# Issues in Caching Prolog Goals

Barry Fagin
Computer Science Division
University of California, Berkeley

*ABSTRACT*

A technique for improving the performance of Prolog and Prolog implementations, goal caching, is proposed. The algorithmic modifications necessary to accommodate goal caching are outlined, some implementation issues are considered, and results of benchmarks executed by a goal caching Prolog interpreter are analyzed. The principal barriers to improved performance using goal caching are discussed, and future directions for research are suggested.

# TABLE OF CONTENTS

# INTRODUCTION

This paper is the result of an investigation into the feasibility and desirability of caching Prolog goals and their solutions. This investigation was undertaken with the aid of the University of New South Wales Prolog interpreter, which was modified to incorporate goal caching. We hoped that such caching would avoid redundant computations and yield a considerable performance improvement over the conventional Prolog execution algorithm. In the course of investigating goal caching, we also hoped that we would gain insight into hardware issues would be gained that would assist in resolving design questions in a Prolog machine currently under construction.

Our study indicates that the presence of a cache for redundant inferences necessitates nontrivial modifications to the Prolog execution algorithm. The extra hardware required, however, is relatively simple. The results of experiments with a Prolog interpreter that incorporates goal caching indicate that a cache for Prolog inferences can be implemented successfully on a Prolog machine. Further investigation indicates, however, that two principal issues must be resolved before implementing a hardware goal cache: the problem of side effects, and the problem of determining exactly how much overhead goal caching requires.

This report is divided into six chapters. Chapter 1 outlines the modifications to the Prolog execution algorithm dictated by goal caching. In chapter 2, some implementation issues are considered, and the execution of a simple program under the modified algorithm of chapter 1 is traced. Chapter 3 contains the results of some benchmark programs executed by the modified Prolog interpreter, while chapter 4 discusses what those results indicate.

Chapter 5 deals with some problems and unresolved issues in goal caching, and chapter 6 sums up the results of the investigation and points out future directions for research.

# CHAPTER 1

## Modifying the Prolog Execution Algorithm

This chapter will discuss what is meant conceptually by a goal cache, and will then discuss the LUSH algorithm Prolog uses to execute programs. The use of a goal cache will be shown to be incompatible with the conventional LUSH algorithm, and necessary modifications will be explained.

### 1. The LUSH Algorithm

A Prolog program is essentially a collection of statements indicating what conditions must hold for certain goals to be satisfied, or solved. These statements are called Horn clauses; they are members of a simple subclass of general logical statements. A horn clause consists of a head and a body, which may in turn consist of several single or compound terms. (A clause with no body is a unit clause). Each clause may be interpreted declaratively as expressing the natural language statement "The statement at the head of this clause is true if the statements in the body of this clause are true." Hence a unit clause has the declarative interpretation "This statement is true."

The execution of a program amounts to the attempt to satisfy a given goal. To solve a goal, Prolog searches for the first clause whose head unifies with the goal. This unification process is a simple pattern matching algorithm for finding the most general common instance of two terms possessing variables and constants. Once a match is found, the goals in the body of the matching clause are executed, from left to right. If Prolog cannot match a goal, it rejects the most recently activated clause and tries to find another clause whose head will match the previous goal. This process is called backtracking. Execution terminates when

there are no more goals waiting to be executed (success) or when all matches for the original goal have been rejected (failure).

This basic algorithm is known as the LUSH algorithm [1] , for Linear resolution with Unrestricted Selection for Horn clauses[2] ; the procedure in the UNSW Prolog interpreter that implements this algorithm is described by the following pseudocode:[3]

```
lush(GOAL)

PARENT: points to the parent of the goal being attempted
ENV: points to the top of the control stack
GOAL: the current goal being attempted
CLAUSE: the current clause in the database GOAL is being
 matched against

        NEW CLAUSE:
                PARENT = top of environment stack

        NEW_GOAL:
                IF no goal remains to be tried
                        GOTO SUCCEED
                ELSE
                        CLAUSE = first clause that can match GOAL

        BACKTRACK_POINT:
                .   /* restore various variable values;
                .       not relevant to this discussion   */
        ALTERNATIVE:
                IF no more alternatives remain that can match GOAL
                        GOTO FAIL
                ELSE
                        IF head of CLAUSE unifies with GOAL
                                increment ENV
                                save environment on control stack
                                GOAL = first goal in body of CLAUSE
                                GOTO NEW_CLAUSE
                        ELSE
                                CLAUSE = next clause in procedure
                                GOTO ALTERNATIVE
```

```
SUCCEED:
        IF PARENT > base of control stack
                restore parent environment
                GOAL = GOAL to right of goal that just
                 succeeded
                GOTO NEW_GOAL
        ELSE
                print variables

FAIL:
        IF ENV > base of control stack
                restore environment by popping control stack
                GOTO BACKTRACK_POINT
        ELSE
                return
```

It is useful to view the result of applying this algorithm on a Prolog program as a tree with alternating "AND" and "OR" levels. We shall refer to the tree produced by LUSH for a given program as the execution tree for that program. For example, the Prolog program of figure 1 has an execution tree shown in figure 2.

```
f(a).
g(a).
g(b).

a(X) :- b(X), c(X), d(X).
a(X) :- e(X).
a(X) :- f(X), g(X).

a(X)?
```

figure 1

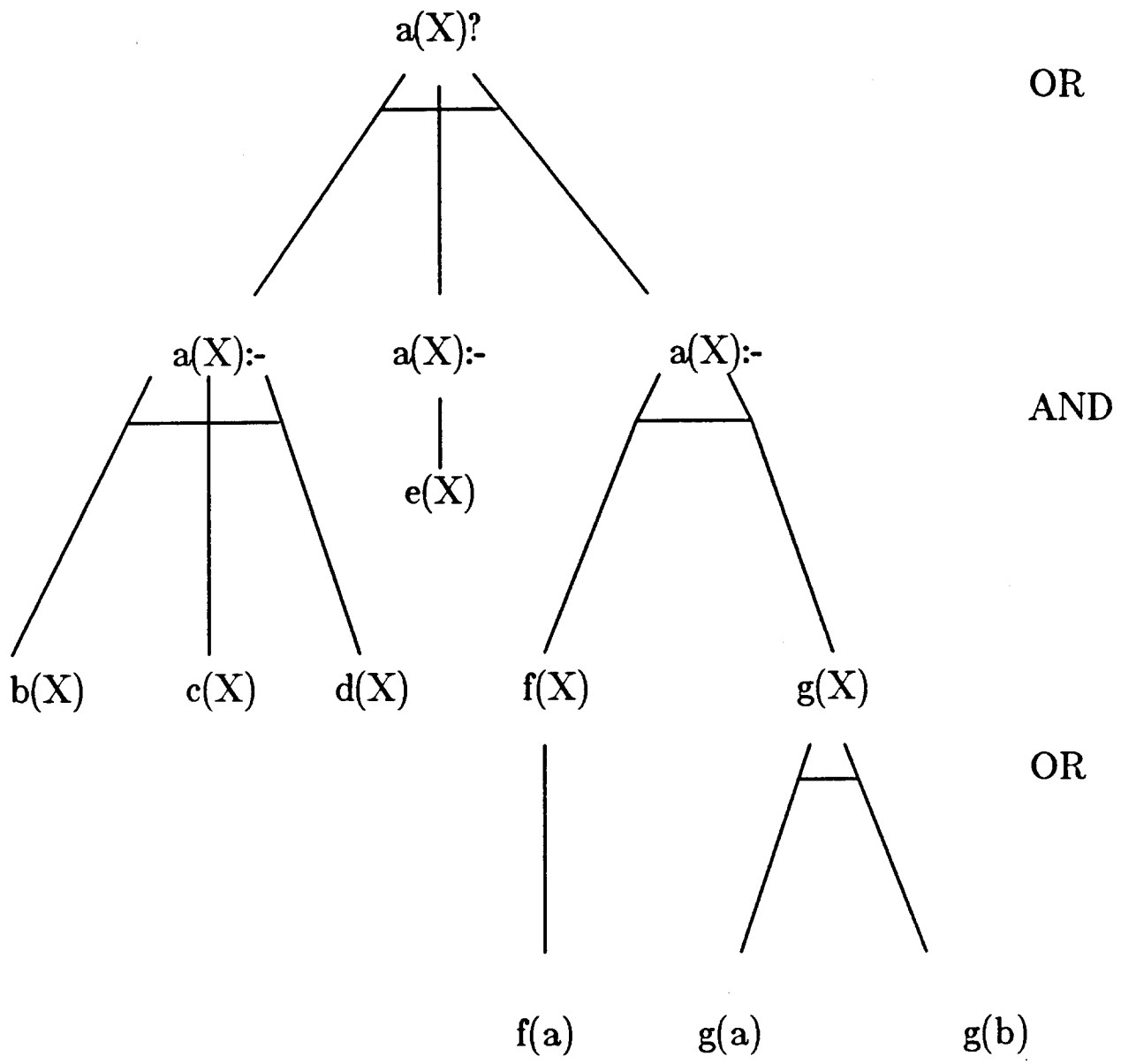


figure 2

## 2. Adding a Goal Cache to Avoid Redundant Computation

One apparent deficiency of the LUSH algorithm is that a great deal of effort may be expended in solving goals that have been solved during previous computation, or have been determined to have no solution. For example, consider the Prolog program in figure 3. Its

execution tree is shown in figure 4.

```
q(5,3).

s(1,4).
s(5,4).
s(8,5).

a(2,3).
a(3,4) :- q(X,3), s(Y,4).

b(3,5) :- a(X,Y), s(1,Y).

f(X,Y) :- a(X,Y), b(3,5).

f(X,Y)?
```

figure 3

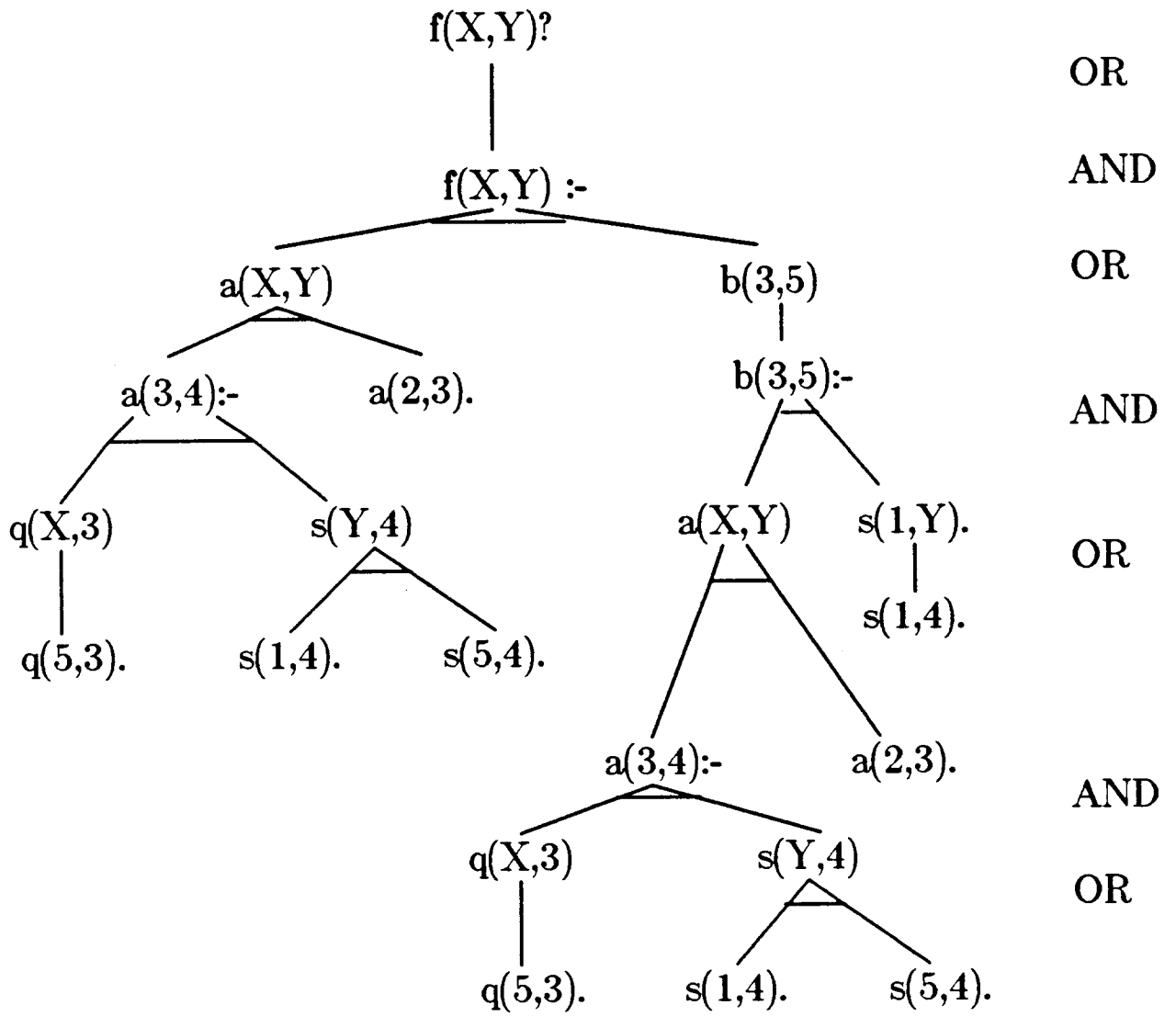


figure 4

First, Prolog attempts to solve the query "f(X,Y)". This unifies with the head of the only appropriate clause, and next Prolog tries to solve "a(X,Y)". This eventually leads to the solution X = 3, Y = 4. Prolog then attempts to solve "b(3,5)". This unifies with the head of the only clause for b, and then Prolog proceeds to solve "a(X,Y)" again. Notice, however, that one answer to this query has already been computed; we know already that one solution to this goal is X = 3, Y = 4. Hence all computation done to solve this goal again is really

unnecessary. We would like Prolog to be able to remember goals it has solved previously and what their solutions were, so that we can avoid redundant computation.

It is true that the amount of superfluous computation in this example is trivial; solving a(X,Y) again involves only a few extra unifications. Consider what would happen, however, if the program instead had procedures for q and s that were not simple unit clauses, but clauses that called other procedures that in turn called others. Such a program might have the execution tree shown in figure 5.

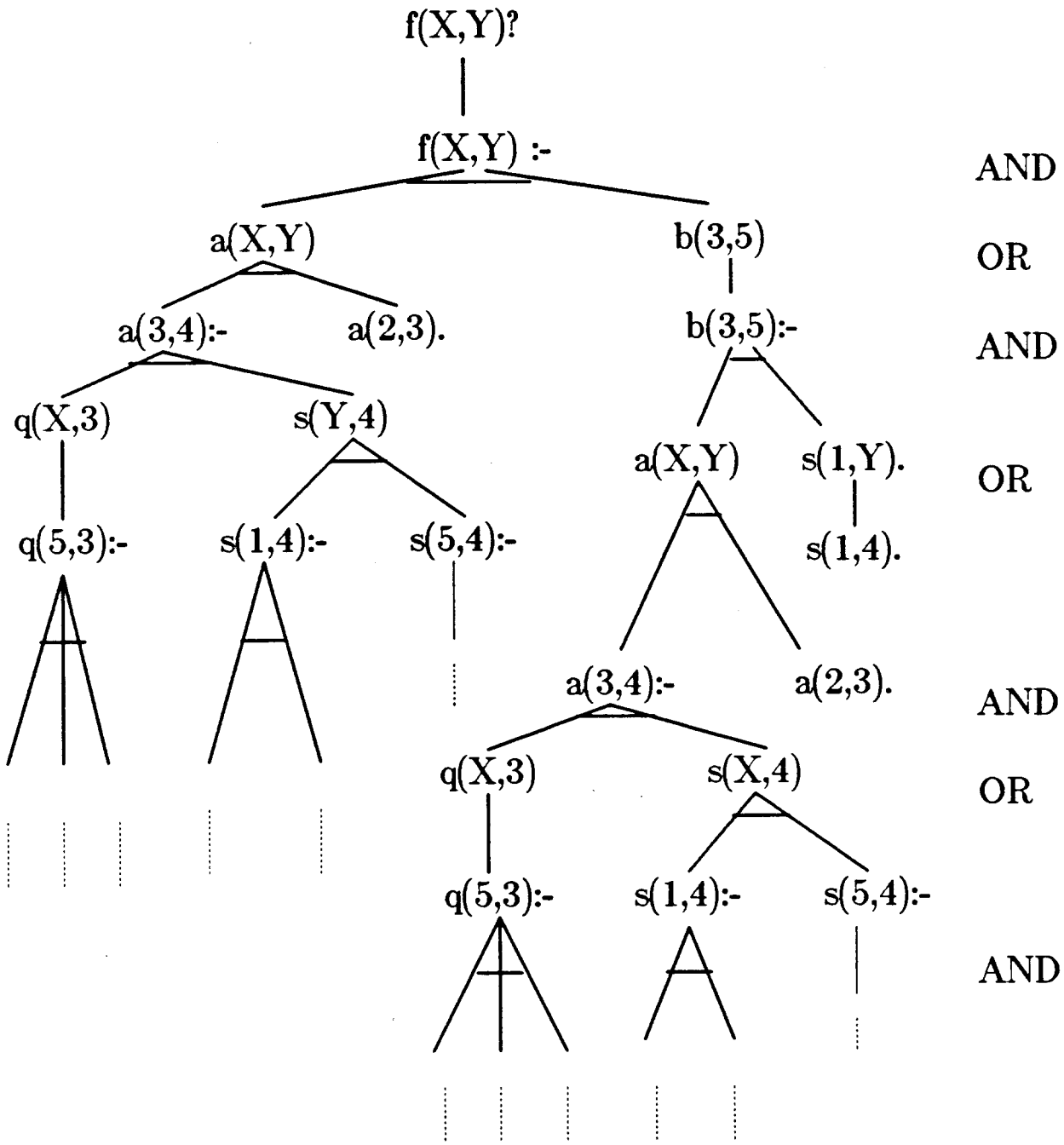


figure 5

If the depth of the subtrees with roots at q and s were significant, we would be performing a huge amount of redundant computation.

The contrived nature of this example may lead to an impression that redundant goal solution does not occur that often in real Prolog programs. In fact, measurements taken of several Prolog benchmarks do show redundant inferences; this will be discussed at the end of the report. For now, let us assume that redundant computation does in fact occur in Prolog programs, and that such computation could be avoided if Prolog had the ability to remember which goals had been solved previously and what their solutions were. To give Prolog this ability, the addition of a goal cache is proposed; a small memory that Prolog would access before attempting to unify with clauses in the program. We will now examine how the LUSH algorithm must be modified in the presence of such a cache.

## 3. How Goal Caching Affects the Prolog Execution Algorithm

There are five basic questions that must be dealt with when adding a goal cache to Prolog. They are:

1) When should answers be stored in the cache?
2) When should the cache be accessed?
3) How should goals with multiple solutions be handled?
4) What action(s) should be taken on a cache hit?
5) What action(s) should be taken on a cache miss?

We will address each one of these questions in turn.

### 3.1. When to Write to the Cache

The goal cache can be viewed as containing the accumulated "knowledge" of a Prolog program; it contains a set of facts. These facts may take the form of solved goals, in which case the solutions themselves will be in the cache, or they may be of the form of goals that are known to have no solution, in which case the cache simply records this information in some way and associates it with the goal. This indicates two different points in the LUSH algorithm at which we wish to write to the cache:

1) When a goal succeeds
2) When a goal fails

When a goal succeeds, we will store its solution in the cache. When a goal fails, we will (among other things), set a bit in the cache called the failure bit. This bit will be useful in determining that a goal has no more solutions.

Thus we may tentatively modify the LUSH algorithm in this manner:

```
.
.
.
SUCCEED:
        Store solution to GOAL in cache
        .
        .
        .
FAIL:
        IF GOAL actually failed (didn't fall through
          from SUCCEED)
                Set failure bit for the goal in cache
                .
                .
                .
```

### 3.2. When to Access the Cache

We wish to position the cache access at a point in the algorithm after the current goal has been determined that will guarantee that redundant computations will be avoided, but at the same time will ensure that the cache is accessed no more often than necessary. The first constraint indicates that the cache be accessed somewhere between NEW_CLAUSE and the point immediately before the unification operation, below ALTERNATIVE. It is not immediately apparent, however, what is indicated by the second constraint. Let us ignore, for the moment, what actions should be taken on a cache hit, and instead consider only cache misses. Suppose that we access the cache after ALTERNATIVE, get a miss, and attempt to unify with an appropriate clause in the program. Suppose that the unification fails.

According to LUSH, a new candidate clause is found and control proceeds back to ALTERNATIVE. As things stand now, the cache would be accessed again. But such an access would be pointless, since we know by virtue of the previous cache miss that no solutions to the current goal lie in the cache (we have not had a success or a failure to update the cache since then). Hence in order to avoid accessing the cache any more often than necessary, the cache access should be done before ALTERNATIVE. We have, then, three remaining possible points in the algorithm at which to insert a cache access: between NEW_CLAUSE and NEW_GOAL, NEW_GOAL and BACKTRACK_POINT, and BACKTRACK_POINT and ALTERNATIVE.

The first choice is not satisfactory. After a goal in the body of a clause succeeds the next goal to the right (if there is one) becomes the current goal and execution continues at NEW_GOAL. If the cache access were placed between NEW_CLAUSE and NEW_GOAL, we would fail to access the cache after such a transfer of control; the cache would be checked only when the current goal is the first goal in the body of a clause. Clearly this is not what is desired.

The second choice, between NEW_GOAL and BACKTRACK_POINT, is also unsatisfactory. Placing the cache access there would mean that we fail to check the cache when Prolog is attempting a retry, endeavoring to solve a goal again. When Prolog attempts to solve a goal again and another solution to it lies in the cache, we would like to avoid carrying out the redundant computations associated with finding that solution. Hence we wish to access the cache on retries.

We are left then with one remaining point for cache access: between BACKTRACK_POINT and ALTERNATIVE. Placing the cache access here will satisfy the twin constraints of avoiding as many redundant computations as possible while minimizing cache accesses. Hence we may further refine LUSH as follows:

```
.
.
.
BACKTRACK_POINT
        check cache for a solution to GOAL
        IF cache hit    /* goal solved previously */
                ?       /* take as yet unspecified action */
                ?
        ELSE            /* no (more) solutions in cache  */
                ?       /* take as yet unspecified action */
                ?
        .
        .
        .
```

### 3.3. Goals With Multiple Solutions

Given that goals can have multiple solutions, we must somehow develop a mechanism that enables Prolog to indicate which solution from the cache is desired. To do this, we associate a number with each goal instance. This number is maintained so that it is always equal to the number of times the goal has been solved. We can then store this number with cache solutions to distinguish multiple solutions to the same goal. When accessing the cache, we supply along with the goal to be solved the number indicating how many previous solutions to the goal have been found. We will call this number the cache counter, or cc for short. Since we desire the next solution from the cache, a cache access with a cc of zero indicates that the first solution from the cache is being sought.

The challenge, then, is to properly maintain the cc of each instance of a goal (the same goals can, and indeed most of the time will, have different cc's at different points in the program). Let us examine what must be done to LUSH in order to manage these cache counters.

Clearly, when a goal is first attempted it's cc must be set to zero; if any solutions to it lie in the cache we desire the first one. It is almost as obvious that the cc of a goal should be incremented by one after the goal has been solved, since the cc is defined to be the number of

solutions that have been found for the goal. It is less obvious, however, that the cc of a goal must be reset to 0 when it fails. This is because when a goal fails, the goal to its left (if it exists) will be attempted again. Should it succeed, the goal that previously failed will be attempted again, this time possibly with new bindings; hence when we try the goal again we desire the first solution from the cache, if a solution exists at all.

Finally, when we store to the cache, we must store the cc of the goal as well. Having the cc in the cache will prove essential to extracting a particular solution from the cache, as well as in determining that a goal has no more solutions. This will be shown in the next chapter.

Thus the LUSH algorithm may now be further modified as follows:

```
.
.
NEW_GOAL:
        GOAL.cc = 0
        .
        .
BACKTRACK_POINT:
        n = GOAL.cc
        Check cache for (n+1)st solution to goal t
        .
        .
        .
SUCCEED:
        store solution, GOAL.cc in cache
        GOAL.cc++
        .
        .
        .
FAIL:
        IF GOAL failed
                store GOAL.cc in cache
                set cache failure bit
                GOAL.cc = 0
        .
        .
        .
```

### 3.4. Actions to be Taken on a Cache Hit

A cache hit corresponds to one of two events: either we have found a new solution to the goal, or the goal is known to have no (more) solutions. In either case, the action is straightforward. If the cache has yielded a new solution then the current goal has succeeded; hence we should take whatever actions are taken upon success. If, however, the goal has no (more) solutions, we should take whatever actions are taken upon failure. We can test for the absence of solutions by examining the failure bit in the cache. Thus simple GOTO statements are all that are necessary to fill in some of the blanks in the previous LUSH algorithm, yielding:

```
.
.
.
BACKTRACK_POINT:
        n = GOAL.cc
        Check cache for (n+1)st solution to goal t
                IF cache hit
                        IF failure bit = 1
                                GOTO FAIL
                        ELSE
                                GOTO SUCCEED
                ELSE
                        ?
                        ?
        .
        .
        .
```

### 3.5. Actions to be Taken on a Cache Miss

A cache miss means that the desired solution to the goal was not in the cache, but may still lie in the program (i.e. the goal is not known to have no more solutions. We will refer to such a goal as being active). Hence we must continue execution in the program. But where? In our present version of LUSH, there is no way to know where to resume execution since we do not know which clause in the procedure will produce the newest solution, the next solution

that is not already in the cache. What is needed then is a pointer for each active goal that indicates the next clause to try when a goal triggers a cache miss. Such a pointer will be called a resume pointer as suggested by Dobry4, since it indicates where to resume execution in the program.

Let us assume the existence of some arbitrary data structure which enables us to uniquely associate a pointer to a clause with a given goal. We would like to maintain such a structure so that, on a cache miss, we could simply find the resume pointer associated with the current goal, and continue execution there. The main task then is to make sure that at any time the resume pointer is always correct. Let us trace through the LUSH algorithm and see at which points it is necessary to add to, delete from, and update the resume pointer data structure.

Clearly, when a goal is first tried we would like to create an entry for it in the resume pointer data structure (RPDS) and initialize it. Thus when the cache is accessed, if the goal generates a cache miss and the first solution was desired, we should create an entry in the RPDS for the goal, and set its resume pointer to the first clause in the appropriate procedure. When a goal fails we want to remove its entry from the RPDS, since all solutions to it lie in the cache, as would the knowledge that the goal had no solutions if such were the case. If we create and delete RPDS entries only at these two times then we guarantee that each active goal will have at least one and only one RPDS entry. We are left, then, to identify the points in the LUSH algorithm at which it is necessary to modify the RPDS entry.

We wish the RPDS entry for a goal to point to the clause to try that will yield the next solution on a cache miss. Consider what happens when the current goal successfully unifies with the candidate in the database. Then the clause at which to continue on a cache miss is the next clause in the database, since if any answers to the current goal will be generated following the unification they will be cached as well, and the next clause in the program is the

one that will yield the newest solution. Hence if the unification with the target clause in the program is successful, we must update the RPDS entry for the goal to point to the next clause.

Thus, to permit correct resumption of execution after a cache miss, we need a collection of pointers to clauses in the program, one for each active goal. To correctly maintain such a structure, we must modify the LUSH algorithm as follows:

```
BACKTRACK_POINT
        n = GOAL.cc
        Check cache for (n+1)st solution to goal t
        IF cache hit
                IF failure bit = 0
                        GOTO SUCCEED
                ELSE
                        GOTO FAIL
        ELSE
        IF GOAL.cc = 0
         (i.e. first time goal tried)
                create RPDS entry for goal
                RPDS entry of goal = CLAUSE
        ELSE
                CLAUSE = RPDS entry of goal
        .
        .
        .
ALTERNATIVE:
        .
        .
        .
        IF head of CLAUSE unifies with GOAL
                RPDS entry of GOAL = clause following
                  CLAUSE in program
                .
                .
                .
FAIL:
        IF GOAL actually failed
                store GOAL.cc in cache, set failure bit
                GOAL.cc = 0
                Remove RPDS entry of GOAL
        .
        .
        .
```

## 4. The LUSHC Algorithm

We are now finished with our modifications to the Prolog execution algorithm. What we have produced is different enough from the original to warrant a new name: the LUSHC algorithm. It is this algorithm, we believe, that a Prolog machine with a goal cache must use to execute programs; we sum up the results of this chapter with its complete description.

```
NEW_CLAUSE:
        PARENT = top of environment stack
NEW_GOAL:
        IF no goals remain to try
                GOTO SUCCEED
        ELSE
                CLAUSE = first clause that can match t
        GOAL.cc = 0
BACKTRACK_POINT:
        n = GOAL.cc
        check cache for (n+1)st solution to GOAL t
        IF cache hit
                IF failure bit = 1
                        GOTO FAIL
                ELSE
                        GOTO SUCCEED
        ELSE
                IF GOAL.cc = 0
                        create RPDS entry for goal
                        RPDS entry of goal = CLAUSE
                ELSE
                        CLAUSE = RPDS entry of goal
ALTERNATIVE:
        IF no more goals remain that can match GOAL
                GOTO FAIL
        ELSE
                IF head of CLAUSE unifies with GOAL
                        RPDS entry of goal = clause following
                          CLAUSE in program
                        increment ENV
                        save environment on control stack
                        GOAL = first goal in body of CLAUSE
                        GOTO NEW_CLAUSE
                ELSE
                        CLAUSE = next clause in procedure
                        GOTO ALTERNATIVE
SUCCEED:
        store solution to GOAL, GOAL.cc in cache
        increment GOAL.cc
        IF PARENT > base of control stack
                restore parent environment
                GOAL = goal to right of goal that succeeded
                GOTO NEW_GOAL
        ELSE
                print variables
FAIL:
        IF GOAL actually failed (didn't fall through from SUCCEED)
                store GOAL.cc in cache
                set failure bit for cache entry
                GOAL.cc = 0
```

```
        remove RPDS entry of goal
IF ENV > base of control stack
        restore environment by popping control stack
        GOTO BACKTRACK_POINT
ELSE
        return
```

# CHAPTER 2

# Goal Caching Hardware

We have seen how the presence of a goal cache requires substantial modifications to the Prolog execution algorithm. Having outlined these modifications, we proceed to the more difficult question of implementation: what mechanism should be used to access the cache and trigger the transmission of the correct solution? To answer this question, we first outline the similarity between conventional caches and the goal cache. We then show that the associative comparison used in comparing tag fields in conventional caches is complicated slightly by the problem of unbound variables in the goal. Fortunately we can solve this problem, so that goals with unbound variables can be associatively compared with one another. We conclude with a detailed analysis of the program of figure 3, executing it using the LUSHC algorithm and the cache mechanisms developed in this chapter.

## 5. Conventional Caches and the Goal Cache

Conventional caches are accessed by comparing associatively the tag field of the desired address with the tag field of a set of cache entries. A match indicates that the data associated with the tag is in the cache (a cache hit), causing the data to be gated out through the port of the cache memory. A goal cache for Prolog would work in a roughly similar fashion. The "address" supplied to the goal cache would be the goal Prolog was trying to solve, along with the cc of the goal. The tag fields in the cache would be representations of goals, while the data items in the cache would correspond to solutions to those particular goals. The tag fields would also include the cc stored there when the solution was written to the cache, to indicate which solution corresponds to each associated data item. In a cache

access cycle, the current goal and its cc would be compared to all tags in the goal cache. A match would indicate that the desired solution was in the cache, and the goal would then be unified with the associated data.

## 6. The Problem of Goal Comparison

How are the current goal and the cache tags to be compared? At first glance, a simple associative match on the characters of the goal might appear to be adequate. The problem with simple matching is that it will fail to detect a large number of previously solved goals, and hence will lead to unnecessary computations. This problem occurs because Prolog often attempts to solve goals that were equivalent to previously solved ones, but not identical. We say two goals are equivalent if they are isomorphic to one another; that is, if they differ only in the names of their variables. For example, the goals glork(A,B) and glork(X,Y) are equivalent, as are the goals foo(a,X,X) and foo(a,C,C).

Equivalent goals have identical solution sets; hence if a goal equivalent to the current one has already been solved then we may use its solutions as solutions of the current goal. An exact matching comparison algorithm, however, will not permit us this luxury. Suppose, for example, that all the answers to the query "glork(X,Y)" were in the cache, and that a Prolog program was currently attempting to solve "glork(A,B)". The answers to this query are identical to the answers to glork(X,Y), which already lie in the cache. Unfortunately, exact matching would yield the result that none of the answers to glork(A,B) are in the cache; only the answers to glork(X,Y). Clearly this is undesirable.

At first glance, it might appear that since only the variables cause difficulty in goal comparison, ignoring the variables in comparing goals will solve our problem. Unfortunately, this is not so. While the goals glork(A,B) and glork(Q,W) are equivalent, glork(A,B) and glork(C,C) are not: the set of all things that glork other things is not the same as the set of

all things that glork themselves. Thus the variables in a goal must be considered in some way when determining if goals are equivalent.

Fortunately, there is an easy way of comparing goals with one another. Wherever an unbound variable in a goal appears, we can replace it with a unique identifying number within the goal (this number is easily determined at compile time). We then use this number in the goal when accessing the cache, and simply compare the goals. For example, the goals glork(A,B) and glork(X,W) would both be reduced to glork(v1,v2) since A and X are the first variables in their respective goals, while B and W are the second. (Here, the "v" indicates that the number represents a variable. In an actual implementation, the bytes corresponding to the variables would probably be tagged to indicate their values represent unbound variables). Similarly glork(A,B) and glork(C,C) would be reduced to glork(v1, v2) and glork(v1, v1), respectively, and a simple comparison would show them to be nonequivalent. Glork(c, A, A, C) and glork(c, D, D, Y) would both be reduced to glork(c, v1, v1, v2); a comparison would show them equivalent.

## 7. An Example

We now illustrate how goal caching might work by tracing through the execution of the program of figure 3 using the LUSHC algorithm.

First, Prolog will attempt to solve the goal "f(X,Y)". Its cc will be set to zero, and the cache will be accessed with "address" f(v1,v2) 0. Since the cache is empty at this point, we will have a cache miss. The cc of f(X,Y) is zero, so we create an entry in the RPDS and set its value to be equal to clist, the first clause that can match f(X,Y). Thus the cache and the RPDS are as shown in figure 6. (In this and all the following figures, "fb" stands for failure bit). f(X,Y) unifies with the head clause of clist, so the RPDS entry for f(X,Y) is updated to the next clause in the procedure (in this case the nil clause), and we have figure 7.

CACHE

empty

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | f(X,Y) :- a(X,Y), b(3,5). |

figure 6

CACHE

empty

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |

figure 7

Prolog then proceeds to access the cache with goal "a(v1,v2) 0". This triggers a series of actions similar to those taken for f(X,Y). Once a(X,Y) unifies with a(3,4) and the RPDS is updated, we obtain figure 8, and after q(X,3) unifies with q(5,3) we obtain figure 9.

Now, however, a goal has succeeded. Following the LUSHC algorithm, we store the goal that succeeded, q(X,3), in the cache, along with its cc and the solution. This gives figure 10. We then increment the cc of q(X,3), and attempt to solve "s(Y,4)".

CACHE

empty

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |

figure 8

CACHE

empty

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |

figure 9

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |

figure 10

The cache is accessed with the "address" s(v1,4) 0; a goal equivalency check with the only tag in the cache fails, so we have a cache miss. This was the first time the goal had been tried, so after it unifies with s(1,4) we have figure 11.

Now s(1,4) has succeeded, so we update the cache to yield figure 12. We restore the parent environment, and find that the parent goal has succeeded as well. Thus we now store the parent goal, a(X,Y), in the cache. The cache and RPDS now look like figure 13.

Prolog then tries to solve b(3,5). This is a new goal, so after it unifies with b(3,5) in the database the cache and RPDS will be as shown in figure 14.

Now, however, the first redundant computation is attempted. The cache is accessed with the goal "a(v1,v2) 0". An associative comparison yields a unique cache hit, and the answer "a(3,4)" is obtained. Since the failure bit is not set, the goal has succeeded, and we have avoided the computation involved with solving "a(X,Y)" again.

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |

figure 11

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |

figure 12

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |

figure 13

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |
| b(3,5) | NIL |

figure 14

Next, Prolog attempts to solve "s(1,4)". This yields a cache miss, and a successful unification with s(1,4) updates the RPDS. The goal then succeeds, so we update the cache and increment the cc of s(1,4) to 1. The parent goal, b(3,5), has now succeeded as well, so we update the cache again. Notice that this goal has no associated data with its solution; its "answer" was simply "yes". We now have a cache and an RPDS as shown in figure 15.

The success of "b(3,5)" means that the original query, "f(X,Y)", has at last succeeded. We update the cache again, and print the solution "X = 3, Y = 4". Now, however, no more new goals remain to be tried, so we fall through to the FAIL portion of LUSHC. We restore the environment of the last goal that succeeded, "s(1,4)", and go to BACKTRACK_POINT.

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |
| b(3,5) | NIL |
| s(1,4) | s(5,4) |

**figure 15**

The contents of the cache at this point are shown in figure 16.

The cache is now accessed again, with the goal "s(1,4) 1". Notice the importance of the new cc value of the goal; it says that the second solution from the cache is desired. Thus the solution to this goal that is already in the cache will be rejected, because its tag is "s(1,4) 0". The query yields a cache miss, but as this was not the first time this goal was attempted, the next candidate clause is taken from the RPDS; thus Prolog will now try to unify "s(1,4)" with

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |
| f(v1,v2) 0 | 0 | 3, 4 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |
| b(3,5) | NIL |
| s(1,4) | s(5,4) |

figure 16

"s(5,4)", instead of "s(1,4)". This unification will, of course, fail, as will the unification with s(8,5). As there are now no more alternatives, the goal "s(1,4)" fails. According to LUSHC, we store the goal and its cc in the cache and set the failure bit, and remove its RPDS entry. This yields the configuration of figure 17. Notice that this cache entry also does not have any associated data, because none is necessary. Should we access the cache with the same tag

again, we will get a hit with the failure bit set, and hence we will know that no more solutions are to be found.

The failure of s(1,4) causes backtracking. Its cc is set to 0, and the goal to the left, a(X,Y), is asked again. Notice, however, that its cc was incremented to 1 when it succeeded previously. Hence the cache is accessed with the goal "a(v1,v2) 1", which misses. We then

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |
| f(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 1 | 1 | none |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | a(2,3). |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |
| b(3,5) | NIL |

figure 17

. Fa

continue at the clause indicated by the goal's entry in the RPDS: a(2,3). a(X,Y) unifies successfully with a(2,3) and succeeds, yielding a cache and RPDS with contents as shown in figure 18.

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |
| f(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 1 | 1 | none |
| a(v1,v2) 1 | 0 | 2, 3 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| a(X,Y) | NIL |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |
| b(3,5) | NIL |

figure 18

Next, s(1,3) is attempted. No way exists to solve this goal, so after all possibilities have been tried, the goal fails. The cache is then accessed with the goal "a(v1,v2) 2", yielding a cache miss. Recovering at the clause indicated by the RPDS entry, nil, we see that this goal now fails as well. Thus we store this fact in the cache, and remove the RPDS entry for the goal. We now have figure 19.

Now that a(X,Y) has failed, the parent goal b(3,5) has failed as well. We update the cache and the RPDS appropriately, reset the cc of b(3,5) to zero, and attempt to solve a(X,Y) again. The cache is now accessed with a(v1,v2) 1 as this instance of the goal has only been solved once. This produces a cache hit, and b(3,5) 0 is attempted. This also produces a hit, avoiding all work done in the previous solution of b(3,5). The original goal f(X,Y) has now succeeded again, so we update the cache and RPDS to give figure 20. We then print the solution "X = 2 Y = 3".

Next, b(3,5) is attempted again. The cache is accessed with b(3,5) 1. This yields a cache hit, but the failure bit is set, so the goal fails. The goal a(X,Y) is attempted again and fails, and finally f(X,Y) is attempted again and fails. No more alternatives remain, so LUSHC returns. At program termination, the contents of the cache and RPDS are as shown in figure 21.

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |
| f(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 1 | 1 | none |
| a(v1,v2) 1 | 0 | 2, 3 |
| s(1,3) 0 | 1 | none |
| a(v1,v2) 2 | 1 | none |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |
| b(3,5) | NIL |

figure 19

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |
| f(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 1 | 1 | none |
| a(v1,v2) 1 | 0 | 2, 3 |
| s(1,3) 0 | 1 | none |
| a(v1,v2) 2 | 1 | none |
| b(3,5) 1 | 1 | none |
| f(v1,v2) 1 | 0 | 2, 3 |

RPDS

| goal | clause to resume at |
|---|---|
| f(X,Y) | NIL |
| q(X,3) | NIL |
| s(Y,4) | s(5,4) |

figure 20

CACHE

| tag | fb | data |
|---|---|---|
| q(v1,3) 0 | 0 | 5 |
| s(v1,4) 0 | 0 | 1 |
| a(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 0 | 0 | none |
| b(3,5) 0 | 0 | none |
| f(v1,v2) 0 | 0 | 3, 4 |
| s(1,4) 1 | 1 | none |
| a(v1,v2) 1 | 0 | 2, 3 |
| s(1,3) 0 | 1 | none |
| a(v1,v2) 2 | 1 | none |
| b(3,5) 1 | 1 | none |
| f(v1,v2) 1 | 0 | 2, 3 |
| a(v1,v2) 2 | 1 | none |
| f(v1,v2) 1 | 1 | none |

RPDS

empty

figure 21

Having analyzed how a Prolog goal cache might be implemented, it is now appropriate to inquire what kind of performance improvement could be expected if all the features discussed up to this point were to be implemented in a Prolog machine. It is to this topic that we now turn.

# CHAPTER 3

## Experimental Results

A Prolog interpreter that implements the LUSHC algorithm, incorporating all the modifications discussed in the previous chapters, is now running under Berkeley 4.2 BSD UNIX. Several benchmark programs were executed using this interpreter, and statistical measurements made to predict the performance improvement. We believe that, since the basic operation in Prolog is unification, counting the unifications made in accessing the database in a goal caching implementation and comparing with the number of unifications in a conventional implementation yields an accurate measure of the speedup obtained by goal caching. Caveats that follow from this assumption as well as others will be outlined after the results are presented.

### 8. The Benchmarks

A total of twelve benchmarks programs were analyzed:

A Symbolic Differentiatior (diff)

One of the benchmarks from Warren's thesis;[5] computes various derivatives symbolically. Run to compute various tenth power derivatives, and to differentiate $(x+1)*(x^{-2}+2)*(x^{-3}+3)$ with respect to x.

The Towers of Hanoi (hanoi)

Taken from Clocksin and Mellish;[6] solves the towers of hanoi problem. Run with two, four, and eight disks.

The Missionaries and Cannibals Problem (mis.can)

Written by Claude Sammut. Solves the problem of transporting three missionaries and three cannibals across a river in one boat.

A Mobeius Counter Simulation (mob)

Written by Jung-Herng Chang. Simulates a mobeius counter, using JK flip flops. Run to produce simulations of length zero through four.

A "Mu Math" Theorem Prover (mu.math)

Written by Al Despain. Generates theorem of the "mu math" system [7] Run to generate all theorems with derivations of length 2, length 3.

Two Circuit Design Programs (nand.design1, nand.design2)

Written by Jung-Herng Chang. Designs optimal NAND gate circuits for a given functional specification. Nand.design2 does the design more "intelligently" by examining fewer superfluous cases. Run to produce two-input circuits with one, two levels.

Quicksort (qsort)

Another benchmark of Warren's; the quicksort algorithm. Run to produce a sorted list of fifty numbers.

The Queens Problem (queens)

Written by Rowland Sammut. Solves the problem of arranging n queens on an n by n chessboard so that no two queens threaten each other. Run with n = 3.

Database Query (query)

A third benchmark from Warren; searches a database to find countries of similar population density.

The Sieve of Eratosthenes (sieve)

Taken from Clocksin and Mellish. Uses Eratosthenes' algorithm to compute all primes below a given integer n. Run with n = 16.

List Reversal (reverse)

The last benchmark from Warren; run to reverse a thirty-element list.

## 9. The Results

The results of running the benchmarks under the interpreter are summed up in the following table. The predicted speedup S and the hit ratio $\Phi$ are shown for various queries that the programs were asked to solve. U is the number of unifications, N is the number of solutions cached rounded up to the nearest power of two (hence N represents the minimum size cache required), and C is the number of clauses in the benchmark program.

| Benchmark Results | | | | | | |
|---|---|---|---|---|---|---|
| program | query | S | Φ | U | N | C |
| diff | divide10 | 2.3 | 26% | 230 | 16 | 10 |
| | log10 | 1.01 | 5% | 375 | 32 | |
| | ops8 | 1.18 | 32% | 386 | 16 | |
| | times10 | 2.20 | 24% | 224 | 16 | |
| hanoi | 2 disks | 1.18 | 8% | 78 | 16 | 3 |
| | 4 disks | 2.73 | 23% | 174 | 32 | |
| | 8 disks | 23.33 | 29% | 366 | 128 | |
| mis.can | solve the problem | 1.16 | 20% | 1935 | 256 | 18 |
| mob | level 0 | 1.00 | 0% | 11 | 2 | 9 |
| | level 1 | 1.09 | 8% | 77 | 16 | |
| | level 2 | 1.31 | 14% | 127 | 16 | |
| | level 3 | 1.49 | 19% | 174 | 32 | |
| | level 4 | 1.82 | 24% | 205 | 32 | |
| mu_math | level 2 | 1.58 | 4% | 731 | 128 | 17 |
| | level 3 | 2.21 | 5% | 1594 | 256 | |
| nand.design1 | level 1 | 1.74 | 22% | 548 | 64 | 14 |
| | level 2 | 18.08 | 47% | 7845 | 512 | |
| nand.design2 | level 1 | 1.55 | 26% | 635 | 64 | 13 |
| | level 2 | 3.45 | 51% | 11214 | 1024 | |
| qsort | 50 numbers | 1.01 | 1% | 3776 | 528 | 6 |
| queens | 3 x 3 board | 1.23 | 12% | 765 | 128 | 17 |
| query | search database | 11.38 | 1% | 4636 | 32 | 52 |
| - reverse | 30 numbers | 1.00 | 0% | 3097 | 512 | 4 |
| sieve | primes below 16 | 1.00 | 0% | 715 | 0 | 8 |

# CHAPTER 4

## Some Comments on the Results

One of the first things one notices after glancing through the table of results is that there is virtually no correlation between the expected speedup and the hit ratio. This shows that what is important to speedup in a goal caching scheme is not how often cache hits occur, but where they occur in the execution tree. Programs with high hit ratios may have the bulk of their redundant goals occurring at or near the leaves and hence little work is avoided by keeping their solutions in a cache. Conversely, a program with only a few hits may have redundant computations near the root, in which case large amounts of computation can be avoided.

The wide range of speedups is also striking. Some problems, in particular the numerically oriented ones, exhibit little or no potential speedup. Others, however, can be sped up by factors ranging from a little less than two to about twenty-three. These results suggest that programs that search for solutions over a problem space are far more likely to have their execution time improved by goal caching than numerically oriented programs. This is fortunate, since while Prolog is well suited to searching over a set of alternatives, it is not likely to be used for mathematical calculations.

The results of the circuit design programs are also enlightening. Nand.design1 exhibited a tremendous speedup when given the task of constructing a relatively simple circuit, while nand.design2 construction showed far less improvement. This is because nand.design2 constructs the circuit in a more "intelligent" manner by making fewer redundant calculations; hence less is gained by avoiding redundancy. If goal caching will produce significant speedups

only for poorly written programs, then goal caching will not be very interesting. Fortunately, the results indicate that even well written programs contain at least some redundant queries that are not apparent to the programmer. Every single program that performed a search over a solution space exhibited a speedup. Even if the speedup increases linearly with the depth of the tree, and not exponentially, the potential performance improvement for execution trees ten times larger than the ones tested is still an order of magnitude. Of course, if the increase is exponential then the expected speedup will be even greater.

One qualification should be made regarding the speedup estimates. These figures were obtained by dividing the number of unifications during conventional execution by the number of unifications made in accessing the program by the goal caching interpreter. In fact, some unifications are associated with cache access. However, only those involving unbound variables need actually be considered. It is believed that the number of these unifications is negligible when compared with the total number of program unifications.

It is conjectured that speedups of more significant magnitude are to be found in programs with very deep execution trees. The author bases this conjecture on the increase in speedup that is always found (in searching problems) when the tree increases. Solving the queens problem for larger board sizes, designing more complex circuits for nand.design2, and longer derivations for "mu_math" are all believed to exhibit speedups of an order of magnitude. Unfortunately, the current version of the interpreter that employs goal caching is too slow to solve queries that are this involved in a reasonable amount of time. A much faster version is currently under construction.

# CHAPTER 5

## Problems and Unresolved Issues

In this chapter we discuss some open implementation questions, and point out some difficult problems.

### 10. Cache Management Schemes

Up to now, the type of memory assumed has been a fully associative one. In fact, this may be too expensive, as we expect the necessary cache size to be quite large. Hence a direct mapped or set associative scheme might be more desirable. In this case, a hash function would be applied to the goal to produce a binary address, which would then be used to access a conventional memory. The contents of that memory location would then be compared with the original goal (several comparisons may be made if the cache is set associative), with a match indicating a hit and triggering unification with data stored in another memory. Should this particular implementation be chosen, it will become important to develop a good, fast hashing function to generate addresses from goals, so as to minimize collisions. A collision resolution policy will have to be developed; it is not immediately clear what course of action is best when two goals map into the same cache address.

### 11. The Problem of Overflow

Regardless of the cache management strategy chosen, only a finite amount of memory will be available. Thus, the possibility of overflowing that memory must be considered. Once there is no more room in the cache, the only acceptable possibility appears to be to ignore further writes to the cache. Goals attempted after the point in time at which the overflow occurred should only consult the database, while goals still on the environment stack would be

permitted to consult the cache as before. Flushing the cache would not ensure program correctness, because RPDS entries indicate the next clause to try after all answers from the cache have been rejected. Flushing the cache would make the RPDS entries invalid; updating them appropriately would be a difficult and time-consuming operation.

## 12. Side Effects

Consider what would happen if a statement with a side effect, such as "print(X)", was cached. The first time "print(X)" was attempted, it would succeed. It would be stored in the cache, and whatever X was instantiated to would be printed by the interpreter, since print is a built in function. Now, suppose the same statement is attempted again. The cache is checked first, yielding a hit (since we have "solved" this goal before). The "solution" is transmitted, and control proceeds, without the associated side effect of printing X! Although this may not appear to be a serious problem from this example, consider the issues involved in caching assert and retract statements; Prolog instructions that actually modify the program. If the side effects associated with these statements are not executed, goals that succeeded in a conventional Prolog implementation could fail, if their success depended on the assertion or retraction of a clause!

Swinson, Pereira, and Bijl[8] have done work on a related problem, with their analysis of base facts and derived facts. Base facts are assertions in a Prolog database, while derived facts are those produced during execution. The authors maintain consistency in a system in which base facts are constantly being inserted and deleted through a fact dependency system, in which any derived facts obtained from a base fact are deleted when the base fact itself is removed. A similar solution would have to be employed in a goal caching scheme, when a clause that lead to the addition of solutions in the cache is retracted. The "derived facts" in the cache would somehow have to be flushed when the clause or clauses that led to them

disappear.

Only two solutions seem immediately practical. We can a) not use goal caching on programs with side effects, or b) rewrite programs to remove the side effects. Until recently, it was believed that a third option existed: invalidating cache entries associated with all nodes in the execution tree from the root to the statement with the side effect, so that statements with side effects would always get executed. Unfortunately, such cache invalidation presents the same problem with the RPDS entries that flushing does in the overflow case. While option b was chosen for the benchmarks presented in this report, it is not clear which choice is better. It is certain, however, that the side effects problem will have to be effectively addressed in any successful implementation of goal caching on a Prolog machine.

## 13. Variable Length Cache Tags and Data

Prolog goals have variable length; clearly fixed length tags and data will be desired when the cache is actually implemented. One effective way to do this might be to limit the arguments a goal may have, and then simply ignore unused argument slots in the cache. The data items in the cache, however, also have variable length. One solution to a goal may be a constant, while another may be a list or a structure. It is unclear at this point exactly how this problem can best be dealt with, but it is certain that the hardware will impose some structure on Prolog goals and solutions.

# CHAPTER 6

## Conclusions

We have seen how the presence of a goal cache calls for modifications to the LUSH algorithm of Prolog, and have outlined in detail what those modifications are. We have also seen that the underlying hardware is relatively simple. Finally, we have gained some insight into the effect of goal caching on Prolog performance, using a modified Prolog interpreter. It now remains to evaluate the merits of goal caching in light of what has been shown, to arrive at an informed decision on the desirability and feasibility of goal caching.

This report has discussed an algorithm for goal caching: the LUSHC algorithm. Even if a Prolog implementation does not employ the LUSHC algorithm directly, it still must deal with the five basic issues that LUSHC addresses. It has also pointed out, however, the two most serious obstacles that bar the way to improved Prolog performance: the problem of side effects, and bookkeeping overhead.

There does not seem to be a simple way to successfully deal with side effects, apart from removing them entirely from the program. One future direction of investigation might involve the examination of Prolog programs written for a Prolog machine and determining how often side effects occur, and how crucial they are to the program. If most useful Prolog programs contain substantial side effects, then goal caching becomes less desirable.

The amount of bookkeeping necessary to manage a goal cache may be prohibitive as well. Operations that occur frequently, such as recovering from the RPDS, deleting and adding RPDS entries, and updating the cc's, may greatly reduce the number of LIPS per second of a Prolog machine. Unfortunately, the amount of overhead associated with goal

caching is extremely model dependent. As an example, consider the problem of goal comparison in the UNSW interpreter and the PLM. In the UNSW interpreter, goals are represented explicitly as data structures; hence comparing two of them is slow. In the PLM, however, the goal being tried is never explicitly represented; it is implied by the current block of code being executed (and hence is implicit in the program counter). Comparing goals and arguments in the PLM would be simpler and faster, since instead of comparing data structures we would be comparing bytes. Thus it is possible that one implementation would require very little overhead, while for another the extra instructions might be prohibitive.

The results obtained from the UNSW interpreter indicate that some programs do indeed contain significant amounts of redundant computation. While this was a necessary condition for implementing goal caching on a Prolog machine, it is not a sufficient one. The results do not tell us anything about the amount of overhead required to implement goal caching on the PLM [9], the Prolog machine currently under construction here at Berkeley. In order to determine if the required overhead on the PLM is prohibitive, Dobry's PLM simulator should be modified to incorporate goal caching just as the UNSW interpreter was modified. Such modification will shed some much-needed light on the biggest unresolved question in goal caching implementation: how much bookkeeping is required and how is performance affected?

Only if 1) the side effects problem can be effectively dealt with and 2) the overhead involved is not prohibitive can goal caching significantly improve the performance of a Prolog machine. Future research should concentrate on resolving these two questions. However, should these two obstacles be overcome, it is believed that the performance improvement justifies whatever other implementation costs might be associated with goal caching (such as the cost in hardware of a large associative memory). Two factors support this claim: 1) programs that did not exhibit a significant speedup were, for the most part, designed for applications for which it is not likely Prolog will be used, and 2) the results indicate that the

performance improvement obtained from goal caching increases with the depth of the execution tree. Future work will attempt to strengthen the latter conjecture by enabling the interpreter to measure the expected performance improvement of programs with deeper execution trees than is currently possible.

The supercomputers of tomorrow will have to contain new architectural ideas in order to solve currently "impossible" problems. Since these problems often involve search operations over a solution space, architectural features that support searching should be investigated as candidates for implementation in hardware. Goal caching is just such a feature. This report has outlined a theoretical basis for goal caching, and has examined some results that show a necessary condition for implementing goal caching: redundancy in programs. It has also pointed out the two principal questions that should be resolved before attempting a goal caching implementation. It is believed that if these questions can be answered satisfactorily, then goal caching should be considered for inclusion in the architecture of a supercomputer.

## References


1. R. A. Kowalski, "Predicate Logic as Programming Language," *Proc. IFIP74*, pp. 569-574 (1974).

2. M. H. van Emden, "Programming With Resolution Logic," pp. 266-269 in *Machine Intelligence 8*, Elis Horwood, Chichester (1977).

3. C. A. Sammut, R. A. Sammut, "The Implementation of UNSW-PROLOG," *The Australian Computer Journal* **15** pp. 58-64 (1983).

4. Tep Dobry, *Implementing Goal Caching on the PLM*, University of California, Berkeley (1983). Unpublished paper

5. David H. D. Warren, "Logic Programming and Compiler Writing," *Software--Practice and Experience* **10** pp. 97-125 John Wiley & Sons, Ltd., (1980).

6. W. F. Clocksin , *Programming in Prolog*, Springer Verlag, New York (1981).

7. Douglas Hofstader, *Godel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York (1979).

8. Peter S. G. Swinson, Fernando C. N. Pereira, and Aart Bicl, "A Fact Dependency System for the Logic Programmer," *Computer-Aided Design* **15** pp. 235-243 ().

9. David H. D. Warren, *Prolog Engine*, SRI International, Menlo Park, CA (1983).

# APPENDIX A

## The Benchmarks

### DIFF

```
%           Differentiation program.

d(U + V, X, DU + DV) :- !,
        d(U, X, DU),
        d(V, X, DV).
d(U - V, X, DU - DV) :- !,
        d(U, X, DU),
        d(V, X, DV).
d(U * V, X, U * DV + V * DU) :- !,
        d(U, X, DU),
        d(V, X, DV).
d(U / V, X, (DU * V - U * DV) / V ^ 2) :- !,
        d(U, X, DU),
        d(V, X, DV).
d(U ^ N, X, DU * N * U ^ N1) :-
        integer(N), !,
        N1 is N - 1,
        d(U, X, DU).
d(- U, X, - DU) :- !,
        d(U, X, DU).
d(e ^ U, X, DU * e ^ U) :- !,
        d(U, X, DU).
d(sin(U), X, DU * cos(U)) :- !,
        d(U, X, DU).
d(cos(U), X, - DU * sin(U)) :- !,
        d(U, X, DU).
d(ln(U), X, DU / U) :- !,
        d(U, X, DU).
d(X, X, 1) :- !.
d(C, X, 0).

d(((((((((x*x)*x)*x)*x)*x)*x)*x)*x)*x,x,Y)?
d((((((((((x/x)/x)/x)/x)/x)/x)/x)/x)/x,x,Y)?
d(ln(ln(ln(ln(ln(ln(ln(ln(ln(x)))))))))),x,Y)?
d((x+1)*(x^2+2)*(x^3+3),x,Y)?
```

## HANOI

```
% This is the Tower of Hanoi problem.  Typing "hanoi(N)?", where
% N is the number of disks on the tower, will execute the correct
% sequence of moves.  From Clocksin and Mellish.
% To have it print out each move as it does it, invoke inform(A,B)
% in the body of the third clause, in between the two move goals.

hanoi(N) :- move(N,left,center,right).

move(0,_,_,_).
move(N,A,B,C) :- M is N-1, M >= 0, move(M,A,C,B), move(M,C,B,A).

% inform(A,B) :- write([move,disk,from,A,to,B]),nl.
```

MIS.CAN

```
%       Missionaries and Canibals program.
%       Written by Claude Sammut.
%       Execute by 'run!'

op(650, yfx, =>)!

newmove(state(M1, C1, left), state(M2, C2, right)) :-
        move(M, C),
        M <= M1,
        C <= C1,
        M2 is M1 - M,
        C2 is C1 - C,
        ok(M2, C2).
newmove(state(M1, C1, right), state(M2, C2, left)) :-
        move(M, C),
        M2 is M1 + M,
        C2 is C1 + C,
        M2 <= 3,
        C2 <= 3,
        ok(M2, C2).

print_soln(start).
print_soln(X => Y) :-
        print_soln(X),
        write(Y),
        nl.

run :-
        solve(start => state(3, 3, left), X),
        print_soln(X).

ok(X, X) :- !.
ok(3, X) :- !.
ok(0, X).

solve(Init => state(0, 0, right), Init => finish) :- !.
solve(Init => S1, Final) :-
        newmove(S1, S2),
        not(member(S2, Init)),
        solve((Init => S1 => S2), Final).

move(2, 0).
move(1, 0).
move(1, 1).
move(0, 1).
move(0, 2).

member(X, Y => X) :- !.
```

```
member(X, Y ==> Z) :- member(X, Y).
run!
```

MOB

```
%   The following program simulates a two-bit Moebius counter.
%   Queries are in the following format:
%       fun([X,Y], Length)?
%   where [X,Y] is the initial state, Length specifies the length of
%   counting sequence.
% definition of JK_FF
jkff(S, 0, 0, S).
jkff(S, 0, 1, 0).
jkff(S, 1, 0, 1).
jkff(1, 1, 1, 0).
jkff(0, 1, 1, 1).
% definition of an INV function
inv(0, 1).
inv(1, 0).
% description of a two-bit Moebius counter implemented by JK_FF
function([X,Y], Length) :- M is Length-1, M >= 0, inv(Y, J0),
        inv(X, K1), jkff(X, J0, Y, Q0), jkff(Y, X, K1, Q1),
        function([Q0, Q1], M).
function(_, 0).
```

MU_MATH

```
% MU MATH SYSTEM OF HOFSTADER (GODEL, ESCHER, BACH; PP 35-36).

rules(S, R) :- rule3(S, R).
rules(S, R) :- rule4(S, R).
rules(S, R) :- rule1(S, R).
rules(S, R) :- rule2(S, R).

rule1(S,R) :-                    % RULES
      append(X, [i], S),
      append(X, [i, u], R).

rule2([m, ..T], [m, ..R]) :- append(T, T, R).

rule3([], -) :- fail.
rule3(R, T) :-
      append([i, i, i], S, R),
      append([u], S, T).
rule3([H, ..T], [H, ..R]) :- rule3(T, R).

rule4([], -) :- fail.
rule4(R, T) :- append([u, u], T, R).
rule4([H, ..T], [H, ..R]) :- rule4(T, R).

theorem(Depth, [m,i]).
theorem(Depth, []) :- fail.

theorem(Depth, R) :-             % TREE TRAVERSAL
      Depth > 0,
      D is Depth - 1,
      theorem(D, S),
      rules(S, R).

append([], X, X).
append([A, ..B], X, [A, ..B1]) :-    % UTILITY
      !,
      append(B, X, B1).
theorem(2,X)?
theorem(3,X)?
```

NAND.DESIGN1

```
%%% Automated Design of 2-input Combinational Logic Circuits with NAND
%
% Queries are in the following format:
%       fun(A,B,C,D,Level,[])?
% where vector (A, B, C, D) corresponds to the output of the circuit
% when the 2 inputs are (00), (01), (10), (11) respectively,
% Level is the level of the circuits in order to control search
% space.
% Output is is the file "result".
% All the possible configurations of the circuits can be found out by
% tracing backward within each successful instance.
% Reference: "Automated Design of Multiple-Valued Logic Circuits by
%       Automatic Theorem Proving Techniques" by W.Wojciechowski and
%       A.S.Wojcik, IEEE Trans. Comput., vol. c-32, pp. 785-798,
%       Sept. 1983

% NAND function
nand(0, 0, 1).
nand(0, 1, 1).
nand(1, 0, 1).
nand(1, 1, 0).
% INV function
inv(0, 1).
inv(1, 0).
% a NAND gate
function(1, 1, 1, 0, Level, Leg):- M is Level-1, M>=0.

% inputs
function(0, 0, 1, 1, _, Leg).
function(0, 1, 0, 1, _, Leg).
function(1, 1, 0, 0, _, Leg).
function(1, 0, 1, 0, _, Leg).
% gates composition whose last stage is a NAND gate
function(C0, C1, C2, C3, Level, Leg):- M is Level-1, M>=0,
      nand(A0, B0, C0), nand(A1, B1, C1), nand(A2, B2, C2),
      nand(A3, B3, C3), function(A0, A1, A2, A3, M, [1,..Leg]),
function(B0, B1, B2, B3, M, [2,..Leg]).
% gates composition whose last stage is an INV gate
function(C0, C1, C2, C3, Level, Leg):- M is Level-1, M>=0,
      inv(A0, C0), inv(A1, C1), inv(A2, C2),
      inv(A3, C3), function(A0, A1, A2, A3, M, [0,..Leg]).
```

NAND.DESIGN2

```
% Queries are in the following format:
%       fun(A,B,C,D,Level,[])?
% where vector (A, B, C, D) corresponds to the output of the circuit
%       when the 2 inputs are (00), (01), (10), (11) respectively,
%       Level is the level of the circuits in order to control search
%       space.
% All the possible configurations of the circuits can be found out by
%       tracing backward within each successful instance.
% Reference: "Automated Design of Multiple-Valued Logic Circuits by
%       Automatic Theorem Proving Techniques" by W.Wojciechowski and
%       A.S.Wojcik, IEEE Trans. Comput., vol. c-32, pp. 785-798,
%       Sep. 1983

% NAND function
nand(0, 0, 1).
nand(0, 1, 1).
nand(1, 0, 1).
nand(1, 1, 0).
% a NAND gate
function(1, 1, 1, 0, Level, Leg):- M is Level-1, M>=0.

% inputs
function(0, 0, 1, 1, _, Leg).
function(0, 1, 0, 1, _, Leg).
function(1, 1, 0, 0, _, Leg).
function(1, 0, 1, 0, _, Leg).
% gates composition whose last stage is a NAND gate
function(C0, C1, C2, C3, Level, Leg):- M is Level-1, M>=0,
        nand(A0, B0, C0), nand(A1, B1, C1), nand(A2, B2, C2),
        nand(A3, B3, C3),
        weight(A0, A1, A2, A3, J), weight(B0, B1, B2, B3, K),
        order(A0, A1, A2, A3, B0, B1, B2, B3, M, Leg, J, K).
% to avoid, as much as possible, generating the same configuration
order(A0, A1, A2, A3, B0, B1, B2, B3, M, Leg, W1, W2):-
        W1 > W2, function(A0, A1, A2, A3, M, [1,..Leg]),
        function(B0, B1, B2, B3, M, [2,..Leg]).
order(A0, A1, A2, A3, B0, B1, B2, B3, M, Leg, W1, W2):-
        W1 == W2, function(A0, A1, A2, A3, M, [1,..Leg]).
% weight of the output
weight(A0, A1, A2, A3, J):- J is (((((A0*2)+A1)*2)+A2)*2+A3).
```

## QSORT

```
%       Quicksort program.
%       NOTE: '<' works for atoms as well as numbers

sort(L0, L) :- qsort(L0, L, []).

qsort([X, ..L], R, R0) :- !,
        partition(L, X, L0, L1),
        qsort(L1, R1, R0),
        qsort(L0, R, [X, ..R1]).
qsort([], R, R).

partition([X, ..L], Y, [X, ..L0], L1) :-
        X < Y, !,
        partition(L, Y, L0, L1).
partition([X, ..L], Y, L0, [X, ..L1]) :- !,
        partition(L, Y, L0, L1).
partition([], _, [], []).
sort([27,74,17,33,94,18,46,83,65,2,32,53,28,85,99,47,28,82,6,11,55,
29,39,81,90,37,10,0,66,51,7,21,85,27,31,63,75,4,95,99,11,28,61,74,
18,92,40,53,59,8],X)?
```

# QUEENS

```
size(3).
int(1).
int(2).
int(3).

q(X, 3) :-
        solve([], X).                % the work starts here

%       newsquare generates legal positions for next queen

newsquare([], square(1, X)) :- int(X).
newsquare([square(I, J), ..Rest], square(X, Y)) :-
        (X is I + 1),
        int(Y),
        not(threatened(I, J, X, Y)),
        safe(X, Y, Rest).


%       safe checks whether square(X, Y) is threatened by any
%       existing queens

safe(X, Y, []).
safe(X, Y, [square(I, J), ..L]) :-
        not(threatened(I, J, X, Y)),
        safe(X, Y, L).


%       threatened checks whether squares (I, J) and (X, Y)
%       threaten each other

threatened(I, J, X, Y) :-
        (I = X),
        !.
threatened(I, J, X, Y) :-
        (J = Y),
        !.
threatened(I, J, X, Y) :-
        (U is I - J),
        (V is X - Y),
        (U = V),
        !.
threatened(I, J, X, Y) :-
        (U is I + J),
        (V is X + Y),
        (U = V),
        !.
```

```
%       solve accumulates the positions of occupied squares

solve([square(X, Y), ..L], [square(X, Y), ..L]) :- size(X).
solve(Initial, Final) :-
        newsquare(Initial, Next),
   solve([Next, ..Initial], Final).
q(3,X)?
```

QUERY

```
query([C1,D1,C2,D2]) :-
        density(C1,D1),
        density(C2,D2),
        D1>D2,
        20*D1 < 21*D2.

density(C,D) :- pop(C,P), area(C,A), D is (P*100)/A.

pop(china,      8250).  area(china,      3380).
pop(india,      5863).  area(india,      1139).
pop(ussr,       2521).  area(ussr,       8708).
pop(usa,        2119).  area(usa,        3609).
pop(indonesia,  1276).  area(indonesia,  570).
pop(japan,      1097).  area(japan,      148).
pop(brazil,     1042).  area(brazil,     3288).
pop(bangladesh, 750).   area(bangladesh,55).
pop(pakistan,   682).   area(pakistan,   311).
pop(w_germany,          620).   area(w_germany,          96).
pop(nigeria,    613).   area(nigeria,    373).
pop(mexico,     581).   area(mexico,     764).
pop(uk,         559).   area(uk,86).
pop(italy,      554).   area(italy,      116).
pop(france,     525).   area(france,     213).
pop(phillipines,415).   area(phillipines,90).
pop(thailand,   410).   area(thailand,   200).
pop(turkey,     383).   area(turkey,     296).
pop(egypt,      364).   area(egypt,      386).
pop(spain,      352).   area(spain,      190).
pop(poland,     337).   area(poland,     121).
pop(s_korea,    335).   area(s_korea,    37).
pop(iran,       320).   area(iran,       628).
pop(ethiopia,   272).   area(ethiopia,   350).
pop(argentina,  251).   area(argentina,  1080).

query([C1, D1, C2, D2])?
```

SIEVE

```
% The Sieve of Eratosthenes, from Clocksin and Mellish
% primes(N,L) instantiates L to a list of primes <= N

primes(Limit,Ps) :- integers(2,Limit,Is), sift(Is,Ps).

integers(Low, High, [Low, ..Rest]) :-
        Low <= High, !, M is Low+1, integers(M,High,Rest).
integers(_,_,[]).

sift([],[]).
sift([I, ..Is],[I, ..Ps]) :- remove(I,Is,New), sift(New,Ps).

remove(P,[],[]).
remove(P,[I, ..Is],[I, ..Nis]) :- not(0 is I mod P),!,remove(P,Is,Nis).
remove(P,[I, ..Is],Nis) :- 0 is I mod P, !, remove(P,Is,Nis).
primes(16,X)?
```

REVERSE

```
nreverse([X | L0], L) :- nreverse(L0, L1), append(L1, [X], L).
nreverse([], []).

append([X | L1], L2, [X | L3]) :- !, append(L1, L2, L3).
append([], X, X).

nreverse([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30],X)?
```