Consider

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

In `(i,u,u)` it appends the variable of the second argument to the list in the first, yielding a list whose tail is the variable: a form of a useful structure called a *difference list* [15].

Now consider:

```
build(L1,L2,L3,L4):-
    transform(L2,L2a), transform(L3,L3a),
    append(L1,L2a,L2b), append(L2b,L3a,L4).
```

We call it in mode `(i,i,i,u)`; `transform/2` works in modes `(i,a)`, and `append/3` in modes `(i,a,a)` and `(a,a,i)`. Now we try to reorder the clause. If we say that `append(i,u,u)` returns `(i,u,u)` (the third argument has a variable in its tail), this good order is rejected:

```
build(L1,L2,L3,L4):-
    append(L1,L2a,L2b), transform(L2,L2a),
    append(L2b,L3a,L4), transform(L3,L3a).
```

On the other hand, if we say that `append(i,u,u)` returns `(i,u,i)`, this illegal order is generated:

```
build(L1,L2,L3,L4):-
    append(L1,L2a,L2b), append(L2b,L3a,L4),
    transform(L2,L2a), transform(L3,L3a).
```

We must forego the first rather than risk the second; can we do better?

Making the mode system more complex makes it more expressive. We give it the flavor of a typing system [12], adding notations for structures and lists that are incomplete (or not known to be complete). Our solution is tentative; we hope to improve it.

A list known to be fully instantiated has mode `i`; otherwise, it has an item of the form

```
(<elt. 1>,...,<elt. *>,...,<elt. n>,<tail>).
```

There is one sub-item for each known element of the list and one for the tail. If the length of the list is known exactly, each element gets a single item. If not, any known leading or trailing elements each get an item, and the other elements are lumped together and given a single mode, with an asterisk indicating zero or more items. The tail is `i` for lists of determinate length, `u` otherwise; recall that the list-constructor function "." has two arguments, so that the list `[a,b,c,d]` is `.(a,.(b,.(c,.(d,[])))).` Thus we might describe one mode of `append/3` as `[(l(a*,i),u,u),(l(a*,i),u,l(a*,u))]`: a list of determinate length but unknown types of elements generates a similar list of indeterminate length.

Functors other than lists, unless known to be fully instantiated, are treated similarly, with mode items of the form `f(<arg 1>,<arg 2>,...)`. If the arity of the functor is known, each `<arg i>` is a mode item; else we treat them as with lists. We describe one mode of the built-in predicate `=..` (or `univ`) [3], as `[(f(a*)),(l(i,a*,i))]`: a term of unknown arity and contents becomes a list of unknown size and contents, constant head, and determinate length.

We have these mode items:

`u`: uninstantiated argument
`a`: argument of uncertain instantiation
`i`: instantiated argument
`ci`: member of the `i`th coupled set
`l`: partially instantiated list
`f`: partially instantiated term (not a list).

We add `p`, found only in output modes: it preserves the mode of the argument; in other words, the predicate does not alter the argument. Neither do predicates alter instantiated arguments; their items are placeholders in output modes.

5.3.2.2. Uses

We designed these modes for reordering, but they may have other uses. They might help for running Prolog on multiprocessors: for instance, they might show that certain goals need not depend on predecessors and might be able to run in parallel with them. Other researchers have recognized that precise mode declarations may be useful for parallel processing [2].

5.4. Mode inference

Not much has been published on automatic mode-inference. Mellish [11] had success with a mode system intermediate in complexity between the DEC-10 system and ours. Tung and Moldovan [16] presented a method for finding modes as part of their work on AND-parallelism. Debray [5] developed a useful method, with accompanying theory, but it generated only DEC-10 modes. He transformed the original program into a simplified version that did nothing apart from propagating instantiations, and then ran this version to infer the modes. We believe that his method can be adapted to inferring legal modes, but have no results at present.

6. Markov-chain method for reordering

We want the expected cost and probability of success for a predicate call in a legal mode, given those for the predicates its clauses call. We model a Prolog clause as an absorbing Markov chain [8], after Li and Wah [10]. The cost and probability of the clause follow from properties of its chain; combining results for clauses gives them for the predicate.

6.1. Pure Prolog

We give the method for a subset of pure Prolog; *viz.*, no recursions, and variable arguments in a clause head unless the clause is a fact. We extend it to full pure Prolog, then to real-life Prolog.

6.1.1. Overview

Consider `k:- a, b, c, d.` We know the probability p_i and cost c_i for each goal i. To find the expected probability and cost of a single solution to the clause, we model it as the Markov chain of Figure 6.1. We give each goal a state, label the arcs with success and failure probabilities, and add absorbing states S and F for success and failure.

Transitions of the chain mimic Prolog execution. We start in the state of b, proceeding to c if b succeeds, or going to F if it fails. In every i we move forward (probability p_i), or backtrack (probability $1-p_i$). Eventually we stop in an absorbing state: success or failure.

If we want the cost of finding all solutions to a conjunction of goals, as in a *bagof*, a *setof*, or a failure-driven loop, we add an arc of probability 1 from S back to the last goal. When we find a solution we backtrack, looking for another, until we fail.

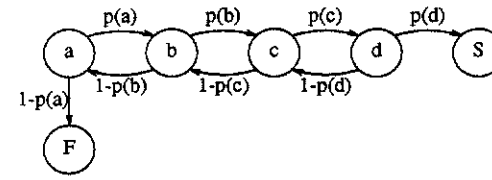

Figure 6.1: A clause body as a Markov chain.

6.1.2. Deriving probability and cost for a clause

The success probability p_{clause} of the clause is that that the process ends in state S. $\sum c_i v_i$, for all goals i, is the expected cost of a solution, c_i being the expected cost of goal i and v_i the mean number of visits by the process to state i. Therefore we calculate p_{clause} and all the v_i: textbook [8] mathematics.

Begin with the transition matrix P. An element p_{ij} is the probability of a transition from state i to state j; S has index 1 and F index 2. The chain has r states, s of which are transient. Consider four submatrices of the $r \times r$ matrix P:

$$P = \begin{bmatrix} I_{r-s} & 0 \\ R & Q \end{bmatrix}$$

I_{r-s}, the identity matrix, for transitions between absorbing states; R, $s \times (r-s)$, for transitions into absorbing states; Q, $s \times s$, for transitions between transient states. The zero matrix, $(r-s) \times s$, shows that nothing leaves an absorbing state. Here is P for our example:

$$P_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1-p_a & 0 & p_a & 0 & 0 \\ 0 & 0 & 1-p_b & 0 & p_b & 0 \\ 0 & 0 & 0 & 1-p_c & 0 & p_c \\ p_d & 0 & 0 & 0 & 1-p_d & 0 \end{bmatrix}$$

The matrix $N = (I-Q)^{-1}$ provides our answers. Its first row contains the numbers of visits v_i for the transient states, because we start in the first state. The product NR gives the success probability p_{clause}: the first element of the column of state S (column 1); we take the dot product of that column of R and the first row of N. We find N_k numerically; if we have N goals, we invert an $N \times N$ matrix, and do $2N$ multiplications and sundry additions for the probability and cost. We might call a C routine from Prolog to generate and invert the matrix.

6.1.3. Applying the results

For each calling mode we find the least costly reordered version of a predicate. Different modes may need different orders, and the cost of a

goal varies with its mode. For the subset of pure Prolog we are considering, we can reorder the goals as we please, choosing the best permutation. An N-goal clause has $N!$ permutations; for $N>4$ we can save effort by culling useless ones (*e.g.*, an expensive goal likely to succeed should not be first). After finding probability and cost for each clause, we combine the probabilities to find the conditional probabilities for the clauses in order, multiply these by the costs, and get the expected probability and cost for the entire predicate.

6.1.4. Initial probabilities and costs

Cost and probability of a clause come from those of its goals; in our subset of Prolog, these come from costs and probabilities of facts. Facts may have non-variable arguments, so that their heads might fail. Evaluating a fact costs one call; its success probability is problematical unless the call is uninstantiated, when it is unity. We can, like Warren [18], find *domains* for each argument; a domain might be the set of constants in that argument of the predicate, or perhaps the set of all constants in the program. We take the probability as $\prod_i |domain_i|^{-1}$ for every position i with a constant in both fact and call.

If we allow full pure Prolog with recursion, it becomes hard to tell the likelihood that a call unifies. if an argument position is non-variable in both goal and clause head, the probability is no longer 1. For constants, we can use domains, as with facts. For structures, the predicates are usually recursive, with a special case matching empty structures. The size of a structure affects both the probability and cost of a call; the system or user must estimate structure size before run-time. Some predicates may require the user to provide an explicit probability of unification for each argument of each clause head.

6.2. Restricted (real-world)

6.2.1. Restrictions

The restrictions discussed earlier forbid many attractive orders, but they greatly reduce the number of permutations we examine. For example, if the third goal of a five-goal clause is fixed, the number plummets from 5!=120 to 2!2!=4.

Control predicates both restrict reordering and generate special Markov chains. A cut makes a chain as in Figure 6.2: the goal after the cut does not backtrack to it, but fails. A disjunction makes a split chain, as in Figure 6.3. An *if-then-else* makes a chain that combines features of the two previous chains (Figure 6.4). this emphasizes that the implication is essentially a cut with effects local to its clause.

Every goal must make a legal call to its predicate. A reordering that prevents this, instantiating a goal improperly, is rejected. We generate a potential order by instantiating a clause head with the mode and scanning the clause goal by goal, keeping track of the variables each goal demands and instantiates. As soon as an illegal mode arises, we backtrack to generate another order, so that we test only legal orders.

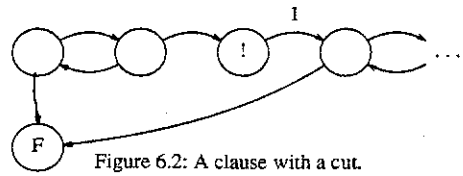

Figure 6.2: A clause with a cut.

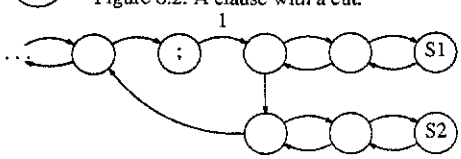

Figure 6.3: A disjunctive clause.

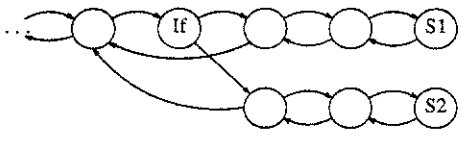

Figure 6.4: An *if-then-else* implication.

6.2.2. Implementation

6.2.2.1. Necessities

Reordering requires much information about a program, provided by the programmer or, preferably, inferred automatically. This includes:

Legal modes for each built-in or user predicate, declared or inferred
Unification probabilities for certain clause heads in certain modes
Probabilities and costs for built-in predicates
A list of entry points
A list of the fixed predicates
A list of the recursive predicates
Probabilities and costs for recursive predicates.

These are Prolog facts, declared in the source file, in files generated by inference programs, or in a hand-written file of information about built-in predicates. We can read them into the clause space of a Prolog interpreter, or compile them and load their object code; however, probabilities and costs should be a dictionary, to be joined by others being inferred for the reordered program. This lets us access all such information uniformly without the clumsiness of asserting new facts.

6.2.2.2. Execution

The reorderer loads the program and the extra facts. Starting at an entry point, it traverses the program depth-first until it reaches a clause whose goals all have known probability and cost. It checks for control predicates and further information about the goals, such as legal modes and mobility. Choosing a legal mode of the predicate, it permutes the body, discarding incorrect permutations. When it makes a correct one, the reorderer calls a routine written in C, giving it the probabilities and costs in order, and the structure of the clause body; the routine returns the expected probability and cost. The reorderer tries all correct versions and keeps the least costly. It prints the new clause under a new name and stores its probability and cost in the dictionary. It repeats this for the other clauses and the other legal modes.

When a predicate is complete, the reorderer combines the stored probability and cost values with any estimates of unification probability, to get the predicate's probability and cost, which it stores. Working upwards, the reorderer handles every user predicate in the program, changing goal names to correspond to the new predicate names.

7. Performance

7.1. What programs benefit?

Our reordering methods are most useful for predicates with these properties:

(1) **Mobility:** We need something to reorder: several goals per clause or clauses per predicate that can be moved. Restrictions and short clauses prevent this.

(2) **Nondeterminism:** If a predicate is deterministic, reordering cannot improve its execution: our system cannot generate an improved deterministic algorithm.

(3) **Diverse probabilities and costs:** Similar ones give us no guidance; we cannot determine a good order.

(4) **Unbalanced search graph:** A balanced search graph exacerbates the lack of the previous property.

Prolog databases, some search programs, and nondeterministic programs in general gain the most. Standard benchmark programs violate at least the first two of our criteria: they are largely deterministic, and there is little in them to reorder.

7.2. Reordered programs

Table 7.1 shows some results of reordering a family-tree program written in nearly pure Prolog. 55 constants in the program represent people. We called each predicate in each mode, with one call for each possible instantiation. Therefore, testing mode `(u,u)` required one call, modes `(u,i)` and `(i,u)` required 55 apiece, and modes `(i,i)` required 3025. All results were generated by an instrumented version of C-Prolog 1.5 [13]. We give figures for the original program, the reordered version, and for the cheapest reordering possible (found by exhaustive enumeration when practical).

Here are pertinent predicates of the original program. There are also 10 facts for `girl/1`, 19 for `wife/2`, and 34 for `mother/2`.

```
female(X) :- girl(X).
female(X) :- wife(_,X).
male(X) :-not(female(X)).
father(X,Y) :-mother(X,Z),wife(Y,Z).
parent(X,Y) :-mother(X,Y).
parent(X,Y) :-father(X,Y).
married(X,Y) :-wife(X,Y).
married(X,Y) :-wife(Y,X).
siblings(X,Y) :-mother(X,Z),mother(Y,Z),unequal(X,Y).
unequal(X,Y) :- X==Y.
sister(X,Y) :-siblings(X,Y),female(Y).
brother(X,Y) :-siblings(X,Y),male(Y).
grandmother(X,Y) :-parent(X,Z),mother(Z,Y).
cousins(X,Y) :-parent(X,Z),parent(Y,W),siblings(W,Z).
cousins(X,Y) :-parent(X,Z),parent(Y,W),siblings(W,V),
       married(V,Z).
aunt(X,Y) :-parent(X,Z),sister(Z,Y).
aunt(X,Y) :-parent(X,Z),brother(Z,W),wife(W,Y).
```

Table 7.1: Results of reordering a family tree program.

Predicate	Mode	Number of calls: original	Markov	optimal
`aunt`	`(u,u)`	1452	985	
	`(u,i)`	61872	1409	
	`(i,u)`	2035	2035	
	`(i,i)`	93354	67238	
`brother`	`(u,u)`	792	792	792
	`(u,i)`	4555	539	530
	`(i,u)`	1004	1004	1004
	`(i,i)`	16003	21449	16003
`cousins`	`(u,u)`	62987	1477	
	`(u,i)`	106606	2034	
	`(i,u)`	63570	2203	
	`(i,i)`	138088	151062	
`grandmother`	`(u,u)`	230	200	200
	`(u,i)`	11254	367	367
	`(i,u)`	548	548	548
	`(i,i)`	28426	18705	18705

Gains are most impressive for the half-instantiated modes. Apparently, partial instantiation is exploited by a good goal order, so that large search trees are pruned early. This does not happen so readily for mode `(u,u)`: the first goal is called uninstantiated, and so must be called in its most expensive form. Similarly, for mode `(i,i)`, enough variables are instantiated that goal order is not crucial; sometimes reordered clauses are more expensive in this mode.

Table 7.2 shows results for several "toy" programs that query small databases. `p58` is Problem 58 from "How to solve it in Prolog" [20]; `meal` plans meals and `team` generates project teams. Only a single clause of `p58` and `meal` can be reordered, and only four clauses of `team` on two levels, so that the performance gains are less impressive.

Table 7.2: Results of reordering several small programs.

Name and mode	Original version	Reordered version
`p58(u,u)`	121	78
`meal(u,u,u)`	17626	16691
`meal(I,i,u)`	8011	7554
`team(u,u)`	34682	9990
`team(u,i)`	33292	8600

8. Conclusions

We have presented a method for reordering Prolog for more efficient execution. We have shown the information necessary to keep a reordering correct, how much of this can be inferred automatically, and how much must still be provided by the programmer. We have introduced a new system of modes which may also be useful for parallel execution. We have extended the Markov-chain reordering method to cover full Prolog, and presented some preliminary results of our work.

Much work remains to be done. We need to see if the mode system is expressive enough, and improve it if necessary. We need better techniques for recursive predicates; the system should derive modes and estimate probabilities and costs, relying upon the programmer only as a last resort. If analytical methods fail, we may have to implement an expert system to handle certain predicates. We can also exploit other properties of Prolog predicates, such as functionality [6]. We intend to investigate various parallel dialects of Prolog. Finally, we should integrate our techniques into one system, so that we can provide a program as input and, with slight intervention by the user, receive a reordered, improved program as output.

References:

[1] M. Bruynooghe and L. M. Pereira, "Deduction Revision by Intelligent Backtracking," in: J. A. Campbell, *ed., Implementations of Prolog*, Ellis Horwood, Chichester, England, 1984.

[2] J.-H. Chang, A. M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis," *Digest of Papers of COMPCON Spring '85*, pp. 218-225, February 1985.

[3] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1984.

[4] J. K. Debenham, Style in Logic Programs, Technical Report 85.7, School of Computing Sciences, New South Wales Institute of Technology, Australia.

[5] S. K. Debray, Automatic Mode Inference for Prolog Programs, Technical Report 85/019, June 1985, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

[6] S. K. Debray, Detection and Optimization of Functional Computations in Prolog, Technical Report 85/020, August 1985, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

[7] S. K. Debray, Optimizing Almost-Tail-Recursive Prolog Programs, Technical Report 84/089, November 1984, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

[8] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, Van Nostrand, New York, 1965.

[9] R. Kowalski, *Logic for Problem Solving*, North-Holland, New York, 1979.

[10] G.-J. Li and B. W. Wah, "How Good are Parallel and Ordered Depth-First Searches?" *Proceedings of the International Conference on Parallel Processing*, pp. 992-999, IEEE, August 1986.

[11] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming*, vol. 1, pp. 43-66, 1985.

[12] A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," *Artificial Intelligence*, vol. 23, pp. 295-307, 1985.

[13] F. C. N. Pereira, C-Prolog User's Manual, version 1.5.

[14] L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren, User's Guide to DECsystem-10 Prolog, September 1978.

[15] L. Sterling and E. Y. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[16] Y.-W. Tung and D. I. Moldovan, "Detection of AND-Parallelism in Logic Programming," *Proceedings of the International Conference on Parallel Processing*, pp. 984-991, IEEE, August 1986.

[17] D. H. D. Warren, Applied Logic -- Its Use and Implementation as a Programming Tool, Technical Note 290, June 1983, SRI International, Menlo Park, CA.

[18] D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Seventh Conference on Very Large Data Bases*, IEEE, 1981.

[19] D. A. Wolfram, "Intractable Unifiability Problems and Backtracking," *Third International Conference on Logic Programming*, pp. 107-121, July 1986.

[20] H. Coelho, J. C. Cotta, and L. M. Pereira, How to Solve it in Prolog, 3rd. ed., July 1982, Laboratório Nacional de Engenhara Civil, Lisbon, Portugal.