# Efficient Reordering of C-PROLOG

**Jian Wang, Jungsoon Yoo, and Tom Cheatham**
Department of Computer Science
Middle Tennessee State University
cswanjin, csyoojp, cheatham@mtsu.edu

**Abstract**

*PROLOG uses a depth-first search of an AND/OR graph to satisfy queries against its database. It searches sequentially through the clauses of a predicate whose head matches the query, trying to satisfy the goals in the clause body in a sequential left-to-right order. The ordering of clauses and goals is a major factor in the efficiency of a PROLOG program. We have developed a profiler for C-PROLOG that collects statistics including the failure rate of clauses and goals in a C-PROLOG program. These statistics are used by any of several re-ordering predicates capable of local or global re-ordering. The intent is to construct a reordered PROLOG program that outputs an equivalent set of answers, and is more efficient. Test results are promising.*

## 1 Introduction

The graceful non-procedural style of PROLOG has attracted many crusaders since its introduction by Robert Cohmerauer to develop a French-to-English translation system [2]. Its strongest support has come from the AI community, especially in Europe, and from the Japanese Fifth Generation Computer Thrust. PROLOG programmers are well acquainted with the inherent inefficiency in PROLOG's depth-first search of an AND/OR graph to satisfy a pending query. To see how the search works, we provide a simple example which will serve a dual purpose.

```
healthy(X) :- eats_right(X),
              sleeps_enough(X),
              exercises(X).
healthy(X) :- young(X).
eats_right(X) :- eats_fruit(X),
                 eats_grain(X),
                 eats_meat(X),
                 eats_dairy(X).
eats_fruit(tom).
eats_fruit(jungsoon).
eats_fruit(jian).

eats_dairy(tom).
eats_dairy(jungsoon).

eats_grain(tom).
eats_grain(jungsoon).

sleeps_enough(jian).
sleeps_enough(jungsoon).

eats_meat(tom).
eats_meat(jungsoon).
eats_meat(jian).

exercises(jungsoon).
young(jian).
```

To settle a query such as

?- healthy(tom).

PROLOG matches the query against the head of a rule or fact in the database, in this case instantiating X to "tom" in the first clause. Next, PROLOG tries the first goal from the right side, namely, "eats_right(tom)." To refute this goal, PROLOG matches with the head of the third clause in the database instantiating X to "tom" and, then tries the first goal from the right side, namely "eats_fruit(tom)" which succeeds and leads to the next goal from "eats_right(tom)," namely, "eats_grain(tom)," and so on. Eventually, the query "healthy(tom)" will fail since, for instance, "sleeps_enough(tom)" can not be satisfied. Thus, the conjunction of the goals within a clause forms an AND node, and the disjunction of the clauses in a predicate forms an OR node in the AND/OR graph for the query "healthy(tom)."

Experienced PROLOG programmers can write more efficient PROLOG programs, but it is often at the expense of clarity and time. It is generally agreed that "efficiency" is not a first-order software engineering principle. Rather, efficiency should be considered after a working system has been developed based on principles that provide for understandability and maintainability. The tools and techniques described in this article make this approach feasible in C-PROLOG programming. A new "profile" predicate is added to the C-PROLOG interpreter that allows the programmer to calculate selectively the success and failure rates for the clauses of a predicate and the goals of a clause. Once statistics have been collected for a reasonable time, the programmer can, after viewing the statistics, select one of the new reordering predicates to reorder part or all of the program.

The example above can be used to see how clause and goal order affect efficiency. Assume that more people "eat right" than "sleep enough" and that more people "sleep enough" than "exercise." Further, assume that most people are not "young" any more. Using these assumptions, the first clause in the "healthy" predicate, "eats_right," will succeed often before the clause "healthy" eventually fails, due to one of the other two goals. The effort it took to succeed the "eats_right" goal is then wasted. It would be better to order the goals by decreasing probability of failure, namely,

```
healthy(x) :-
    exercises(X), sleeps_enough(X), eats_right(X).
```

When we reorder goals in a clause body, we move the goals that are more likely to fail to the front of the list, preventing time spent on intermediate successes which eventually lead to failure.

One is inclined to reverse the order of the two "healthy" clauses since it appears that the second has a lower "cost" to evaluate. However, since we assumed that "most of us are not young any more," the second clause would fail often and probably should remain in the original order. When we reorder clauses in a predicate, we will move the clauses that are most likely to succeed to the top of the clause list for the predicate, thereby avoiding extra work.

## 2 Related Work and Assumptions

There have been numerous attempts to improve the efficiency of PROLOG. The "cut" was added to prevent undesirable backtracking. PROLOG compilers were developed to reduce translation time. A parallel unification machine has been suggested by Sibai, Watson, and Lu [5] to reduce the time spent in unification. Clause indexing has been suggested by Warren [7] to improve the efficiency of clause selection.

Recently, Gooley and Wah [3], using a Markov-chain to model the execution of a PROLOG query, have suggested a heuristic method for reordering clauses and goals within a PROLOG program to improve efficiency. Their method puts the clauses that are "more likely to succeed" and "inexpensive to evaluate" near the beginning of a predicate and the goals that are "more likely to fail" and "inexpensive to evaluate" near the beginning of the clause body. Probabilities for success and failure as well as costs must be entered by the programmer, at least for base clauses. Further, the calculations required for their methods will be expensive if implemented in a real system.

We have implemented a reordering mechanism in C-PROLOG. Our method suffers from a number of weaknesses which will be addressed in a future version. Any practical method is likely to have weaknesses. Currently, we assume the "cost" of evaluating a goal is a constant; that is, every goal costs the same to evaluate. Our C-PROLOG profiler collects the number of goals, in the AND/OR graph for a predicate p, that are called and that fail while profiling the predicate p. The average of these measures over the number of calls to predicate p could be viewed as a limited measure of the "cost" of evaluating the predicate p. In the current version, cost is not used as a factor in reordering. We base our decision to reorder solely on the probability of success and failure, collected during profiling, of the requisite clauses and goals. The programmer is not required to assign costs or probabilities, not even for base clauses. He/she must decide when to reorder based on the statistics collected by the profiler and whether further reordering is required.

Some predicates can not be reordered. For instance, in the clause (a1) of the predicate a no other order for the goals is acceptable:

```
(a1)    a :- write('You are'), b, write('welcome.').
(a2)    a :- write('not welcome.').
```

Built-in predicates that perform I/O, like "write," or alter the PROLOG database, like "asserta," can not be reordered when used as goals. If such a predicate appears as a goal in the body of a clause, as with "write" in the clause (a1) above, then clause (a1) is fixed in its predicate.

The "cut," denoted "!," restricts reordering. When the cut appears as a goal in a clause body, goals can not be reordered from one side of the cut to the other. However, we do allow reordering on either side of the cut. We allow clauses containing the cut to be reordered within their predicate. We do not address "implication" or "disjunction." We do not consider the effect of "modes" on reordering as discussed in Gooley and Wah [3].

## 3 New C-PROLOG Predicates

C-PROLOG which can be licensed from the University of Edinburgh includes the source code written in the C-language. It consists of nearly 10,000 lines of mostly uncommented code. Understanding the structure is a challenge. Ten predicates have been added to the C-PROLOG interpreter: profile/1, profile/2, reorderc/1, reorderg/1, reorderallc/0, reorderallg/0, setfixity/0, listto/1, listprog/0, and performance/0.

Table 1: Profile of Health Program

| Clause | Calls | Success | Failure | Failure % |
|---|---|---|---|---|
| healthy(VAR) :- eats_right(VAR), sleeps_enough(VAR), exercises(VAR). | 30 | 10 | 20 | 66.67 |
| healthy(VAR) :- young(VAR). | 20 | 5 | 15 | 75.00 |
| eats_right(VAR):- eats_fruit(VAR), eats_grain(VAR), eats_meat(VAR), eats_dairy(VAR). | 30 | 25 | 5 | 16.67 |

The last three predicates allow the programmer to inspect the results of the profiling and the reordering. "Listto(filename)" will redirect C-PROLOG's output to the specified file providing a "dribble" or "audit log" of the session. "Listprog" displays each clause in the PROLOG database and its corresponding profiling statistics, such as the number of times it was instantiated and the number of times it failed. In the output from "listprog," all variable names are replaced by "VAR." If one wants a conventional listing of the database, the standard built-in predicate "listing" can be used. Invoking "performance" reports a table, as shown in a later section, summarizing the performance results gathered during testing. For comparison, "performance" should be printed before reordering and after.

Before reordering can be done, statistics must be collected, using the "profile" predicate, to drive the reordering heuristic. A typical request to profile the "healthy/1" predicate in the example above would be:

```
profile(healthy(_), on).
```

Statistics will be collected for the "healthy" predicate and all goals and clauses in its AND/OR graph until profiling is turned off by:

```
profile(healthy(_), off).
```

The statistics are cumulative, over the session, that is, over the various calls to "profile." For each clause in the AND/OR graph of a predicate p being profiled, the system accumulates the number of calls to the clause and the number of times it fails. The probability of failure of a goal in a clause is calculated as the product of the probability of failure of the clauses in the corresponding predicate. Assume there is a predicate

```
happy(X) :- healthy(X), has_friends(X).
```

that contains the "healthy" predicate described above as a goal. The probability of the failure of the "healthy(X)" goal is estimated by the product of the failure probabilities of the two clauses in the "healthy" predicate, as collected by the profiler. It is obvious that the failure rates of the two clauses in the "healthy" predicate are not independent. Due to PROLOG's search order, the second clause will only be tried if the first fails. The failure rate calculated by our profiler, the number of failures divided by the number of instantiations, is not an independent probability. For convenience, we use it as an estimate of the independent probabilities. A sample of the output from the profiler for the "health" program is given in Table 1. The data are the result of 30 queries to the "healthy/1" predicate and serves only to demonstrate the form of the profiler's output.

The programmer can inspect the statistics collected to date by the profiler and determine "if" and "when" to reorder. Considerable latitude is provided in the reordering process. Both clauses and goals (within clauses) can be reordered, separately or in conjunction, either locally or globally. We recommend separate local reordering of goals and clauses, goals first with "reorderg/1" and then clauses with "reorderc/1." Then, when problems arise as a result of the reordering, as they will, it will be easier to identify the error(s). It is, however, possible to reorder globally across all clauses in the program with "reorderallc/0" and across all goals within the program with "reorderallg/0." Before reordering, the programmer should run "setfixity/0" to turn on the "fixed flag" for all built-in predicates that should not be reordered.

## 4 Performance Evaluation

Does reordering goals and/or clauses in a C-PROLOG program improve its efficiency? Not always, with our current version. Consider, for example, a predicate "mux/1" short for "mutual exclusion":

```
(m1)    mux(X) :- X > 2.
(m2)    mux(X) :- X ≤ 2.
```

which is tested with random values of X in the range 0 - 9 inclusive. Theoretically, (m1) will succeed 70% of the time it is called and (m2) will succeed exactly the other 30% of the time. The two clauses of "mux" are already in the best order! However, testing "mux" with

Table 2: Performance of Clause Reordering

| Predicate | After Call | After Fail | Before Call | Before Fail | Reduced Call % | Reduced Fail % |
|---|---|---|---|---|---|---|
| Family Program | | | | | | |
| brother/2 | 1976 | 1568 | 2388 | 2036 | 17.25 | 22.98 |
| father/2 | 7386 | 6968 | 7763 | 7345 | 4.86 | 5.13 |
| husband/2 | 10439 | 9667 | 11366 | 10594 | 8.16 | 8.47 |
| mother/2 | 8418 | 7958 | 8730 | 8270 | 3.57 | 3.77 |
| sister/2 | 996 | 636 | 1852 | 1492 | 46.22 | 57.37 |
| wife/2 | 9722 | 8952 | 10119 | 9349 | 3.92 | 4.25 |
| | | | | | | |
| Parts Inventory | | | | | | |
| partsof/2 | 1937 | 1118 | 2284 | 1465 | 15.19 | 23.69 |
| partsoflist/2 | 3840 | 2120 | 4276 | 2556 | 10.20 | 17.06 |
| | | | | | | |
| Fibonacci Number | | | | | | |
| fib/3 | 940 | 24 | 1220 | 352 | 22.95 | 93.18 |

Table 3: Performance of Goal Reordering

| Predicate | After Call | After Fail | Before Call | Before Fail | Reduced Call % | Reduced Fail % |
|---|---|---|---|---|---|---|
| brother/2 | 2388 | 2036 | 2388 | 2036 | 0.00 | 0.00 |
| father/2 | 1265 | 1084 | 7763 | 7345 | 83.70 | 85.73 |
| husband/2 | 1540 | 1344 | 11366 | 10594 | 86.45 | 89.30 |
| mother/2 | 1074 | 902 | 8730 | 8270 | 87.70 | 89.09 |
| sister/2 | 1852 | 1492 | 1852 | 1492 | 0.00 | 0.00 |
| wife/2 | 999 | 799 | 10119 | 9349 | 90.13 | 91.45 |

Table 4: Performance of Clause and Goal Reordering

| Predicate | After Call | After Fail | Before Call | Before Fail | Reduced Call % | Reduced Fail % |
|---|---|---|---|---|---|---|
| brother/2 | 1976 | 1568 | 2388 | 2036 | 17.25 | 22.99 |
| father/2 | 678 | 518 | 7763 | 7345 | 91.27 | 92.95 |
| husband/2 | 613 | 417 | 11366 | 10594 | 94.61 | 96.06 |
| mother/2 | 618 | 458 | 8730 | 8270 | 92.92 | 94.46 |
| sister/2 | 996 | 636 | 1852 | 1492 | 46.22 | 57.37 |
| wife/2 | 602 | 402 | 10119 | 9349 | 94.05 | 95.70 |

100 random queries, with profiling turned on, (m1) will succeed 70 times out of the 100 tries, and (m2) will succeed 30 times out of the 30 times it is called, on the average. Since the success ratio of (m2) is greater, the reordering predicate, reorderc, will reverse their order:

```
(m2)    mux(X) :- X ≤ 2.
(m1)    mux(X) :- X > 2.
```

which is clearly inferior. Repeated use of profiling and reordering, using the same test queries, will correct this error. In general, a second cycle of "profile and reorder" will correct a misordering of clauses if there is a significant difference in the "actual, independent" failure probabilities of the clauses. If the difference is not significant, a reordering error is not critical.

Reordering can change a working program into one that, for instance, does not terminate, especially where recursive predicates are involved. Inspecting the statistics gathered by profiling the reordered system will quickly pinpoint an infinite loop.

While reordering does not always work as one would hope, it will often yield a more efficient C-PROLOG program. Several factors affect the expected improvement from reordering. These factors include mobility, nondeterminism, dispersion of failure probabilities, and the size of the fact base [3], [6]. Predicates containing fewer fixed goals will benefit the most from reordering. The efficiency of a deterministic predicate will not be improved by our methods. If the failure rates of the various clauses in a predicate are nearly equal, little gain can be expected from reordering. Predicates with large fact bases generally show a greater gain from our "caching" clause reordering.

We present performance statistics based on three simple systems: a two-predicate recursive Fibonacci number calculator; a parts inventory program from Clocksin and Mellish [1]; and, a "pure PROLOG" "family" program containing twenty clauses divided among 11 predicates. The family database contains approximately 60 facts. The evaluation is divided into three parts, and a before-and-after performance is shown for clause reordering alone, for goal reordering alone, and for clause and goal reordering working in tandem. The results in Tables 2, 3, and 4 are based on 40 queries to each predicate in the family program; 40 queries to the two predicates from the parts inventory program; and 8 queries to the Fibonacci predicate. The same queries are used in the before-and-after tests. Only the family predicates appear in Tables 3 and 4 because goals of the other three predicates can not be reordered.

In Table 2, we can see that the gain from clause reordering is impressive for some predicates, such as f/3 which shows a 93% reduced failure rate, and sister/2 which shows a 50% decrease in both calls and failures, but not for all. Goal reordering in the family program, as indicated in Table 3, shows attractive improvements, over 80% reduction in calls and failures in four of the six predicates profiled and no improvement in the other two. When the reordering techniques are combined, as shown in Table 4, the best overall improvement is achieved – four of the six predicates show over 90% decrease in both calls and failures. It is possible, with the current system, to actually degrade performance.

Improvements in various predicates can vary greatly. In addition to the factors described above, programming style also greatly influences the amount of improvement expected from reordering. Overall, our approach has shown promise.

## 5 Conclusions and Future Work

We have implemented a feasible reordering system in the C-PROLOG interpreter, capable of reordering the clauses of a predicate and the goals within a clause. Clauses and goals can be reordered separately or together, for one predicate or for the entire program. Reordering is based on the probability of success or failure as estimated by the profiler from typical execution scenarios. Performance evaluation indicates a significant decrease, as much as 90%, in the number of calls and the number of failures for certain types of systems.

Our heuristic system for reordering clauses and goals, while not perfect, does provide a first step toward improving the efficiency in a C-PROLOG program. A second version of the system is being developed that provides a more theoretical foundation, including independent probabilities for the clauses of a predicate, and which bases the reordering not only on the probabilities, but also on the "cost." The new version will provide the programmer the ability to set the "fixed flag" for user-defined predicates, if necessary. Incorporating Gooley and Wah's [3] work on modes would be a next step.

## References

[1] Clocksin, W. F. and Mellish. C. S., *Programming in Prolog*, Springer-Verlag, New York, NY, 1984.

[2] Cohen Jacques, "A View of the Origins and Development of PROLOG," *Communications of ACM*, Volume 31, Number 1, 1988, pp. 26-36.

[3] Gooley, M. M. and Wah, B. W., "Efficient Reordering of PROLOG Programs," *IEEE Transactions on Knowledge and Data Engineering*, Volume 1, Number 4, 1989, pp. 470-482.

[4] Pereira, Fernando and Tweed, Christopher, *C-PROLOG User's Manual* Version 1.5 and 1.5+, SRI International, Menlo Park, CA, 1988.

[5] Sibai, F. N., Watson, K. L., and Lu, Mi, "A Parallel Unification Machine," *IEEE Micro*, Volume 10, Number 4, 1990, pp. 21-33.

[6] Wang, Jian, *Efficient Reordering in C-Prolog*, MS Thesis, Middle Tennessee State University, Murfreesboro, TN, 1992.

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