Efficient Reordering of Prolog Programs

MARKIAN M. GOOLEY, STUDENT MEMBER, IEEE, AND BENJAMIN W. WAH, SENIOR MEMBER, IEEE

Abstract—Prolog programs are often inefficient: execution corresponds to a depth-first traversal of an AND/OR graph; traversing subgraphs in another order can be less expensive. We show how reordering of clauses within Prolog predicates, and especially of goals within clauses, can prevent unnecessary search. We characterize and show how to detect restrictions on reordering. We propose a new system of calling modes for Prolog, geared to reordering, and discuss ways to infer them automatically. We summarize the information needed for safe reordering, and consider which types can be inferred automatically and which must be provided by the user. We present an improved method for determining a good order for the goals of Prolog clauses, and use it as the basis for a reordering system, showing how the proper information can guide it to generate reordered Prolog that behaves correctly.

Index Terms—Best-first search, logic programming, Markov chains, Prolog, query optimization.

I. Introduction

A. Motivation for Our Research

PROLOG programs tend to be inefficient. The inference engine that executes a program must traverse an AND/OR graph [14] depth first. It will often traverse several large subgraphs, each corresponding to a goal, only to fail on a later goal. On failure, the engine 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 [23]: 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 (but a good compiler or Prolog engine renders this less useful, as we shall see).

The best order for goals or clauses is often problematic, even for small programs; in some cases it will be different for different calls of a predicate. Programmers tend to write code in orders that are easy to understand, especially if the constituent goals or clauses seem to have about the same cost and chance of success. For large, complex predicates, efficiency may well be sacrificed for clarity.

B. Defining the Problem

We want to reorder Prolog clauses and goals to minimize, or at least reduce, the cost of evaluating a query.

Manuscript received March 1, 1989; revised August 15, 1989. This work was supported in part by the National Aeronautics and Space Administration under Grants NAG 1-613 and NCC 2-481.

The authors are with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign Urbana, IL 61801.

IEEE Log Number 8931635.

(We measure this as the number of predicate calls or unifications; CPU time is too coarse a measure and sometimes misleading.) Ideally, a system for reordering should require little intervention from the user; in practice, we find that programs often need annotations. The reordered program need not be formally equivalent to the original, but must be correct in some sense, as we discuss later.

C. Assumptions and Definitions

The reader should know as much about Prolog as he might by having read the first few chapters of an introductory text, such as Clocksin and Mellish [6] or Sterling and Shapiro [23]. Examples are in the *de facto* standard DEC-10 Prolog [20] 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, one that cannot be undone by backtracking (i.e., built-in predicates for I/O or modifying the 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 [23]. Our techniques are not limited to pure Prolog, although 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 run-time.

D. An Example of Improvement

We express a family tree in Prolog. These predicates give the 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, and 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: thus, 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.

E. Previous Work

Warren [25] presented a method for reordering goals of conjunctive queries in Prolog. The queries were automated translations of questions in English: a user typed in a question on geography, and a parser generated a query. The order of the goals in the query corresponded to the order of the words in the question. Such orders were often inefficient. Warren gave each goal of each predicate a number: the factor by which the goal multiplies the number of alternatives the system must consider. (A goal country(C), with C uninstantiated, multiplies the number of possibilities by the number of countries in the database—about 150. With C instantiated, the goal is a test; it eliminates possibilities and gets a value less than 1.) Warren estimated this number crudely: he divided 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. Although 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 [25]. 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 would obviously be useless for predicates with unpredictable instantiations or data structures.

Li and Wah [14] 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 ground 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.

F. 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.

II. Equivalence

A. 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 as follows.

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") will affect the same subtrees 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. (This happens, for instance, when goals preceding a cut are reordered.) The new program produces results that are a subset or superset of the original's.

4) Inequivalence: The results of the new program are not necessarily a subset or superset of the original's; side-effects may also be expressed too often, not often enough, or in the wrong order.

B. Applications

Transformation systems for Prolog's usually try to preserve reflexive equivalence (e.g., Debray's method for making tail recursions [8]). This may be too restrictive; for many applications, such as databases, the order of answers is not important and set-equivalence suffices. If only one answer, any answer, is needed out of a set, tree-equivalence may suffice; in a tree-equivalent reordering, we try to make the Prolog engine 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) probably renders 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.

III. 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.

Standard Prolog, in executing a predicate, tries its clauses in the order they are written. Should the calling goal fail to unify with the head of a clause, or if it does unify but the clause fails despite backtracking among its goals, the system tries the next clause. Similarly, the Prolog engine evaluates the goals of a clause in order from left to right; on a failure, it successively backtracks, first to alternative clauses of the predicate called by the failing goal, and then to those of earlier goals.

Intuitively, we can see that we should try to put a clause early in a predicate if it is likely to succeed and is inexpensive to evaluate: we get an initial answer to a query as quickly and inexpensively as possible. Similarly, we should try to put a goal early in a clause if it is inexpensive and likely to fail: if the clause does fail, we have made it fail as inexpensively and cheaply as possible.

Note that there is some formal justification for choosing these new orders for clauses and goals. Consider again a Prolog program as an AND/OR graph, with predicates as OR-nodes and clauses as AND-nodes. Nodes of one type have children of the other. Suppose that for a depth-first search (Prolog execution) the children i of a node (the clauses of a predicate or the goals of a clause), succeed with independent probabilities p_i (and, hence, fail with probabilities $q_i = 1 - p_i$), and cost c_i to execute. Li and Wah [14] have proven that ordering the children of an OR-node (clauses) by decreasing (p_i/c_i), and the children of an AND-node (goals) by decreasing (q_i/c_i) minimizes the expected cost of a depth-first search. Of course, the costs and success-probabilities of clauses and goals in Prolog are not independent of each other: in fact, they can vary greatly with order. Still, this result shows that the intuitive ordering has some theoretical basis.

A. Reordering of Clauses

Fig. 1 shows an example of what we seek to do. The subtrees are marked with probabilities of success p, costs c (boldface), and the ratio (p/c) (italic).

If we need only one answer to a predicate, reordering of clauses makes it likely that we find the answer of lowest cost. (Of course, if we need all answers, we have gained nothing: the search tree is no smaller than before, and we must still search it all.) For example, consider again the predicate depicted in Fig. 1. Suppose that we want only a single solution. If we assume that the success-probabilities for the clauses are independent, the expected cost for a single solution of the original tree is $(0.7)(100) + (1 - 0.7)(0.8)(100 + 80) + (1 - 0.7)(1 - 0.8)(0.5)(100 + 80 + 100) + (1 - 0.7)(1 - 0.8)(1 - 0.5)(0.9)(100 + 80 + 100 + 40)$, or 130.24. The (optimal) expected cost for the new tree is only 49.64.

Reordering of clauses can also prevent futile attempts at unification. For instance, many predicates that process lists 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: although the second clause costs far less to evaluate than the first, it will match only the empty list. Were it first, the Prolog engine would attempt unification on every call. *Clause indexing* [27] can have the same effect: at run-time, the engine checks the type of the arguments to a call (usually just the first argument) and tries only the clauses for which those arguments and the call's might unify. However, unless the engine always indexes on the proper arguments, reordering can still be useful here.

a clause if it is inexpen-
lause does fail, we have
d cheaply as possible.
l justification for choosing
d goals. Consider again a
graph, with predicates as
nodes. Nodes of one type
pose that for a depth-first
children i of a node (the
oals of a clause), succeed
p_i (and, hence, fail with
cost c_i to execute. Li and
dering the children of an
ng (p_i/c_i), and the chil-
y decreasing (q_i/c_i) min-
pth-first search. Of course,
ies of clauses and goals in
ach other: in fact, they can
this result shows that the
oretical basis.

what we seek to do. The
bilities of success p, costs
c) (italic).
to a predicate, reordering
we find the answer of low-
ced all answers, we have
is no smaller than before,
1.) For example, consider
n Fig. 1. Suppose that we
we assume that the success-
independent, the expected
original tree is (0.7)(100)
80) + (1 − 0.7)(1 −
+ (1 − 0.7)(1 − 0.8)(1
00 + 40), or 130.24. The
new tree is only 49.64.
also prevent futile attempts
any predicates that process
repeatedly decapitating it;
its head ends the recursion.

t,C1,L).

h the second clause costs far
it will match only the empty
engine would attempt unifi-
indexing [27] can have the
engine checks the type of the
just the first argument) and
ich those arguments and the
unless the engine always in-
nts, reordering can still be

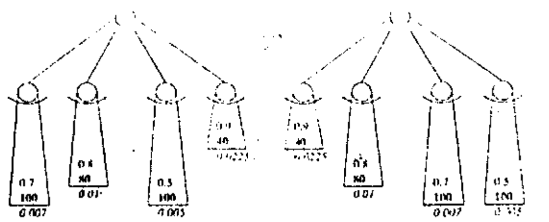

Fig. 1. Reordering a predicate.

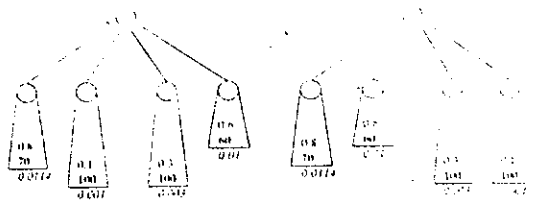
Fig. 2. Reordering a clause.

B. Reordering of Goals

Fig. 2 shows an example for this case. This time the probabilities q are for failure, and the ratio given (again in italics) is (q/c). For this example, we can readily show how reordering will reduce the expected cost of a failure. Originally, a failure costs (0.8)(70) + (0.2)(0.1)(70 + 100) + (0.2)(0.9)(0.3)(70 + 100 + 100) + (0.2)(0.9)(0.7)(0.6)(70 + 100 + 100 + 60), or 98.928; after reordering, the expected cost is only 78.968.

Apart from tending to make failing clauses fail earlier, reordering of goals also reduces backtracking. By way of comparison, *intelligent backtracking* (IB) [3] lets Prolog backtrack to the first predecessor goal that might provide fresh instantiations, avoiding later goals that cannot. Thus, IB is only a palliative: it does not prevent expensive goals from being tried first, as reordering does. Further, a full implementation requires NP-hard computations [29].

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. Thus, we make a program larger that it might be faster: a program's dynamic use of memory, not the size of its code, is usually what limits its utility.

C. Necessities

In order to go beyond "toy" problems and reorder real Prolog, we must determine what constructs restrict reordering, and take the restrictions into account for any method of reordering. Recursive and built-in predicates cause the two classes of restrictions: restrictions on mobility and restrictions due to modes.

Section IV 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 V is about modes: specifically, the restrictions they cause and a special type of mode intended for reordering: the *legal mode*. Section VI presents the basic reordering method, first for a subset of pure Prolog, then for real programs: the information gained by the methods of Sections IV and V prevents incorrect reorderings. These techniques work together; Fig. 3 shows them as modules

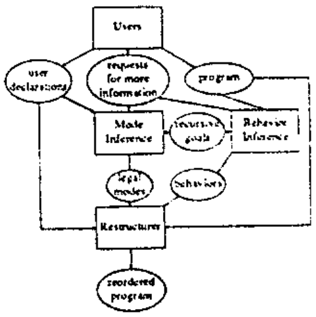

Fig. 3. The reordering system.

between which various pieces of information flow: program code, declarations by programmers, and information inferred automatically. Note that the system informs the programmer when it cannot infer properties of the program, and when declarations are inconsistent.

IV. Restrictions on Movement

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

B. Predicates with Side-Effects: Fixity

Many predicates built into standard dialects of Prolog have *side-effects*: in particular, predicates that do input/output (I/O). (Predicates that modify the program, such as **asserta/1**, **retract/2**, and **abolish/2**, which we are not considering, also have side-effects.) I/O predicates include **read/1**, **write/1**, **get/1**, **put/1**, and most of the file-handling predicates. 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**, in that order although not necessarily consecutive, 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, the failure of a would have kept b from being reached. 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 (if neighboring goals are bound to succeed, if fixity depends on mode), but this requires deeper 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 in a way that will make any of the three clauses succeed when it is 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 whether clauses are mutually exclusive, analyzing I/O globally) might show certain reorderings to be 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 x 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) [23], writing the collected information to a file as facts.

C. Semifixity

Some predicates return very different results 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 the arguments responsible for the changed behavior; otherwise, the reordered program will be wrong. Both built-ins and some user predicates can depend on modes in this way: usually the problem is a unification or test that always succeeds or fails in some mode, but 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. A reordering that changes the latter situation to the former is wrong. We call such predicates *semifixed*. In this case the third argument is the *culprit*, and a variable as the third argument of a call to a would be a *culprit variable*.

An easy way to preserve the mode of a culprit variable is to fix the semifixed goal with respect to other goals that might change the variable's instantiation; for example, suppose that s(X,Y) is semifixed in its first argument, and we have

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

If X is uninstantiated and a(X) does not instantiate it, but d(X) does, then a(X) and d(X) may not cross s(X,Y), although 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 "semifixed.")

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

D. Control Predicates

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

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: such reordering will at most bolster an inadequate indexing system.

At first glance, a cut appearing in the body of a clause seems to behave almost like a fixed goal: it is not mobile, but it does not fix the ancestors of the predicate containing the clause. Instead, it immobilizes every *descendant* of every goal that precedes it in the clause, which is far more restrictive.

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

2) Disjunction: A predicate is a disjunction of its clauses. Prolog, however, also 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 always mere shorthand: sometimes it saves repeating a costly unification. We can also, side-effects permitting, make two clauses that share initial goals into a single disjunctive clause, so that the initial goals run only once.

Disjunction provides a "semipermeable" 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.

3) Implication: The implication is Prolog's analog to the *if–then* statements of algorithmic languages. Only if we can satisfy a set of goals p do 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, exactly like goals before a cut.

4) Failure-Driven Loops: In pure Prolog, a failing clause does nothing, because the instantiations it makes are undone on failure. In real implementations, goals call predicates with side-effects that failure cannot undo. For instance, a clause can call complicated predicates, print the results of their computation, and then fail. Prolog systems without garbage collection often require such code, because stack and heap space is reclaimed on failure.

Several varieties 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 new-line character, 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 within it, but we may reorder them as usual.

5) Negation: The built-in not/1, also written \+, takes 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 semifixed in all its variables, but reorder multiple goals within its argument.

6) Set Predicates: The built-ins bagof/3 and setof/3 generate lists of terms that satisfy certain properties specified by a conjunction of goals. We reorder the internal goals, as with not/1 but treat calls to these predicates as semifixed, pending further study.

7) Recursion: Reordering the goals of a recursive predicate is often unsafe. Consider this example [23] 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 partial form of *tail-recursion optimization* [27]. A *tail-recursion* is one in which the recursive call is the last goal of its clause; the optimization lets stack space be reclaimed on each such call, saving memory and speeding execution somewhat. It is tempting to reorder a recursion so that the recursive goal is last: Debray [8] has a method that works for some cases, but we know of none that works in general.

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 easily 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.

Table I summarizes the results of this section. For each type of restriction, it shows the constructs responsible for it, the effects they have upon clauses and goals, and the way that they might propagate to other parts of the program.

V. Legal Modes

A. Introduction

Some Prolog compilers, notably the classic one for the DEC-10 [20], let users specify the modes in which a predicate will be called. The compiler then generates object

TABLE I
RESTRICTIONS ON REORDERING

Restriction	Cause(s)	Effect		Propagation
		on goals of clauses	on clauses of predicates	
Mode	Demands of built-in predicates; recursions	order must satisfy demands	none	demands may pass to ancestors
Fixity	Predicates with side effects (e.g. I/O)	goal immobile within clause	clause immobile within predicate	ancestors become fixed
Semi-fixity	Certain success (failure) in some modes	fixed with respect to certain variables	none	ancestors semi-fixed (depends on variables)
Immobility	Cut (!) as goal	can't reorder goals before cut	clauses can't cross bearer of cut	descendants of goals before cut: immobile
Control	Explicit disjunction: a ; b	confined to halves of disjunction	none	none
Control	Implication: if -> then ; else	if immobile; then, else confined	none	none
Recursive	Recursions	avoid orders that cause infinite loops	same	none

code for those modes only; the result is more compact and faster than the general-purpose version.

Many 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 "safe" or *legal*, the more reorderings we can attempt. The usual mode declarations do not tell us what modes are legal, merely which might arise during the execution of the original program.

What makes a mode legal? As we have seen, 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—often worthwhile even though time-consuming and error-prone—or we can infer them automatically.

B. Restrictions Due to Modes

Fixity, semifixity, and control structures already reduce the ways that we may permute the goals of a clause or the clauses of a predicate; modes reduce them further. Built-in, semifixed, and recursive predicates are the usual sources of mode restrictions.

Most built-in predicates have modes in which they cannot function: they demand that certain subsets of their arguments be correctly instantiated. An example is the built-in functor/3: 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 SB-Prolog [10], for example, it gives a run-time error and fails.

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 a copy of the list with one instance of the first-argument atom deleted. However, if only its first argument is instantiated, it produces an infinite set of solutions. The first argument appears in successive positions of a list *ad infinitum*.

Unfortunately, we cannot always tell beforehand what modes yield infinite recursions. A solution sufficient for most cases will probably involve showing that for an infinite recursion in a particular mode, a structure grows larger but is essentially unchanged, and nothing in the predicate can stop the growth [13]. Barring that, the programmer can declare particular modes legal for a predicate, or run a program that infers modes automatically (discussed presently). The recursive predicate can then be considered to demand these modes.

C. Mode Systems

Mode declarations for the DEC-10 Prolog compiler [20] (and many successors) are clauses of the form :- mode <predicate>(<modes>). Such a clause 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 ?'s is superfluous).

The DEC-10 notation, although we will consider modifications of it, suffices for reordering; the conventional mode declarations themselves do not. Recall that for safe reordering we need the modes in which a predicate delivers a useful result, not the modes arising in the original program. DEC-10 declarations need not be exact: they can give a superset of the actual modes. If they include modes that never actually arise, the only bad effect is time wasted in compilation and execution. Ours must give a (preferably improper) subset of the legal modes: any illegal mode makes a program wrong.

Certain modes are safe when calling a given predicate: its legal *input modes*. Each predicate also has an *output mode*, at least as instantiated as its input mode. We can write input and output modes as pairs: for instance, delete/3 has the pairs [(?,+,?), (+,+,+)], [(+,?,+), (+,+,+)], and [(−,−,+), (−,?,+)]. The last of these may seem incorrect, but with only its third argument instantiated (to a list), delete inserts its first argument into a copy of its third, and returns the result in its second. This result is *partly-instantiated*, and we use ? to show that.

D. Mode Systems and Run-Time Tests

A three-symbol system might not seem expressive enough. One apparent problem is *coupled* (or *aliased*) variables. These have different names but have been unified together, losing their distinctness: instantiating one instantiates the other. If two variables are coupled but we take no account of it, one might appear to be uninstantiated when neither is: this would not cause illegal calls, but it might prevent legal reorderings.

Partly-instantiated structures cause another potential difficulty. In our last example with delete, we mentioned that it can make a list containing uninstantiated variables; we considered the list to have the mode-item ?. Had we treated it as + and fed it to a recursive predicate that demands an instantiated argument, the predicate would after a few calls have exposed the variable and gone (say) into an infinite recursion. Consider

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

In mode (+,−,−) 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* [23].

Now consider

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

We call build/4 in mode (+,+,+,−); transform/2 works in modes (+,?), and append/3 in modes (+,?,?) and (?,?,+). Now we try to reorder the clause. If we say that append(+,−,−) returns (+,−,−) (the third argument has a variable in its tail, after all), the following 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(+,−,−) returns (+,−,+), 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?

In an earlier version of this paper [11] we presented a scheme which made the legal-mode system more expressive by making it more complex. We gave it the flavor of a typing system [18], [22] by adding notation for incomplete structures and lists, aliasing, and so forth. Such a complex system of modes might allow better reorderings, but it makes annotation difficult and error-prone, and stultifies automatic programs for mode-inference [9].

Testing instantiations at run-time can yield many of the advantages of a complex system of modes. The necessary tests are already part of Prolog, e.g., nonvar/1 succeeds if its argument is instantiated. ==/2 succeeds only for a pair of identical terms (such as aliased variables). Such tests involve only comparison of flags or addresses [26]. Suppose that a promising reordering is safe only if particular variables, whose runtime instantiations we are unsure of, are instantiated. We can replace the original clause's body with an *if-then-else*: the tests are the *if*, the reordered version is the *then*, and the original is the *else*. If the variables pass the tests, we use the new order and gain efficiency; if they fail, we use the original order and lose only the cost of the tests.

E. Mode Inference

There has been increasing interest over the past few years in *abstract interpretation* or *flow analysis* [2], [5], [9], [15]–[17], [28] of Prolog programs. The idea is to execute a program symbolically over an *abstract domain*, usually a complete lattice or complete partial ordering of finite height. This "abstract execution" yields a finite description of the program's behavior during an actual execution [28]. To infer modes, we let the elements of the abstract domain be the mode-symbols.

Most researchers consider mode-inference as an application when they present methods for abstract interpretation; Bruynooghe and his colleagues [2], [4], and Debray and his [9], [28], appear to place the most emphasis on modes. Debray's work is particularly interesting because, rather than finding modes by executing a program on an abstract interpreter, he generates a transformed program that, when executed conventionally, yields the mode information. Running the transformed program takes far less time than interpreting its original. It appears to be possible to modify Debray's technique to find legal modes: it even generates output modes internally, so that we need only save them [9].

VI. Markov-Chain Method for Reordering

We want to know the expected cost and probability of success for a legal call to a predicate, given those for the predicates its clauses call. We model the body of a Prolog clause as an absorbing Markov chain [12] after Li and Wah [14]. The expected cost of executing the clause body, and the probability that it succeeds, both follow from properties of its chain. We combine these with the probabilities that the clause heads will match the call, and from this obtain the results for the predicate.

A. 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.

1) Overview: Consider k:− a, b, c, d. Suppose that we know the success probability p_i and c_i for each goal i. To find the expected probability and cost of only a single solution to the clause body (e.g., if we called k/0 interactively and wanted only one solution, or for k as a goal preceding a cut), we model it as the Markov chain of Fig. 4. We give each goal a state, label the arcs with success and failure probabilities, and add absorbing states S (only one solution) and F for success and failure.

Transitions of the chain mimic Prolog execution. We start in the state of a, proceeding to b if a 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 is more usual, we add an arc of prob-

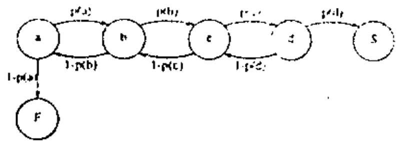

Fig. 4. A clause body as a Markov chain (single solution).

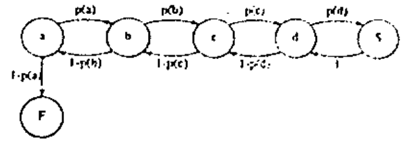

Fig. 5. A clause body as a Markov chain (all solutions).

ability 1 from S back to the last goal, as in Fig. 5. When we find a solution we backtrack, looking for another, until we fail.

Note that this model only approximates the way the Prolog engine executes a clause body. It is the basis of a *heuristic* method: it glosses subtleties of execution that would make analysis impractical. The Prolog engine preserves its state on a stack [26], but a Markov process is memoryless, keeping no history of how it reaches a state. A goal within a clause body succeeds some number of times (perhaps zero), with varying probabilities, and then fails; a transition between states of the chain has a single fixed probability. Probabilities that goals succeed are not necessarily independent; probabilities of transitions are. In return for making these approximations, we obtain a practical way to predict the behavior of a clause.

2) Deriving Probability and Cost for a Clause Body: To find the probability p_{body} that a clause body succeeds, we use the single-solution chain of Fig. 4, even if we are looking at a multiple-solution clause. We take p_{body} as the probability that the process ends in state S. For the expected cost of a solution we use the appropriate chain. $\Sigma_{i=1}^{n} c_i v_i$, for the n goals of the body, is the expected cost of all the solutions put together, c_i being the expected cost of goal i, and v_i the mean number of visits by the process to state i. Hence, the expected cost for solving a single-solution clause is

$$c_{single} = \sum_{i=1}^{n} c_i v_i$$

and the expected cost for one solution of a multiple-solution clause is

$$c_{multiple} = \frac{1}{v_S} \sum_{i=1}^{n} c_i v_i$$

where v_S is the number of visits to the success-state S.

Calculating p_{body} and all the v_i is textbook [12] mathematics. Begin with the transition matrix P for the single-solution case. 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. Because we start in the first state, its first row contains the numbers of visits v_i for the transient states. The product NR gives the p_{body}, the probability of success: 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. If the reorderer is written in Prolog, we might call a routine written in a more-suitable language, such as C, to generate and invert the matrix.

For the multiple-solution chain, S is no longer an absorbing state, and for our example the matrix P becomes

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

It happens that in this case there is a tidy form for the v_i, so that the expected cost of a solution can be written as

$$c_{multiple} = \frac{1}{v_S} \sum_{i=1}^{n} c_i v_i = \sum_{i=1}^{n} c_i \left(\prod_{i=1}^{n} \frac{p_{i-1}}{1 - p_i} \right).$$

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 first, we can reorder the goals as we please, choosing the best permutation. An n-goal clause has $n!$ permutations; for $n > 3$, trying all of these can be expensive.

We can adapt an idea presented by Smith and Genesereth [21] for reordering conjunctive queries whose conjuncts perform no inference: use a best-first search such as A^* [1] to decide the least-expensive order of goals. We do a best-first search on a tree of ordered subsets of the clause body, using as a heuristic the expected cost of finding all solutions to the Markov chain for a subset. Every time we expand a node, we choose the one with the least-expensive sequence. Clearly, the all-solutions formula gives an admissible heuristic for A^* [1], because the cost the formula gives for a partial sequence of goals is always an underestimate of that for the complete sequence that begins the same way.

4) Initial Probabilities and Costs: Cost and probability of a clause come from those of its goals; in the subset of Prolog we are considering at this point, these come from costs and probabilities of facts. Facts may have nonvariable arguments, so that their heads might fail. Evaluating a fact costs one call; its success probability is problematic unless the call is uninstantiated, when it is unity. We can, like Warren [25], 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 $\Pi \, |domain_i|^{-1}$ 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 has a variable in neither goal nor clause head, the probability is no longer unity. 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 sizes before run-time. Some predicates may require the user to provide an explicit probability of unification for each argument of each clause head.

B. Restricted (Real-World)

1) Restrictions: The restrictions discussed earlier forbid many attractive orders, but they greatly reduce the number of permutations we need 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 restrict reordering as well. We can model them using special Markov chains [11], but this is impractical because it is easier to handle the separate conjunctive parts as if they were bodies of short clauses.

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.

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

legal modes, input and output, 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 put in 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.

The reorderer loads the program and the extra facts. Starting at an entry point, it traverses the program depth-first until it reaches a predicate whose clauses have goals with known probability and cost; then it works on the clause one by one. If a clause contains control predicates, the reorderer divides it into blocks of goals, deciding which require the one-solution chain and which the multiple-solution. Choosing a legal mode of the predicate, it propagates to the goals the instantiations produced by that mode.

The reorderer looks, in order, at the one or more blocks of goals from the clause. If a block is immobile, it uses the one-solution Markov chain, calling an external predicate (written in C) to construct and invert the matrix; this gives probability and cost for the block. It permutes other blocks exhaustively and computes their cost, saving the least expensive order; or, if too many permutations are possible, it reorders them using best-first search. It finds the success-probability by, for the best order, evaluating a one-solution chain. It discards illegal orders.

Combining the probabilities and costs for the blocks of goals yields values for the clause body; combining results for clause bodies with match probabilities for heads yields an estimated probability and expected cost for a call to the reordered predicate in that legal mode. The reorderer stores the reordered version under a new name, along with mode, probability, and cost. It repeats this for every legal mode of the predicate. Working upwards, the reorderer handles every user predicate in the program, changing goal names as necessary to correspond to the new predicate names.

VII. Performance

Our reordering methods are most useful for predicates with the following properties.

1) Mobility: We need something to reorder: several mobile goals in a clause or clauses in a predicate. Restrictions and short clauses prevent this.

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

3) *Diverse Probabilities and Costs:* If probabilities and costs of goals are too similar for different goals, they give no guidance towards a better order.

4) *Unbalanced Search Graph:* A balanced search graph exacerbates the lack of the previous property: again, goals appear too similar.

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.

Table II 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 (−,−) required one call, modes (−,+) and (+,−) required 55 apiece, and modes (+,+) required 3025. All results were generated by an instrumented version of C-Prolog 1.5 [19]. We give figures for the original program, the reordered version, and for the cheapest reordering possible (found by exhaustive enumeration when practical).

Fig. 6 gives pertinent predicates of the original program. There are also 10 facts for girl/1, 19 for wife/2, and 34 for mother/2.

Fig. 7 gives reordered versions of the above predicates. (Clearly, predicates with clauses of one goal cannot be reordered.) This is essentially raw output from the reorderer, with variables represented as numbers preceded by underscores.

Note the new names for the versions of predicates that are tuned to a particular mode: the terminal letters are for uninstantiated and instantiated. Each predicate that the user might call interactively will need a dummy predicate, for example:

```
aunt(X,Y) :-
  var(X) →
    ( var(Y) →
        aunt_uu(X,Y)
    ;
        aunt_ui(X,Y)
    )
  ;
    ( var(Y) →
        aunt_iu(X,Y)
    ;
        aunt_ii(X,Y)
    )
  ).
```

This looks complicated, but actually the Prolog engine needs merely to test two tag bits. In many cases, the reorderer produces only one or two distinct versions of a predicate, allowing fewer clauses and tests.

TABLE II
RESULTS OF REORDERING A FAMILY-TREE PROGRAM

Predicate	Mode	Number of calls			ratio of improvement
		original	Markov	optimal	
aunt	(−,−)	1452	985		1.47
	(−,+)	61872	1409		43.91
	(+,−)	2035	2035		1.00
	(+,+)	93354	67210		1.39
brother	(−,−)	792	792	792	1.00
	(−,+)	4555	539	530	8.45
	(+,−)	1094	1094	1004	1.00
	(+,+)	16003	21449	16003	0.75
cousins	(−,−)	62987	1477		42.65
	(−,+)	106606	2035		52.40
	(+,−)	65530	2203		29.86
	(+,+)	138008	151062		0.91
grandmother	(−,−)	230	200	200	1.15
	(−,+)	11254	367	367	30.66
	(+,−)	549	549	548	1.00
	(+,+)	28426	18705	18705	1.52

```
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).
```

Fig. 6. Predicates from the family-tree program.

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 (−,−): the first goal is called uninstantiated, and so must be called in its most expensive form. Similarly, for mode (+,+), enough variables are already instantiated that goal order is not crucial; sometimes reordered clauses are more expensive in this mode.

We also restructured some rules from a corporate database (over 100 employees) written in Prolog; Table III summarizes the results. (These results and those in Table IV were generated using SB-Prolog 2.3 [10].) The facts in this database are indexed on the employee identification number; once that is instantiated, many goals of the rules become trivial. Reordering essentially becomes a way to make the rules find, as quickly and inexpensively as possible, the smallest superset of these numbers whose owners satisfy the rule.

Table IV shows results for several other programs; most of them query small databases. p58 is Problem 58 from "How to solve it in Prolog" [7]; meal plans meals and team generates project teams. kmbench is a substantial program: a theorem-prover running a set of benchmark problems. Only a single clause of p58, meal, and kmbench can be reordered; only four clauses of team on two levels: the gains in performance are less impressive than with our other programs.

W
mor
tion
of thi
still
a no
reord

```
father_uu(_5804,_5805):-wife(_5805,_5806),mother(_5804,_5806).
father_ui(_5804,_5805):-wife(_5805,_5806),mother(_5804,_5806).
father_iu(_5804,_5805):-mother(_5804,_5806),wife(_5805,_5806).
father_ii(_5804,_5805):-wife(_5805,_5806),mother(_5804,_5806).
siblings_uu(_7234,_7235):-mother(_7234,_7236),mother(_7235,_7236),unequal(_7234,_7235).
siblings_ui(_7234,_7235):-mother(_7235,_7236),mother(_7234,_7236),unequal(_7234,_7235).
siblings_iu(_7234,_7235):-mother(_7234,_7236),mother(_7235,_7236),unequal(_7234,_7235).
siblings_ii(_7234,_7235):-unequal(_7234,_7235),mother(_7234,_7236),mother(_7235,_7236).
sister_uu(_7434,_7435):-female(_7435),siblings_ui(_7434,_7435).
sister_ui(_7434,_7435):-female(_7435),siblings_ui(_7434,_7435).
sister_iu(_7434,_7435):-siblings_iu(_7434,_7435),female(_7435).
sister_ii(_7434,_7435):-siblings_ii(_7434,_7435),female(_7435).
brother_uu(_8054,_8055):-siblings_uu(_8054,_8055),male(_8055).
brother_ui(_8054,_8055):-male(_8055),siblings_ui(_8054,_8055).
brother_iu(_8054,_8055):-siblings_iu(_8054,_8055),male(_8055).
brother_ii(_8054,_8055):-siblings_ii(_8054,_8055),male(_8055).
grandmother_uu(_9346,_9347):-parent_uu(_9346,_9348),mother(_9348,_9347).
grandmother_ui(_9346,_9347):-mother(_9348,_9347),parent_ui(_9346,_9348).
grandmother_iu(_9346,_9347):-parent_iu(_9346,_9348),mother(_9348,_9347).
grandmother_ii(_9346,_9347):-mother(_9348,_9347),parent_ii(_9346,_9348).
cousins_uu(_8342,_8343):-married_uu(_8344,_8345),parent_ui(_8342,_8345),
        siblings_ui(_8346,_8344),parent_ui(_8343,_8346).
cousins_uu(_8332,_8333):-parent_uu(_8332,_8334),siblings_ui(_8335,_8334),
        parent_ui(_8333,_8335).
cousins_ui(_8332,_8333):-parent_iu(_8333,_8334),siblings_iu(_8334,_8335),
        parent_ui(_8332,_8335).
cousins_ui(_8342,_8343):-parent_iu(_8343,_8344),siblings_iu(_8344,_8345),
        married_iu(_8345,_8346),parent_ui(_8342,_8346).
cousins_iu(_8342,_8343):-parent_iu(_8342,_8344),married_ui(_8345,_8344),
        siblings_ui(_8346,_8345),parent_ui(_8343,_8346).
cousins_iu(_8332,_8333):-parent_iu(_8332,_8334),siblings_ui(_8335,_8334),
        parent_ui(_8333,_8335).
cousins_ii(_8332,_8333):-parent_iu(_8332,_8334),parent_iu(_8333,_8335),
        siblings_ii(_8335,_8334).
cousins_ii(_8342,_8343):-parent_iu(_8342,_8344),married_ui(_8345,_8344),
        parent_iu(_8343,_8346),siblings_ii(_8346,_8345).
aunt_uu(_11311,_11312):-wife(_11313,_11312),brother_ui(_11314,_11313),
        parent_ui(_11311,_11314).
aunt_uu(_11302,_11303):-sister_uu(_11304,_11303),parent_ui(_11302,_11304).
aunt_ui(_11311,_11312):-wife(_11313,_11312),brother_ui(_11314,_11313),
        parent_ui(_11311,_11314).
aunt_ui(_11302,_11303):-sister_ui(_11304,_11303),parent_ui(_11302,_11304).
aunt_iu(_11302,_11303):-parent_iu(_11302,_11304),sister_iu(_11304,_11303).
aunt_iu(_11311,_11312):-parent_iu(_11311,_11313),brother_iu(_11313,_11314),
        wife(_11314,_11312).
aunt_ii(_11311,_11312):-wife(_11313,_11312),brother_ui(_11314,_11313),
        parent_ii(_11311,_11314).
aunt_ii(_11302,_11303):-parent_iu(_11302,_11304),sister_ii(_11304,_11303).
```

Fig. 7. Reordered predicates from the family-tree program.

TABLE III
RESULTS OF REORDERING A CORPORATE DATABASE PROGRAM

Name and mode	Original version	Reordered version	Ratio of improvement
benefits(-,-)	1288	551	2.34
pay(-,-,-)	1112	1112	1.00
pay(-,jane,-)	55	55	1.00
maternity(-,-)	294	142	2.07
maternity(-,jane)	20	20	1.00
average_pay(-,-)	611	611	1.00
fms(-,-)	284	243	1.17
fms(-,jane)	29	29	1.00

TABLE IV
RESULTS OF REORDERING SEVERAL PROGRAMS

Name and mode	Original version	Reordered version	Ratio of improvement
p58(-,-)	121	78	1.55
meal(-,-,-)	17626	1669	1.06
meal(+,+,-)	8011	7554	1.06
team(-,-)	34682	9990	3.47
team(-,+)	33292	8600	3.87
kmbench	161816	142559	1.14

VIII. CONCLUSIONS

We have presented a method for reordering Prolog for more efficient execution. We have shown what information is 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 presented a novel way of looking at modes, one better-suited to reordering than is the usual one. We have extended the Markov-chain method of heuristic reordering to cover full Prolog, and presented some results of our work.

There is scope for some additional work. Better techniques for recursive predicates would be useful, as well as an adaptation of some method of mode inference to deriving all legal modes of a program. The reordering system should also estimate nearly all probabilities and costs on its own, relying upon the programmer only as a last resort. Unfolding of goals [24] (replacing them with the goals of the clauses of the predicates they call) might greatly increase the possibilities for reordering, especially when clauses of a program are short and have many side-effects. 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] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufmann, 1981.
[2] M. Bruynooghe *et al.*, "Abstract interpretation: Towards the global optimization of Prolog programs," in *Proc. 1987 Symp. Logic Programming*, Sept. 1987, pp. 192–204.
[3] M. Bruynooghe and L. M. Pereira, "Deduction revision by intelligent backtracking," in *Implementations of Prolog*, J. A. Campbell, Ed. Chichester, England: Ellis Horwood, 1984, pp. 194–215.
[4] M. Bruynooghe and G. Janssens, "An instance of abstract interpretation integrating type and mode inferencing," in *Proc. 5th Int. Conf. Symp. Logic Programming*, Aug. 1988, pp. 669–683.

E II
FAMILY-TREE PROGRAM

mber of calls Markov	optimal	ratio of improvement
985		1.47
1409		43.91
2015		1.01
67218		1.39
792	792	1.00
539	530	1.45
1004	1004	1.00
21219	16603	0.75
1477		42.65
2054		52.41
2303		29.46
151062		0.91
200	200	1.15
367	367	20.66
549	548	1.00
18705	18705	1.52

```
).

(Y,Z), unequal(X,Y).

(Y).
Y).
her(Z,Y).
Y,W), siblings(W,Z).
Y,W), siblings(W,V), married(V,Z).
).
W), wife(W,Y).
```

he family-tree program.

o distinct versions of a pred-
and tests.
ve for the half-instantiated
istantiation is exploited by a
ge search trees are pruned
so readily for mode (−,−):
tiated, and so must be called
Similarly, for mode (+,+),
instantiated that goal order
rdered clauses are more ex-

rules from a corporate da-
written in Prolog; Table III
se results and those in Table
-Prolog 2.3 [10].) The facts
on the employee identifica-
antiated, many goals of the
ering essentially becomes a
is quickly and inexpensively
rset of these numbers whose

several other programs; most
es. p58 is Problem 58 from
" [7]: meal plans meals and
s. kmbench is a substantial
running a set of benchmark
lause of p58, meal, and
only four clauses of team
performance are less impres-
rams.

W
more
tion
of thi
still
a nov
reord

[5] J.-H. Chang, A. M. Despain, and D. DeGroot, "AND-parallelism of logic programs based on a static data dependency analysis," in *Digest of Papers of COMPCON Spring '85*, Feb. 1985, pp. 218-225.
[6] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. Berlin: Springer-Verlag, 1984.
[7] H. Coelho, J. C. Cotta, and L. M. Pereira, "How to solve it in Prolog," Laboratorio Nacional de Engenhara Civil, Lisbon, Portugal, July 1982.
[8] S. K. Debray, "Optimizing almost-tail-recursive Prolog programs," Dep. Comput. Sci., State Univ. New York at Stony Brook, Stony Brook, New York, Tech. Rep. 84/089, Nov. 1984.
[9] S. D. Debray and D. S. Warren, "Automatic mode inference for Prolog programs," in *Proc. 1986 Symp. Logic Programming*, Sept. 1986, pp. 78-88.
[10] S. D. Debray, "The SB-Prolog system, Version 2.3.2: A user manual," Dep. Comput. Sci., Univ. Arizona, Tucson, Tech. Rep. 87-15, Mar. 1988.
[11] M. M. Gooley and B. W. Wah, "Efficient reordering of Prolog programs," in *Proc. 4th Int. Conf. Data Eng.*, Feb. 1988, pp. 110-117.
[12] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*. New York: Van Nostrand, 1965.
[13] R. Kowalski, *Logic for Problem Solving*. New York: North-Holland, 1979.
[14] G.-J. Li and B. W. Wah, "How good are parallel and ordered depth-first searches?" in *Proc. Int. Conf. Parallel Processing*, Aug. 1986, pp. 992-999.
[15] H. Mannila and E. Ukkonen, "Flow analysis of Prolog programs," in *Proc. 1987 Symp. Logic Programming*, Sept. 1987, pp. 205-214.
[16] C. S. Mellish, "Some global optimizations for a Prolog compiler," *J. Logic Programming*, vol. 1, pp. 43-66, 1985.
[17] —, "Abstract interpretation of logic programs," in *Proc. 3rd Int. Conf. Logic Programming*, July 1986, pp. 463-474.
[18] A. Mycroft and R. A. O'Keefe, "A polymorphic type system for Prolog," *Artificial Intell.*, vol. 23, pp. 295-307, 1985.
[19] F. Pereira, C-Prolog User's Manual, Version 1.5, Edinburgh Computer Aided Architectural Design, Edinburgh, Scotland, Aug. 1983.
[20] L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren, User's Guide to DECsystem-10 Prolog, Sept. 1978.
[21] D. E. Smith and M. R. Genesereth, "Ordering conjunctive queries," *Artificial Intell.*, vol. 26, pp. 171-215, 1985.
[22] Z. Somogyi, "A system of precise modes for logic programs," in *Proc. 4th Int. Conf. Logic Programming*, May 1987, pp. 769-787.
[23] L. Sterling and E. Y. Shapiro, *The Art of Prolog*. Cambridge, MA: M.I.T. Press, 1986.
[24] H. Tamaki and T. Sato, "Unfold/fold transformation of logic programs," in *Proc. 2nd Int. Conf. Logic Programming*, 1984, pp. 127-138.
[25] D. H. D. Warren, "Efficient processing of interactive relational database queries expressed in logic," presented at the 7th Conf. on Very Large Data Bases, 1981.
[26] —, "An abstract Prolog instruction set," SRI Int., Menlo Park, CA, Tech. Note 309, Oct. 1983.
[27] —, "Applied logic—Its use and implementation as a programming tool," SRI Int., Menlo Park, CA, Tech. Note 290, June 1983.
[28] R. Warren, M. Hermenegildo, and S. K. Debray, "On the practicality of global flow analysis of logic programs," in *Proc. 5th Int. Conf. Symp. Logic Programming*, Aug. 1988, pp. 684-699.
[29] D. A. Wolfram, "Intractable unifiability problems and backtracking," in *Proc. 3rd Int. Conf. Logic Programming*, July 1986, pp. 107-121.

Markian M. Gooley (S'84) received the B.S. degree in mathematics *summa cum laude* from Millikin University, Decatur, IL, in 1981, and the M.S. degree in computer science from the University of Illinois at Urbana-Champaign in 1985.

At present he is working towards the Ph.D. degree in computer science at the University of Illinois. His research interests include search algorithms, logic programming, and computer design.

Benjamin W. Wah (S'74-M'79-SM'85) received the Ph.D. degree in computer science from the University of California, Berkeley, in 1979.

He had been on the faculty of the School of Electrical Engineering at Purdue University, West Lafayette, IN, between 1979 and 1985. He is currently a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. Between 1988 and 1989, he served as a Program Director of the Microelectronic Systems Architecture Program, National Science Foundation. He has published extensively in the areas of computer architecture, parallel processing, artificial intelligence, distributed databases, and computer networks. For his contributions to research, he has been selected as a University Scholar of the University of Illinois in 1989.

Dr. Wah is the Associate Editor-in-Chief of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, an Area Editor of the *Journal of Parallel and Distributed Computing*, and an Editor of *Information Sciences*. He serves as a member of the Governing Board of the IEEE Computer Society and a Program Evaluator for ABET (computer engineering) and CSAC (computer science). Previously, he served as an Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and a Distinguished Visitor of the IEEE Computer Society.