The Culprit Pointer Method for Selective Backtracking

by

Jeffrey Mark Siskind
B.A. Computer Science, Technion, Israel Institute of Technology
(1979)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements of the Degree of
Master of Science in
Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology
January 1989

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
January 30, 1989

Certified by ______________________________
David Allen McAllester
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

The Culprit Pointer Method for Selective Backtracking

by
Jeffrey Mark Siskind

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 1989, in partial fulfillment of the
requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Many techniques have been proposed for performing selective backtracking in Prolog. These techniques tradeoff time spent in supporting the search pruning versus time saved from the search. One technique which performs very effective pruning is the Multiple Nogood algorithm which computes all minimal nogoods upon unification failure, hyperresolves these nogoods upon goal failure, and selects as the backtrack choice-point, the earliest choice-point among the latest choice-points of each resulting nogood. Unfortunately, this method can require exponential time and space per backtrack decision. This thesis attempts to answer the open question as to whether an algorithm exists which makes the same backtrack decisions as the Multiple Nogood algorithm in polynomial time. A novel algorithm, the Culprit Pointer Method, is presented which appears to make the same backtrack decisions as the Multiple Nogood algorithm, but only for goal failures which do not contain nested failures for which a selective backtrack was taken. This method requires space overhead which is linear in the depth of the choice-point stack, and time overhead per unification failure which is quadratic in the depth of the choice-point stack. The algorithm is very general and can be applied to variety of search problems ranging from constraint satisfaction problems to extensions of Prolog such as CLP($\Re$). This thesis presents the algorithm, proves its correctness, discusses its incorporation into an efficient implementation of Prolog based on the Warren Abstract Machine, and gives some benchmarks of that implementation.

Thesis Supervisor: David Allen McAllester
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

This thesis is long overdue. Although I have spent over five and a half years completing my masters, and three and a half years of that on this project, I am very embarrassed by the result, or lack thereof. If it were not for stubbornness, and the misguidance of faculty and fellow graduate students, I would have abandoned this research topic long ago and switched to a more fruitful one to produce a more satisfying thesis. One thing that I have learned from this experience—something that I hope others will learn as well—is that one *can* make an *honest* mistake in pursuing a research topic that leads nowhere. That can happen no matter how much due diligence you invest up front to determine the project's feasibility and difficulty. It takes a certain amount of responsibility to realize when your assessment of the project changes and to cut ones loses by switching topics. At least three times during this project I had the foresight to talk with the faculty at M. I. T. about switching topics. Each time I was advised to stick with the project, finish the preliminary results that I had, and write them up. No one could accept the fact that despite all the time I had invested in the topic, I had a great deal of understanding but no tangible results. "You must have gained some knowledge which is worth sharing with others", is a sound which still rings in my ears. Traditionally, acceptable results in Artificial Intelligence and Computer Science have fallen into one of three categories: demonstrated useful working systems, proven positive results, and mathematically proven negative results. Unlike other fields, discussions of approaches which have been tried but which have failed are usually not considered respectable without some proof that explains why the failure is fundamental and must have occurred. I did not have any acceptable results then. Unfortunately, two years later I still do not have any results. This thesis is more an expression of bureaucratic desperation rather than scientific accomplishment.

A number of people and organizations have offered tremendous support to me during this long project. Without them I would not have been able to waste so much time and yet learn so much from that process. Charles Leiserson, my first advisor, helped me return to academia after a three year absence. How true was his insight in predicting that it would take me about two years to get back into the swing of things at M. I. T. I am greatly indebted to him for supporting me, and to the theory group for providing me a haven, during that period of readjustment. David McAllester, first an unofficial advisor, and finally the professor who will sign this thesis, has been a continual sounding board for my ideas. Not the kind which just absorbs and muffles the sound, but rather which selectively resonates with and amplifies the good ideas while filtering out the bad ones. About a year ago, David asked whether I would mind having such a junior faculty member as him as an advisor. I replied that every dentist has some patient whose teeth are his first to drill. Well, in retrospect it hasn't been that painful. I truly believe that David will grow up to be a great dentist—oops great professor—some day.

AT&T Bell Laboratories has provided exceedingly generous support for the past two and a half years through a Ph.D. scholarship. I hope someday to repay that generosity in ways above and beyond my long distance phone bill which has grown tremendously during the past month. Xerox PARC has supported me and my research during two summer visits and one winter visit. Being at PARC is like being a child in a candy store—with grandpa there to buy you whatever you want. I can't remember ever having as euphoric a research experience as I have had during my stays at PARC. I can't express my appreciation to Johan de Kleer and Per-Kristian Halvorsen, my supervisors at PARC, for offering me that experience, and for believing in me even when I didn't.

Ramin Zabih has been my partner in crime. Hey Ramin, do you finally agree with me that it is not worthwhile to work on an algorithm with greater than linear complexity!? Philip Klein, Ronald Greenberg, and Tom Cormen have listened to my babbling, given useful feedback, and in general helped unwedge my mathematics as well as my LaTeX macros. Mark Shirley cushioned the brunt of my frustration during the final months of thesis writing. I wish I had gotten to know him better earlier on during his and my tenure at M. I. T. Finally, Jeremy Wertheimer and I spent two years as scientific חברותא, meeting twice a week to push each other along to finish our respective theses. Jeremy, would you ever have believed that we would both actually finish?! Lets get on with it and move on to more productive research topics.

I am far too ashamed to acknowledge my family, friends, and teachers on such a meager and paltry document. That will have to wait for my Ph.D. Perhaps then I can present a document worthy of acknowledging them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Prolog programs, as well as other problem solvers performing blind search, often exhibit anomalous backtracking behavior. Consider the example given in figure 1.1. When the first alternative is taken for the goals `gen(X)` and `gen(Y)`, the goal `test(X)` will fail and backtrack to the goal `gen(Y)` even though no possible alternative for this goal can lead to a solution. We desire a technique which will allow backtracking directly to the goal `gen(X)` in such a situation.

In [29] de Kleer enumerates four different kinds of anomalous search behavior—futile backtracking, rediscovering contradictions, rediscovering inferences, and incorrect ordering—and gives examples of the occurrence of each while solving a simple constraint problem. Each of these anomalous behaviors may also be exhibited by Prolog programs as well as other search problems. Of the four different anomalies, the program in figure 1.1 exhibits mainly *futile backtracking*. This thesis focuses on one technique which attempts to reduce futile backtracking. This technique can be applied both to Prolog implementations as well as to other search problems such as constraint satisfaction problems.

Many techniques have been proposed in the literature for alleviating the search anomalies discussed by de Kleer. Different techniques focus on different subsets of these anomalies. Some attempt to alleviate only futile backtracking while others attempt to alleviate all four of the anomalies. Previously published accounts of search pruning techniques include those applied to Prolog[88, 42], non-deterministic Lisp[11, 95, 97, 98], constraint languages, and a variety of problem solvers[33, 32, 83, 39, 38, 61, 62, 63, 64, 80, 41, 29, 30, 31] Within the framework of Prolog implementations, strategies which focus on

```
?- gen(X),gen(Y),test(X).
gen(a).
gen(b).
gen(c).
gen(d).
gen(e).
test(b).
test(d).
```

Figure 1.1: A Prolog program demonstrating futile backtracking. Failure of the goal `test(X)` will backtrack to the goal `gen(Y)` even though no alternative for that goal will prevent the failure of `test(X)`. Selective backtracking refers to a class of techniques which ideally would backtrack directly to the goal `gen(X)` in situations similar to this.

alleviating only futile backtracking (in contrast to the other three search anomalies) have commonly been called *selective backtracking* or *intelligent backtracking* techniques.[1] Many methods have been proposed for performing intelligent backtracking in PROLOG[8, 23, 25, 9, 57, 55, 50] These techniques differ from one another in both their effectiveness, how well they succeed in pruning the search space, and in their cost, how much overhead is involved in performing the pruning. It can be shown that, irrespective of the cost of applying the different techniques, generally no technique dominates another in the effectiveness of the pruning it performs. Nonetheless, experience has shown that in practice, there is usually a tradeoff between cost and effectiveness; increased effectiveness implies increased cost and decreased cost implies decreased effectiveness. The objective is to choose a technique with the proper balance for solving some class of problems efficiently.

One very effective selective backtracking technique is the MULTIPLE NOGOOD algorithm. Although this algorithm seems never to have been published, it combines ideas from numerous other published methods and seems to be well known. It has been alluded to by a number of authors. The essence of this algorithm is as follows. Conceptually, the MULTIPLE NOGOOD algorithm maintains a single set of nogoods called the *nogood cache*. Each nogood is a set of choices which is known to entail failure. A choice is an association of a choice-point with a particular alternative chosen for that choice-point. Each choice implies some constraint which in the case of PROLOG is the unification of a goal term associated with the choice-point with the head of the clause chosen as the current alternative for that choice-point. A set of choices may be inconsistent, and thus be a nogood, for one of two reasons: either the constraints implied by those choices are themselves immediately inconsistent, or it is known that within the context of the current search problem, there is no solution which is a superset of those choices. On unification failure, the MULTIPLE NOGOOD algorithm computes all minimal non-unifiable subsets of the failed set of unifications and adds the choices underlying those minimal sets to the nogood cache as nogoods. This process, called dependency analysis, produces nogoods of the first type. On goal failure, hyperresolution is used to derive new nogoods from those produced by dependency analysis. These new nogoods are of the second type. Hyperresolution proceeds as follows. The set of all choices corresponding to all of the alternatives of the failing goal is viewed as a positive clause. This clause is hyperresolved against all of the nogoods in the nogood cache which are viewed as negative clauses. If the positive clause contains the choices $\{\alpha_1, \ldots, \alpha_k\}$, then for each choice $\alpha_i \in \{\alpha_1, \ldots, \alpha_k\}$ find a nogood n_i in the nogood cache which contains that assumption. A new nogood can be computed from these nogoods by the following formula:

$$\bigcup_{i=1}^{n} n_i - \{\alpha_i\}$$

The new nogoods produced by hyperresolution are then added to the nogood cache. The culprit, i.e. the desired backtrack point, is selected by finding the most recent choice-point from each nogood produced by hyperresolution and selecting the least recent choice-point from this set. Finally, when a choice-point is popped off the stack, all nogoods in the nogood cache which reference this choice-point are discarded. This last step forfeits the ability to perform lateral pruning[95, 97, 98] in an attempt to reduce the overhead of the algorithm.

Although this algorithm can be particularly effective at reducing futile backtracking, that effectiveness comes at a high cost. Wolfram has shown[93] that there may be an exponential number of minimal non-unifiable subsets of a set of unifications. Therefore, dependency analysis, and thus the process of making each backtracking decision can require exponential time and space, at least with a straightforward implementation. It is an open question whether an algorithm exists which is as effective as the MULTIPLE

[1]The terms dependency-directed, intelligent, and selective backtracking have been used both imprecisely and ambiguously in the literature to refer to various techniques which attempt to alleviate different subsets of the aforementioned anomalies. Throughout this thesis, the term *selective backtracking* is used to refer to *any* technique which maintains the stack based discipline of depth-first search but which allows backtracking directly to a choice which is not the most recent one on the stack. With the exception of chapter 6, this thesis refrains from using the other terminology.

NOGOOD algorithm but whose time and space complexity for making a backtrack decision at each failure are polynomial in the depth of the choice-point stack at that failure. Note that it is conceivable that such an algorithm exist despite the fact that for finite problems, the search problem as a whole is $\mathcal{NP}$-complete because finding a solution may still require an exponential number of (polynomial time computable) backtracks.

This thesis discusses an attempt to answer this question. An algorithm is presented, called the CULPRIT POINTER METHOD, which makes backtrack decisions in polynomial time. One advantage of this algorithm is that it is applicable to almost any search problem, not just ones like PROLOG, which are based on unification. Its drawback however, is that it only performs selective backtracking for failures which are not selectively-nested. A failure is selectively-nested if a selective backtrack was performed for a failure nested under the currently failing choice-point. On the positive side, it seems that the CULPRIT POINTER METHOD does make the same selective backtrack decisions as the MULTIPLE NOGOOD algorithm for non-selectively-nested failures although this conjecture has not yet been proven.

This thesis addresses the open question of whether an algorithm exists which makes the same backtrack decisions as the MULTIPLE NOGOOD algorithm with polynomial overhead per backtrack decision. During the research which led to this thesis, significant effort was expended both trying to find such an algorithm, as well as proving that it could not exist. While the CULPRIT POINTER METHOD comes close, it appears to do so only for non-selectively-nested failures and even this conjecture has not been proven. Wolfram[93] believes that his results show that a polynomial time equivalent to the MULTIPLE NOGOOD algorithm cannot exist. The last chapter of this thesis discusses why I believe that his results are not conclusive and that if it indeed turns out that a polynomial time equivalent to the MULTIPLE NOGOOD algorithm does not exist, techniques more powerful then those given in [93] will be necessary to demonstrate this fact.

The remainder of this thesis is divided into five chapters. Chapter 2 present the CULPRIT POINTER METHOD as a technique for performing selective backtracking while solving constraint satisfaction problems (CSPs). The CULPRIT POINTER METHOD is presented first in the context of CSPs rather than directly for PROLOG because CSPs lack the hierarchal dependent choices of PROLOG and are thus a simpler framework for presenting the essential ideas of the algorithm. In this chapter, the algorithm is presented, its soundness is proved, and several examples of its operation are given. In particular, an example is given which demonstrates how the technique can be unsound if it is extended to selectively-nested failures. Chapter 3 extends the CULPRIT POINTER METHOD to PROLOG. An abstraction of PROLOG called AND/OR-trees is presented which can model a variety of search problems other than those based on unification. Since the CULPRIT POINTER METHOD is applicable to all such AND/OR-trees, it can be extended without modification to other constraint-based search languages such as CLP($\Re$)[46, 47, 45].

Many previous search pruning algorithms, especially those based on computing, manipulating, and storing nogoods, are difficult to integrate into efficient PROLOG implementations without incurring a large overhead. One advantage of the CULPRIT POINTER METHOD is that in addition to having low complexity on failure, the constant factor of the overhead is small because it can be integrated fairly smoothly into efficiently compiled PROLOG code. Chapter 4 illustrates this by presenting a PROLOG compiler based on the Warren Abstract Machine which has been modified to generate code which incorporates the CULPRIT POINTER METHOD. Chapter 5 compares the performance of this compiler—with the CULPRIT POINTER METHOD both enabled and disabled—on a number of well known benchmarks from the literature. Chapter 6 summarizes the results of the thesis and offers some guidelines for future work.

Chapter 2

The Culprit Pointer Method

This chapter describes an algorithm for solving constraint satisfaction problems (CSPs) using a technique called the CULPRIT POINTER METHOD. Constraint satisfaction problems are simpler than PROLOG programs since CSPs have only a single level of choices and lack the dependent choices needed to represent arbitrary PROLOG programs. Because of the simpler nature of CSP problems over PROLOG programs, the CULPRIT POINTER METHOD is presented first in the context of CSPs. The next chapter extends the CULPRIT POINTER METHOD to cover a broader class of search problems which include dependent choices, including PROLOG programs.

2.1 The Algorithm

A constraint satisfaction problem[49] consists of a finite set of variables $\{x_1, \ldots, x_n\}$ where each x_i ranges over some enumerable set D_i called its *domain*. An *assignment* X is a partial map from variables to elements in their domains. An assignment is *complete* if it maps all n variables to some value. Along with the variables and their domains, a function CHECK is provided which maps assignments, both complete and incomplete, to $\{\textbf{true}, \textbf{false}\}$. If CHECK($X$) returns **true** then we say that X is consistent; otherwise it is inconsistent. We require that CHECK be monotonic, i.e. every subset of every consistent set must be consistent, and every superset of every inconsistent set must be inconsistent. The problem then, is to find one (or perhaps all) complete consistent assignments.

One method for solving constraint satisfaction problems is depth-first backtracking search. It is well known that in many situations, backtracking search can exhibit thrashing behavior. The CULPRIT POINTER METHOD is a technique, which when added to backtracking search, can sometimes reduce the amount of thrashing at the expense of some more computation.

An assignment X' is an *extension* of another incomplete assignment X if X' agrees with X on all of the variables mapped by X. During depth-first search, finding that an assignment is inconsistent is called a *check failure*. Furthermore, completing the exploration of all extensions of some assignment $\{x_1, \ldots, x_{j-1}\}$ is called an *exhaustion failure* of choice-point j. Some of these extensions may be solutions (i.e. complete and consistent) while others may be found to be inconsistent. If some assignment $\{x_1, \ldots, x_{j-1}\}$ exhausts because for every value in the domain of x_j the extension $\{x_1, \ldots, x_j\}$ is inconsistent, then this exhaustion is called an *internal* exhaustion failure. Exhaustion failures which are not internal are called *external*. In normal backtracking, when some assignment $\{x_1, \ldots, x_{j-1}\}$ exhausts, the search continues by exploring the extensions of $\{x_1, \ldots, x_{j-2}\}$ which have not yet been considered. This is called *backtracking* from choice-point j to choice-point $j-1$. Sometimes, however, there exists some $k < j-1$ such that when $\{x_1, \ldots, x_k, \ldots, x_{j-1}\}$ exhausts, it can be proven that no complete extension of $\{x_1, \ldots, x_k\}$, is consistent. In this case, it would be *sound* to ignore searching those extensions and resume the search with an extension of $\{x_1, \ldots, x_{k-1}\}$ which has not yet been considered. This is called *selectively backtracking* from choice-point j to choice-point k.

There are many methods for performing sound selective backtracking. Some perform selective backtracking only for external exhaustion failures. The method described in this thesis, the CULPRIT POINTER METHOD, performs sound selective backtracking not only at external exhaustion failures but many internal ones as well. The essence of the CULPRIT POINTER METHOD, is captured by the following soundness theorem.

Theorem 1 *If Y is an assignment, and X is an extension of Y, and D is a set of extensions of Y such that*

- *every complete extension of X is an extension of some assignment in D, and*
- *no assignment in D maps variables that are mapped by X but not by Y, and*
- *every assignment in D is inconsistent,*

then every complete extension of Y is inconsistent.

> *Proof:* Since CHECK is monotonic, it suffices to show that every complete extension of Y is an extension of some assignment in D. This can be proved by contradiction. Assume that some assignment Z is a complete extension of Y but is not an extension of any assignment in D. Construct the assignment Z' which is an extension of X which agrees with Z for those variables not mapped by X. Since Z' is a complete extension of X it must be an extension of some assignment in D. But since Z' differs from Z only on variables mapped by X but not Y, and no assignment in D maps these variables, Z must be an extension of the same assignment in D as Z'. Contradiction. $\Box$

The soundness theorem provides a sufficient (though not necessary) condition for determining when it is sound to selectively backtrack from choice-point j to choice-point k. Let X be an assignment

$$\{x_1, \ldots, x_k, \ldots, x_{j-1}\}$$

which exhausts and let Y be the assignment $\{x_1, \ldots, x_k\}$ for some $k < j$. If X exhausts without finding a solution then it does so by examining a finite set D' of inconsistent extensions of X of the form

$$\{x_1, \ldots, x_k, \ldots, x_{j-1}, x_j, \ldots, x_i\}$$

where i and $x_j, \ldots, x_i$ vary for each extension. Take D to be the assignments in D' with the mappings for $x_{k+1}, \ldots, x_{j-1}$ removed. Each assignment in D thus has the form

$$\{x_1, \ldots, x_k, x_j, \ldots, x_i\}.$$

Finding the largest k such that each assignment in D is inconsistent will allow selectively backtracking from choice-point j to choice-point k.

In ordinary depth-first search the first and third conditions of the soundness theorem are true whenever some choice-point j exhausts without finding a solution. The CULPRIT POINTER METHOD is a technique for finding some k such that the second condition of the soundness theorem is also true, allowing a selective backtrack from choice-point j to choice-point k. It does this by maintaining for each choice-point j, a *culprit pointer* K_j. This culprit pointer is initialized to zero when the choice-point is created. Whenever during the search some assignment $\{x_1, \ldots, x_i\}$ is found to be inconsistent, all culprit pointers K_j for $1 \leq j \leq i$ are updated by the following method. If the assignment $\{x_j, \ldots, x_i\}$ is inconsistent

Algorithm SOLVE CSPS USING THE CULPRIT POINTER METHOD:

1. [INITIALIZATION] Set i to 0.
2. [SOLUTION] If $i = n$ then $\{x_1, \ldots x_n\}$ is a solution. Reset the culprit pointers by looping over all $j \leq n$ to set K_j to $j - 1$ and go to step 4.
3. [CREATE A CHOICE-POINT] Set i to $i + 1$, K_i to 0, and A_i to D_i.
4. [BACKTRACK] If A_i is empty then backtrack. If $K_i \neq i - 1$ then it is first necessary to reset the culprit pointers by looping over all $j \leq K_i$ and setting K_j to $j - 1$. Backtrack by setting i to K_i. If $i = 0$ then halt. Otherwise, repeat step 4.
5. [CHOOSE AN ALTERNATIVE] Choose a member x_i of A_i. Remove x_i from A_i.
6. [CHECK CONSISTENCY] If $\{x_1, \ldots, x_i\}$ is consistent then go to step 2.
7. [UPDATE CULPRIT POINTERS] For each j from 1 to i such that $\{x_j, \ldots, x_i\}$ is consistent find the least k such that $\{x_1, \ldots, x_k, x_j, \ldots, x_i\}$ is inconsistent and set K_j to the maximum of k and its previous value. Go to step 5. □

Figure 2.1: A depth-first search algorithm for solving constraint satisfaction problems which incorporates the CULPRIT POINTER METHOD for performing selective backtracking at non-selectively-nested failures.

then K_j is not updated. Otherwise, find the smallest k such that the assignment $\{x_1, \ldots, x_k, x_j, \ldots, x_i\}$ is inconsistent. Such a k clearly exists since if $k = j - 1$ the original inconsistent assignment results. Update K_j to the maximum of k and its previous value.

Whenever choice-point j exhausts, it is sound to backtrack to K_j (which results is a selective backtrack if $K_j < j - 1$) if two criteria are met. First, no solution has been found as an extension of $\{x_1, \ldots, x_{j-1}\}$. This is necessary to enforce the third condition of the soundness theorem. Second, no selective backtrack has been taken for some choice-point $j' > j$ on an extension of $\{x_1, \ldots, x_{j-1}\}$. This is necessary to enforce the first condition of the soundness theorem. One way to enforce both of these restrictions is to simply reset all of the culprit pointers K_j to $j - 1$ whenever either a solution is found, or a selective backtrack is taken. The exhaustion of some choice-point j is termed *selectively-nested* if some choice-point $j' > j$ exhausted and performed a selective backtrack on an extension of $\{x_1, \ldots, x_{j-1}\}$. Figure 2.1 gives a variation on depth-first search for solving constraint satisfaction problems which enforces the soundness theorem using culprit pointers. This algorithm performs selective backtracks only for exhaustion failures which are not selectively-nested.

2.2 An Example

The easiest way to illustrate the operation of the CULPRIT POINTER METHOD on constraint satisfaction problems is by way of an example. Consider the well known N-Queens problem, the problem of placing n non-attacking queens on an $n \times n$ chess board. Although deterministic methods, as well as fast nondeterministic heuristic search techniques are known[85, 40, 79] for finding one solution to the N-Queens problem for each n, no fast method is known for enumerating all solutions for a given n, so at present there is no recourse but to resort to search.

In the N-Queens problem, situations arise where it is sound to selectively backtrack upon an external

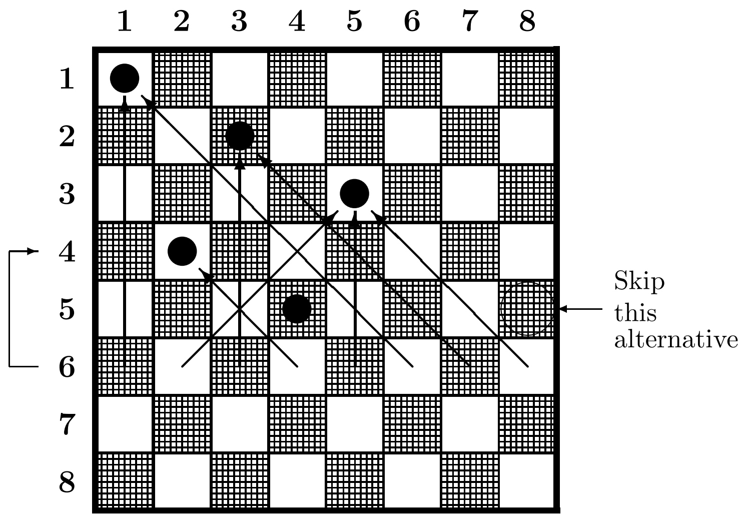

Figure 2.2: An example demonstrating selective backtracking at an external exhaustion in the 8-Queens problem. Row 6 is an external exhaustion since all of its alternatives are attacked by previously placed queens. Since each alternative is still attacked even if the queen in row 5 is removed, the search can selectively backtrack to row 4, thus skipping over the alternative $(5, 8)$ in row 5.

exhaustion failure. Figure 2.2 shows a scenario which might arise after placing five queens on an 8×8 board in positions $(1, 1)$, $(2, 3)$, $(3, 5)$, $(4, 2)$, and $(5, 4)$. When attempting to place a queen in row 6 we discover that all of the alternatives for that row are attacked by previously placed queens; i.e. it externally exhausts. Applying the Culprit Pointer Method to this problem maintains a culprit pointer for row 6 as follows: It is set to row 1 when position $(6, 1)$ is found to be inconsistent, and then to row 3 when position $(6, 2)$ is found to be inconsistent. When position $(6, 3)$ fails, there is no change in the culprit pointer since its current value is row 3 and the earliest queen which attacks $(6, 3)$ is in row 2. The failure of position $(6, 4)$ sets the culprit pointer to row 4. As no subsequent alternative increments the culprit pointer any further, the search can selectively backtrack to row 4 when row 6 exhausts. This eliminates the consideration of the consistent alternative $(5, 8)$ in row 5.

Although the previous scenario illustrates selective backtracking on an external exhaustion failure, situations can arise where it is sound to backtrack selectively on an internal exhaustion failure as well. Such a situation is illustrated by the example in figure 2.3.[1] In this scenario, six queens have been placed on an 8×8 board at locations $(1, 2)$, $(2, 6)$, $(3, 3)$, $(4, 7)$, $(5, 4)$, and $(6, 1)$. When trying to place a queen in row 7 we find that positions $(7, 1)$, $(7, 2)$, $(7, 3)$, $(7, 4)$, $(7, 6)$, $(7, 7)$, and $(7, 8)$ are attacked by queens in rows 1 through 4 even if the queens in rows 5 and 6 are removed. Although position $(7, 5)$ is not under attack, placing a queen at position $(7, 5)$ causes row 8 to exhaust. Furthermore, each of the alternatives for row 8 is still attacked even if the queen in row 6 is removed (but not if the queen in row 5 is removed). Therefore, there is no need to try the remaining alternative $(6, 8)$ for row 6 as this will inevitably lead to a failure. When row 7 exhausts, it can selectively backtrack to row 5.

Lets see how the Culprit Pointer Method applies to the example in figure 2.3. Before placing any queens in row 7, a choice-point is created with its culprit pointer initialized to 0. When a queen

[1] I highlight this scenario because not all selective backtracking techniques can perform as well as the Culprit Pointer Method for internal exhaustion failures such as the one in this scenario.

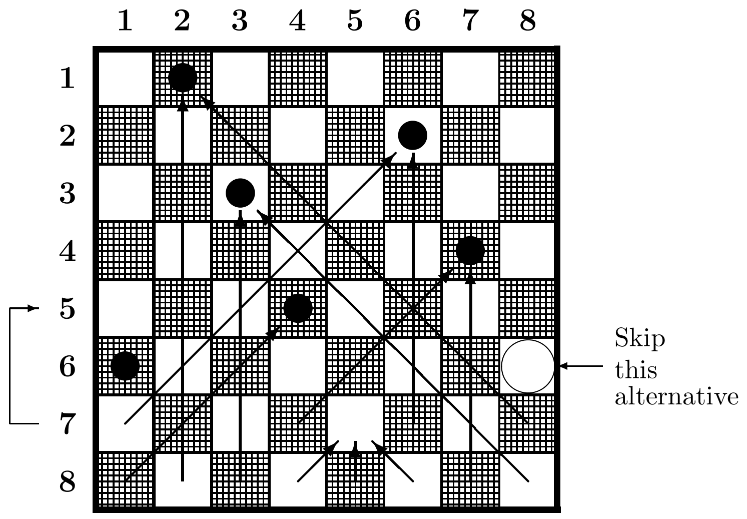

Figure 2.3: An example demonstrating selective backtracking at an internal exhaustion in the 8-Queens problem. Row 7 is an internal exhaustion since although position $(7, 5)$ is not attacked by previously placed queens, placing a queen at that position will result in row 8 exhausting. Even if the queen in row 6 is removed, all of the alternatives for row 7, except for $(7, 5)$ remain under attack, and likewise, after placing a queen at $(7, 5)$, all of the alternatives for row 8 remain under attack. Thus the search can selectively backtrack from row 7 to row 5, skipping over the alternative $(6, 8)$ in row 6.

is placed at $(7,1)$ the culprit pointer for row 7 is set to 2. When a queen is placed at $(7,2)$ the culprit pointer for row 7 is unchanged because the newly placed queen is inconsistent with the queen in row 1. Placing a queen at $(7,3)$ sets the culprit pointer for row 7 to 3, while placing a queen at $(7,4)$ sets it to 4. Position $(7,5)$ is consistent so a new choice-point is created for row 8 and its culprit pointer is initialized to 0. Placing a queen at $(8,1)$ sets the culprit pointers for both row 7, as well as row 8, to 5. Placing queens at either $(8,2)$ or $(8,3)$ does not change either culprit pointer as these locations are attacked by queens in rows 1 and 3 respectively. Placing a queen at $(8,4)$ does not update the culprit pointer for row 7 as the queens in rows 7 and 8 are mutually inconsistent. It could potentially update the culprit pointer for row 8 except that that culprit pointer already points to row 5 which is the value it would be updated to. Placing a queen at $(8,5)$ does not update the culprit pointer for row 7 for the same reason as before. It does however, update the culprit pointer for row 8 to point to row 7. At this point, the culprit pointer for row 8 cannot be incremented any further. When row 8 exhausts, no selective backtracking will occur. Alternative $(8,6)$ does not cause the culprit pointer for row 7 to be updated because it is inconsistent with $(7,5)$. The remaining alternatives for row 8, namely $(8,7)$ and $(8,8)$, do not increment the culprit for row 7 as they are inconsistent with rows 3 and 1 respectively, and the culprit pointer already points to row 5. At this point row 8 exhausts, backtracking to row 7. The remaining alternatives for row 7, namely $(7,6)$, $(7,7)$, and $(7,8)$, do not increment the culprit pointer for row 7 as they are attacked by queens in rows 2, 3, and 1 respectively, and the culprit pointer already points to row 5. Thus when row 7 exhausts, it can selectively backtrack to row 5.

The soundness of the CULPRIT POINTER METHOD crucially depends on performing selective backtracks only at failures which are not selectively-nested. If a selective backtrack is attempted at a failure which is selectively-nested, then a solution can be missed. This is illustrated by the scenario arising during the 11-Queens example shown in figure 2.4.[2] At this point seven queens have already been placed at locations $(1,2)$, $(2,5)$, $(3,8)$, $(4,11)$, $(5,1)$, $(6,4)$, and $(7,6)$. A full trace of how the search proceeds from this point is given in table 2.1. The table shows the queen positions for rows 1 through 11 as well as the culprit pointers for rows 8 through 11 after each inconsistent queen placement. The problem illustrated by this example is that when row 8 exhausts, the search selectively backtracks to row 6, skipping over the alternative $(7,7)$ for row 7, which would have lead to the solution $(2,5,8,11,1,4,7,10,3,9,6)$. This happens because, the selective backtrack from row 11 to row 9 prevents the consideration of the partial solution $(2,5,8,11,1,4,6,10,3,9)$ which would have incremented the culprit pointer for row 8 to point to row 7. Backtracking to row 7, instead of to row 6, when row 8 exhausts, would have lead to a solution.

2.3 Complexity Analysis

In general, the overhead required to maintain and update the culprit pointers is $O(n^2)$ additional calls to CHECK upon each CHECK failure. This is to determine the earliest k needed to update the culprit pointer for each choice-point j. Two simple optimizations are possible. First, if the culprit pointers are updated in decreasing order for j, then whenever $\{x_j,\ldots,x_i\}$ is found to be inconsistent, the updating process can be terminated. Second, whenever some $K_j = j-1$ it is not necessary to perform the $O(n)$ calls to CHECK to find the smallest k in that situation since the culprit pointer is already at its maximum value. In general, since a culprit pointer can only be increased, it is only necessary to check the consistency of assignments $\{x_1,\ldots,x_k,x_j,\ldots,x_i\}$ for $k > K_j$. If calls to CHECK during culprit pointer updating are limited to such values of k, each time that CHECK returns **true** a culprit pointer is actually incremented. Thus, for each CHECK failure, it is only necessary to perform at most $j-K_j-1$ calls to CHECK to

[2] When this algorithm was first developed this was not realized. The unsoundness of the algorithm without this check (i.e. the fact that it does not enforce the first requirement of the soundness theorem) was only discovered four months later as a result of this example. It is surprising that despite the extensive testing of implementations of this algorithm, it took so long, and such a nontrivial example, to discover this bug.

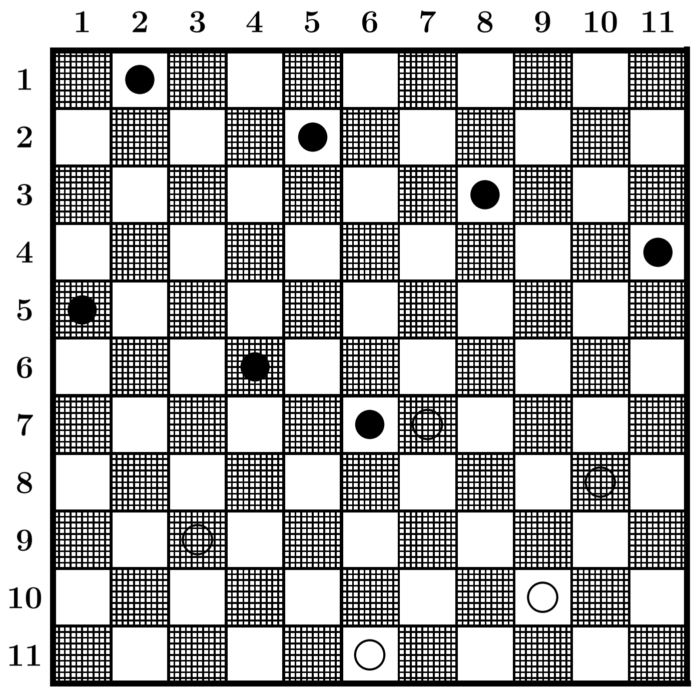

Figure 2.4: An scenario arising during the 11-Queens problem demonstrating how the Culprit Pointer Method can be unsound if applied to selectively-nested exhaustion failures. A trace of the backtracking behavior exhibited by this example is given in table 2.1. The solid dots indicate queens positions at an exhaustions failure for row 8. Selectively backtracking to row 6 at this exhaustion failure misses the solution indicated by the solid dots in rows 1 through 6 combined with the outlined dots in rows 7 through 11.

Queen Positions											Culprit Pointers				Comments
1	2	3	4	5	6	7	8	9	10	11	8	9	10	11	
2	5	8	11	1	4	6	1				5				Row 8's culprit ptr. becomes 5
2	5	8	11	1	4	6	2				5				
2	5	8	11	1	4	6	3				5				
2	5	8	11	1	4	6	4				5				
2	5	8	11	1	4	6	5				5				
2	5	8	11	1	4	6	6				6				Row 8's culprit ptr. becomes 6
2	5	8	11	1	4	6	7				6				
2	5	8	11	1	4	6	8				6				
2	5	8	11	1	4	6	9				6				
2	5	8	11	1	4	6	10	1			6	5			Row 9's culprit ptr. becomes 5
2	5	8	11	1	4	6	10	2			6	5			
2	5	8	11	1	4	6	10	3	1		6	5	3		Row 10's culprit ptr. becomes 3
2	5	8	11	1	4	6	10	3	2		6	5	3		
2	5	8	11	1	4	6	10	3	3		6	5	7		Row 10's culprit ptr. becomes 7
2	5	8	11	1	4	6	10	3	4		6	5	7		
2	5	8	11	1	4	6	10	3	5		6	5	7		
2	5	8	11	1	4	6	10	3	6		6	5	7		
2	5	8	11	1	4	6	10	3	7	1	6	5	7	5	Row 11's culprit ptr. becomes 5
2	5	8	11	1	4	6	10	3	7	2	6	5	7	5	
2	5	8	11	1	4	6	10	3	7	3	6	5	9	9	Row 10's and row 11's culprit ptr. become 9
2	5	8	11	1	4	6	10	3	7	4	6	5	9	9	
2	5	8	11	1	4	6	10	3	7	5	6	5	9	9	
2	5	8	11	1	4	6	10	3	7	6	6	5	9	9	
2	5	8	11	1	4	6	10	3	7	7	6	5	9	9	
2	5	8	11	1	4	6	10	3	7	8	6	5	9	9	
2	5	8	11	1	4	6	10	3	7	9	6	6	9	9	Row 9's culprit ptr. becomes 6
2	5	8	11	1	4	6	10	3	7	10	6	7	9	9	Row 9's culprit ptr. becomes 7
2	5	8	11	1	4	6	10	3	7	11	6	7	9	9	Row 11 exhausts, selectively backtrack to row 9
2	5	8	11	1	4	6	10	4			6	7			
2	5	8	11	1	4	6	10	5			6	7			
2	5	8	11	1	4	6	10	6			6	7			
2	5	8	11	1	4	6	10	7			6	7			
2	5	8	11	1	4	6	10	8			6	7			
2	5	8	11	1	4	6	10	9			6	8			Row 9's culprit ptr. becomes 8
2	5	8	11	1	4	6	10	10			6	8			
2	5	8	11	1	4	6	10	11			6	8			Row 9 exhausts, backtrack to row 8
2	5	8	11	1	4	6	11				6				Row 8 exhausts, unsound backtrack to row 6

Table 2.1: A trace of the unsound backtracking behavior exhibited during the search of the 11-Queens example in figure 2.4. When row 11 exhausts, a selective backtrack is taken to row 9. This selective backtrack prevents encountering a CHECK failure which would update the culprit pointer for row 8 to point to row 7. Because of this, row 8 selectively backtracks to row 6 when it exhausts when it can soundly backtrack only to row 7.

update each K_j. Additionally, this places an upper bound of $j - K_j - 1$ on the number of times that Check can return **true** while maintaining K_j among *all* Check failures during the lifetime of choice-point j. Unfortunately, there is no similar bound on the number of times that Check can return **false** in such situations. Since when $K_j = j - 1$ no calls to Check are needed, while Check must be called whenever $K_j < j - 1$, even if the difference $j - K_j - 1$ is small, it may be the case that more calls to Check are expended to prove that a selective backtrack is possible then are actually saved by the resulting selective backtrack. More work is needed to determine when it is desirable to punt the task of finding a selective backtrack point for a given choice-point j, by setting K_j to $j - 1$, thus reducing the effort expended.

In the N-Queens problem, or more generally any binary CSP problem, special structure allows a greater reduction in the overhead of maintaining the culprit pointers. Since each inconsistency results from a pairwise interaction of only two queens, and since each call to Check checks a new extension to a previously consistent assignment, Check need only check the consistency of the new queen placement with each of the previously placed queens. As a by-product of this single call to Check, which takes $O(n)$ time, the index k, of the earliest queen, as well as the index l, of the latest queen which attack the new placement, can be calculated in the same $O(n)$ time. Using these indices it is possible to update all of the culprit pointers in $O(n)$ time with no further calls to Check. Simply update only those culprit pointers for choice-points $j > l$ to the maximum of the single k just found, and their previous value.

Chapter 3

Applying the Culprit Pointer Method to Prolog

The previous chapter described the CULPRIT POINTER METHOD, a technique for performing selective backtracking on non-selectively-nested failures while solving constraint satisfaction problems. This chapter extends this technique to apply to the execution of PROLOG programs.

3.1 AND/OR-Trees

It is easiest to describe how the CULPRIT POINTER METHOD can be applied to PROLOG by examining an abstraction of PROLOG called AND/OR-trees.

Definition: An *AND/OR-tree* T is a (possibly infinite) tree where each vertex is either an *atomic vertex* labeled with an *atomic constraint*, or otherwise it is labeled either $\wedge$ or $\vee$. Non-atomic vertices must have a finite number of daughters, while atomic vertices must not have any daughters. Furthermore, we impose the following additional restrictions on AND/OR-trees.

- The root vertex must be labeled $\wedge$.
- Every daughter of every $\vee$-vertex must be labeled $\wedge$.
- One daughter of every $\wedge$-vertex, except for the root, must be an atomic vertex.
- The non-atomic daughters of every $\wedge$-vertex must be labeled $\vee$.
- Every $\vee$-vertex must have at least one daughter.

In this abstraction, the problem solver is presented with an AND/OR-tree T along with a function CHECK which defines the meaning of the atomic constraints appearing in T. CHECK maps sets of atomic constraints into $\{\mathbf{true}, \mathbf{false}\}$. If CHECK$(X)$ returns **true** we say that X is *consistent*; otherwise we say that X is *inconsistent*. The CHECK function defines the solutions of an AND/OR-tree.

Definition: An *AND-subtree* T' of an AND/OR-tree T is a subtree of T such that:

- T' contains the root of T,
- for every vertex u in T' except for the root of T, T' contains the parent of u, and
- for every $\vee$-vertex u in T', T' contains at most one daughter of u in T.

An AND-subtree T' of an AND/OR-tree T is *complete* if:

- for every $\wedge$-vertex u in T', T' contains all of the daughters of u in T, and
- for every $\vee$-vertex u in T', T' contains exactly one daughter of u in T.

The *atomic constraint set* of a (finite) subtree of an AND/OR-tree is the set of atomic constraints labeling its atomic vertices. A (finite) subtree of an AND/OR-tree is *consistent* if its atomic constraint set is consistent. A *solution* of an AND/OR-tree T is the atomic constraint set of some complete consistent AND-subtree of T.

The goal of a problem solver is to enumerate the solutions of an AND/OR-tree given together with a CHECK function.

PROLOG programs can be unraveled, with suitable renaming of variables, into (potentially infinite) AND/OR-trees. The root $\wedge$-vertex corresponds to a query which is a conjunction of goals. Each $\vee$-vertex corresponds to a goal where the daughters are alternative ways of satisfying that goal. Each $\wedge$-vertex besides the root corresponds to an attempt to use some clause to satisfy the goal corresponding to its parent vertex. One daughter of such an $\wedge$-vertex is labeled with an atomic constraint, which is a pair of terms $\langle s, t\rangle$, where s is the goal literal and t is the head literal of the clause being used to satisfy the goal. The remaining daughters of the $\wedge$-vertex, if any, correspond to the goals in the body of the clause. For PROLOG programs, the CHECK function is the unifier, which when given a set of pairs $\{\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle\}$, determines whether there is a most general unifier σ such that $\sigma(s_1) = \sigma(t_1), \ldots, \sigma(s_n) = \sigma(t_n)$. Various extensions of PROLOG, like CLP($\Re$)[46, 47, 45], correspond to replacing the unifier with a more comprehensive CHECK function capable of determining the consistency of atomic constraints which are more expressive than simple unifiability of terms. The generality of the AND/OR-tree abstraction allows the CULPRIT POINTER METHOD to apply to these extensions as well.

3.2 Searching AND/OR-Trees Using Depth-First Search

Searching an AND/OR-tree to enumerate its solutions requires a representation for the AND/OR-tree. Since the AND/OR-trees corresponding to unraveled PROLOG programs may be infinite, we will represent them intensionally via the following three functions:

ROOT() Returns the root vertex of the AND/OR-tree. By definition, this will be an $\wedge$-vertex.

ANDDAUGHTERSOFOR(u) Returns the set of daughters of the $\vee$-vertex u. By definition, these will all be $\wedge$-vertices.

ATOMICCONSTRAINT(v) Returns the label of the lone atomic daughter of the $\wedge$-vertex v. This function can not be called on the root vertex.

ORDAUGHTERSOFAND(v) Returns the set of non-atomic daughters of the $\wedge$-vertex v. By definition, these will all be $\vee$-vertices.

Using these functions it is possible to define a search procedure for finding the solutions to an AND/OR-tree which operates analogously to the conventional search procedure used by PROLOG. The code for this procedure is given in figure 3.1. It is this procedure which will be modified to incorporate the CULPRIT POINTER METHOD.

This search procedure, DEPTH-FIRST SEARCH, operates by starting with a trivial AND-subtree containing only the root, and extending this AND-subtree by the addition of vertices, until it is either complete or inconsistent. Whenever it extends an AND-subtree with an $\vee$-vertex it sets up a *choice-point* corresponding to that vertex. As the search is depth-first, the choice-points are maintained on

```
  **procedure** DEPTH-FIRST SEARCH
    $i \leftarrow 0$;
    $C \leftarrow NewVector$;
    $C[0].S \leftarrow \{\}$;
    $C[0].F \leftarrow$ ORDAUGHTERSOFAND(ROOT());
    $C[0].A \leftarrow \{\}$;
*loop* :
    **if** $C[i].F = \{\}$
    **then** SOLUTION($C[i].S$); **go to** *fail* **fi**;
    $u \leftarrow choose(C[i].F)$;
    $C[i].F \leftarrow C[i].F - \{u\}$;
    $i \leftarrow i + 1$;
    $C[i].A \leftarrow$ ANDDAUGHTERSOFOR($u$);
*fail* :
    **if** $C[i].A = \{\}$
    **then** $i \leftarrow i - 1$;
        **if** $i < 0$ **then return else go to** *fail* **fi fi**;
    $v \leftarrow choose(C[i].A)$;
    $C[i].S \leftarrow C[i-1].S \cup \{$ATOMICCONSTRAINT($v$)$\}$;
    $C[i].F \leftarrow C[i-1].F$;
    $C[i].A \leftarrow C[i].A - \{v\}$;
    **if** $\neg$CHECK($C[i].S$)
    **then go to** *fail* **fi**;
    $C[i].F \leftarrow C[i].F \cup$ ORDAUGHTERSOFAND($v$);
    **go to** *loop* **end**
```

Figure 3.1: DEPTH-FIRST SEARCH, a depth-first search procedure for AND/OR-trees which is analogous to the search methodology typically used for PROLOG.

a stack $C[i]$. Each choice-point has three fields, $\langle S, F, A\rangle$. The S field maintains the set of atomic constraints contained in the AND-subtree represented by that choice-point. Typical PROLOG implementations represent S by a combination of the unification environment, and the trail, which is necessary to restore a previous unification environment. The F field maintains the set of $\vee$-vertices that must be expanded and included in the AND-subtree for it to become complete. PROLOG implementations represent F, the list of goals remaining to be satisfied, via a stack of saved program counters pointing to code which must be executed to satisfy those goals. The A field maintains the set of $\wedge$-vertices which are the remaining alternative daughters of the $\vee$-vertex corresponding to this choice-point. This field too, is typically represented by a program counter pointing to the code for the remaining alternatives.

The DEPTH-FIRST SEARCH algorithm is underspecified. Line 9 nondeterministically chooses which $\vee$-vertex to add to the current AND-subtree. In PROLOG this corresponds to the choice of which goal to pursue next. Line 16 nondeterministically chooses which daughter $\wedge$-vertex to select from the remaining alternative daughters of the $\vee$-vertex associated with some choice-point. In PROLOG this corresponds to the choice of which clause to use to attempt to satisfy a goal. The fixed left-to-right pursuit of goals, and the fixed top-to-bottom pursuit of clauses in PROLOG, can be accomplished by making choices in the appropriate fixed order. Alternatively, the techniques of search rearrangement[96] can be incorporated into DEPTH-FIRST SEARCH via more complex choice functions.

3.3 The Culprit Pointer Method For AND/OR-Trees

DEPTH-FIRST SEARCH can be modified to incorporate the CULPRIT POINTER METHOD for performing selective backtracking on non-selectively-nested failures. The essence of this modification is captured by a variation on the soundness theorem given for constraint satisfaction problems. Before stating this soundness theorem, several definitions are needed.

Definition: Let X and Y be two AND-subtrees of the same AND/OR-tree T. Furthermore, let d be a set of atomic vertices from T. The notation $X+Y$ denotes the tree which contains all of the vertices of X augmented with vertices from Y as follows. For every $\wedge$-vertex u in $X+Y$, $X+Y$ also contains all of the daughters of u which are contained in Y. For every $\vee$-vertex u in $X+Y$, if Y contains a daughter of u and X does not, then $X+Y$ contains the daughter of u in Y. It should be obvious that $X+Y$ must always denote an AND-subtree of T. If the vertices of X are a subset of the vertices of Y then we say that Y is an *extension* of X. If no $\wedge$-vertex which is an ancestor of a vertex in d is a sibling of a vertex in X then we say that X ic *compatible* with d. Likewise, if no $\wedge$-vertices in X and Y are siblings then we say that X is *compatible* with Y. If X is compatible with d, then $X \cap d$ denotes the tree containing all of the vertices in d as well as all of the ancestors of vertices in d so long as they are also in X. It should be obvious that $X \cap d$ must always denote an AND-subtree of T. If X is compatible with Y then $X \cup Y$ denotes the tree which contains all of the vertices of X combined with all of the vertices of Y. It should be obvious that $X \cup Y$ must always denote an AND-subtree of T. Finally, if D is a nonempty set whose members are all sets of atomic vertices from T, and X is compatible with each $d \in D$, and every complete extension of X is an extension of some $d \in D$, then we say that D *spans* X.

Note that since we require that every $\vee$-vertex in an AND/OR-tree to have at least one daughter, every AND-subtree has a complete extension. If the unraveling of a PROLOG program produces an AND/OR-tree with leaf $\vee$-vertices, these vertices must be removed along with their parent $\wedge$-vertices. In the following we take some additional liberty in notation. When we state that $\alpha \subseteq \beta$, where either α or β or both are trees, we mean that the vertices in α are a subset of the vertices in β. Additionally, if X and Y are trees, the notation $X-Y$ denotes the set of vertices which are in X but not in Y.

To prove the soundness theorem we begin with the following lemma.

Lemma 2 *If X and Z are two AND-subtrees of an AND/OR-tree T, and d' is a set of atomic vertices of T, and $X \cap d' \subseteq Z$, and $d' \subseteq X + Z$ then $d' \subseteq Z$*

> *Proof:* Consider an arbitrary member e of d'. Since $d' \subseteq X + Z$ then either $e \in X$ or $e \in Z$. If $e \in Z$ the conclusion holds. If $e \in X$ then since $e \in d'$ and $X \cap d' \subseteq Z$ then $e \in Z$. $\square$

Now we can prove the soundness theorem.

Theorem 3 *If T is an AND/OR-tree, and X is an AND-subtree of T, and D is a set whose members are all sets of atomic vertices of T such that D spans X, and*

$$Y = \bigcup_{d \in D} (X \cap d)$$

then D spans Y.

> *Proof:* Consider an arbitrary complete extension Z of Y. Such a complete extension must exist since we require all $\vee$-vertices to have at least one daughter. Let Z' be any arbitrary complete extension of $X+Z$. Likewise, Z' must exist as well. Since Z' is a complete extension of X and D spans X there must exist some $d' \in D$ such that $d' \subseteq Z'$. We will first show that the same $d' \subseteq X+Z$ and then that $d' \subseteq Z$. To show that $d' \subseteq X+Z$ notice that since $Y \subseteq Z$ it follows that $X - Z \subseteq X - Y$ and therefore $\mathit{descendants}(X - Z) \subseteq \mathit{descendants}(X - Y)$. Furthermore, since Z is complete, any vertex added to $X+Z$ to form Z' must be a descendant of some vertex in $X - Z$ and therefore of some vertex in $X - Y$ as well. To show that $d' \subseteq X + Z$ it suffices to show that d' does not contains vertices which are descendants of $X - Y$. Since $X \cap d' \subseteq Y$ it follows that $X - Y \subseteq X - (X \cap d')$. Thus it suffices to show that any ancestor of any vertex in d' that is in X is also in $X \cap d'$. This is true by the definition of $X \cap d'$. Having shown that $d' \subseteq X + Z$ we now will show that $d' \subseteq Z$. Since $Y \subseteq Z$ and for all $d \in D$, $X \cap d \subseteq Y$ it follows that $X \cap d' \subseteq Z$. Therefore, by lemma 2, $d' \subseteq Z$. $\square$

This soundness theorem can be used to maintain culprit pointers while searching AND/OR-trees in a fashion similar to their use in searching constraint satisfaction problems. Each choice-point is supplemented with two additional fields, a field K containing the culprit pointer for that choice-point, and a field C containing the atomic constraint labeling the lone atomic daughter of the current alternative chosen for the $\vee$-vertex represented by that choice-point. At each CHECK failure, the culprit pointer for each choice-point j is updated to the maximum of its previous value and the smallest k such that the set

$$\{C[0].C, \ldots, C[k].C, C[j].C, \ldots, C[i].C\}$$

is inconsistent. Each such inconsistent set of atomic constraints corresponds to some $d \in D$. When the choice-point j exhausts, the choice-point stack containing choices-points 0 through j corresponds to the AND-subtree X. The collection of all inconsistent sets discovered underneath choice-point j, i.e. the set D, spans X. Furthermore, the initial segment of the choice-point stack containing choice-points 0 through $C[j].K$ corresponds to the AND-subtree Y. Forming $X \cap d$ corresponds to removing from the set

$$\{C[0].C, \ldots, C[k].C, C[j].C, \ldots, C[i].C\}$$

the constraints $\{C[j].C, \ldots, C[i].C\}$ leaving only the set $\{C[0].C, \ldots, C[k].C\}$ which can be characterized by the single number k. Forming

$$Y = \bigcup_{d \in D} (X \cap d)$$

corresponds to taking the maximum of all such k found. Thus the culprit pointer for choice-point j indicates an AND-subtree Y associated with the AND-subtree X corresponding the the choice-point j itself. By the soundness theorem, since D spans X, D also spans Y. But since every element in D is inconsistent, and CHECK is monotonic, Y can have no complete consistent extension and thus is it safe to backtrack to the choice-point indicated by the culprit pointer for choice-point j when choice-point j exhausts.

As when solving constraint satisfaction problems, the CULPRIT POINTER METHOD is sound only when two conditions are enforced. First, it is safe to backtrack selectively when choice-point j exhausts only when no solution has been found underneath that choice-point. The is necessary to enforce the condition that every element of D be inconsistent. Second, it is safe to backtrack selectively when choice-point j exhausts only when no selective backtrack was taken underneath that choice-point. This is necessary to enforce the condition that D span X. Therefore, both when a solution is found, and when a selective backtrack is taken, it is necessary to reset all of the culprit pointers to point to the immediately preceding choice-point to prevent unsound selective backtracks.

There is one additional difference between the implementation of the CULPRIT POINTER METHOD for searching constraint satisfaction problems and that for searching AND/OR-trees. When searching constraint satisfaction problems, the culprit pointer for a choice-point is initialized to -1 when that choice-point is created. When searching AND/OR-trees however, the culprit pointer for a choice-point associate with an $\vee$-vertex u must be initialized to point to the choice-point associated with the grandparent $\vee$-vertex of u. If u does not have a grandparent then the culprit pointer is initialized to -1. The reason for this difference is that all of the inconsistent constraint sets d examined by DEPTH-FIRST SEARCH all contain descendants of the lone leaf $\vee$-vertex in X, the exhausting AND-subtree. Therefore, the tree $X \cap d$ must contain this $\vee$-vertex u as well as the choice-point which selects the parent $\wedge$-vertex of u as the alternative for the $\vee$-vertex which is the grandparent of u.

Figure 3.2 gives the CULPRIT POINTER DEPTH-FIRST SEARCH algorithm, a modification of the DEPTH-FIRST SEARCH algorithm from figure 3.1 which incorporates the CULPRIT POINTER METHOD. The code modifications needed to add the CULPRIT POINTER METHOD into DEPTH-FIRST SEARCH are summarized below. First of all, the F field of a choice-point is modified to contain a set of pairs rather than a set of vertices. Each pair comprises an $\vee$-vertex along with the index of its grandparent's choice-point. This index is used to initialize the culprit pointer for the choice-point which will be created for that $\vee$-vertex. Lines 5 through 7, lines 39 through 40 and line 15 contain the necessary modifications to maintain pairs rather than vertices in the F field of choice-points. Lines 9 and 14 initialize the culprit pointers to the index of their grandparent choice-point when a choice-point is created. Likewise, line 27 initializes the C field of a choice-point upon creation. Line 12 resets the culprit pointers when a solution is found while line 20 resets the culprit pointers when a selective backtrack takes place. Line 21 actually performs the selective backtrack. Finally, lines 29 through 37 are responsible for updating the culprit pointers during CHECK failures. Note that, like DEPTH-FIRST SEARCH, the CULPRIT POINTER DEPTH-FIRST SEARCH algorithm is underspecified. This demonstrates how the techniques of rearrangement search[96] are orthogonal to, and can be combined with, the CULPRIT POINTER METHOD.

3.4 Complexity Analysis

The complexity of the CULPRIT POINTER METHOD for searching AND/OR-trees is the same as it is for constraint satisfaction problems: $O(n)$ space to store the additional fields on the choice-point stack, and $O(n^2)$ time per CHECK failure to update the culprit pointers, where n is the depth of the choice-point stack. Likewise, it is only necessary to check the consistency of the sets

$$\{C[0].C, \ldots, C[k].C, C[j].C, \ldots, C[i].C\}$$

```
    **procedure** Culprit Pointer Depth-First Search
        $i \leftarrow 0$;
        $C \leftarrow \mathit{NewVector}$;
        $C[0].S \leftarrow \{\}$;
        $C[0].F \leftarrow \{\}$;
        **for** $w \in$ OrDaughtersOfAnd(Root())
        **do** $C[0].F \leftarrow C[0].F \cup \{\langle -1, w \rangle\}$ **od**;
        $C[0].A \leftarrow \{\}$;
        $C[0].K \leftarrow -1$;
*loop* :
        **if** $C[i].F = \{\}$
        **then** Solution($C[i].S$);
            **for** $j$ **from** $0$ **to** $i$ **do** $C[j].K \leftarrow j - 1$ **od**;
            **go to** *fail* **fi**;
        $\langle C[i+1].K, u \rangle \leftarrow \mathit{choose}(C[i].F)$;
        $C[i].F \leftarrow C[i].F - \{\langle C[i+1].K, u \rangle\}$;
        $i \leftarrow i + 1$;
        $C[i].A \leftarrow$ AndDaughtersOfOr($u$);
*fail* :
        **if** $C[i].A = \{\}$
        **then if** $C[i].K \neq i - 1$
            **then for** $j$ **from** $0$ **to** $C[i].K$ **do** $C[j].K \leftarrow j - 1$ **od fi**;
            $i \leftarrow C[i].K$;
            **if** $i < 0$ **then return else go to** *fail* **fi fi**;
        $v \leftarrow \mathit{choose}(C[i].A)$;
        $C[i].S \leftarrow C[i-1].S \cup \{$AtomicConstraint($v$)$\}$;
        $C[i].F \leftarrow C[i-1].F$;
        $C[i].A \leftarrow C[i].A - \{v\}$;
        $C[i].C \leftarrow$ AtomicConstraint($v$);
        **if** $\neg$Check($C[i].S$)
        **then** $S' \leftarrow \{\}$;
            **for** $j$ **from** $i$ **to** $1$ **by** $-1$
            **do** $S' \leftarrow S' \cup \{C[i].C\}$;
                $S'' \leftarrow S'$;
                **for** $k$ **from** $1$ **to** $C[j].K$
                **do** $S'' \leftarrow S'' \cup \{C[k].C\}$ **od**;
                **while** Check($S''$)
                **do** $C[j].K \leftarrow C[j].K + 1$;
                    $S'' \leftarrow S'' \cup \{C[C[j].K].C\}$ **od od**;
            **go to** *fail* **fi**;
        **for** $w \in$ OrDaughtersOfAnd($v$)
        **do** $C[i].F \leftarrow C[i].F \cup \{\langle i, w \rangle\}$ **od**;
        **go to** *loop* **end**
```

Figure 3.2: Culprit Pointer Depth-First Search, a modification of the Depth-First Search algorithm from figure 3.1 which incorporates the Culprit Pointer Method.

for $k > C[j].K$. In the case of Prolog, where Check is the unifier, this is not particularly helpful. This is because checking the consistency of some set, entails calling the unifier on each element in that set. This entails work which is equivalent to checking the consistency of each set

$$\{C[0].C, \ldots, C[k].C, C[j].C, \ldots, C[i].C\}$$

for each k. This highlights an undesirable property of the Culprit Pointer Method when applied to Prolog: As culprit pointers increase, the effort required to maintain them increases although their pruning ability decreases. Furthermore, there is a discontinuity: As soon as a culprit pointer is increased to reach its limit, no further effort need be expended to maintain it. Accordingly, there is great need for heuristics to determine when to punt and reset a culprit pointer.

Chapter 4

Implementation as an Extended Warren Abstract Machine

The previous chapter showed how the Culprit Pointer Method can be used to perform selective backtracking at failures which are not selectively-nested during Prolog execution. One of the advantages of this method over other methods for performing selective backtracking is that it lends itself to efficient implementation. Most efficient implementations of Prolog are based on the architecture proposed by Warren[90], commonly known as the Warren Abstract Machine or WAM. This chapter demonstrates how the WAM architecture can be modified, with relatively little overhead, to incorporate the Culprit Pointer Method.

Ideally, this chapter should have presented the necessary modifications in terms of the actual Warren Abstract Machine. As I did not have access to an existing WAM implementation to modify, and was more fluent with Lisp, I chose instead to first create a simple compiler which translated Prolog into Lisp, and then instead, modify this compiler to incorporate the Culprit Pointer Method. Rather than risk making errors while presenting the modifications in terms of the WAM, whose correctness has not been verified by implementation, I will instead present the Lisp implementation which I know has run successfully.

The strategy for compiling Prolog into Lisp is similar to that used by LM-Prolog[48]. In order to more strongly support the claim that the Culprit Pointer Method can be incorporated into a low overhead WAM implementation, the target Lisp code generated for Prolog programs attempts to capture most of the essential features of the WAM architecture. For this reason, the compilation strategy discussed here will be called the LispWAM and the associated compiler will be called the LispWAM compiler. In order to mimic the WAM architecture as much as possible in the underlying architecture of the Lisp implementation, it was necessary to sacrifice portability, and make the compiler generate some highly implementation specific code. The integration of the Culprit Pointer Method into the LispWAM depends not only on style of Lisp code generated by the LispWAM compiler for Prolog input, but on the architecture of the underlying Lisp implementation as well. In this case, the underlying Lisp architecture is the Symbolics 3600 Genera 7.2 architecture which is henceforth simply referred to as the Symbolics architecture. While, the details of the compiler discussed in this section are both particular to the Symbolics architecture as well as somewhat involved and boring, I present them in full as they typify the kinds of modifications which can be applied to other implementations including the WAM architecture itself.

One implementation dependent detail is the stack organization. The LispWAM tries to mimic the WAM architecture by representing choice-points as frames on the underlying Lisp function call stack. Lisp functions generated by the Prolog compiler which produce choice-points (i.e. frames on the function call stack) are called *choice-point functions*. Three types of choice-point functions are created. First, a *Clause function* is generated for each non-fact clause in the PROLOG program, which when called, produces the choice-point associated with the first goal in that clause. Inside each clause function, a *goal continuation* is generated for each subsequent goal which when called, produces a choice-point for that goal. Finally, the top level query generates a *query function* which when called, produces a choice-point for the first goal in the query. Choice-points for subsequent goals in a query are produced by goal continuations as they are for clause functions.

The implementation of the CULPRIT POINTER METHOD relies on the correspondence of stack frames to choice-points. In particular, it assumes that every stack frame between the one created be the query function, and the top of the stack, is a choice-point. No intermediate non-choice-point stack frames can be introduced and no choice-point stack frames can be elided say by β-substitution. Furthermore, the implementation is sensitive to the organization of choice-point stack frames. Certain choice-point slots are represented as parameters in the choice-point stack frame and must reside at fixed known offsets to be accessible be routines which maintain the culprit pointers. Great care was taken in the implementation to enforce these constraints to mimic the operation of the WAM as closely as possible.

For purposes of comparison, the first section of this chapter illustrates how PROLOG programs can be compiled into LISP using the LISPWAM architecture, without the additional encumbrances of the CULPRIT POINTER METHOD. The second section then illuminates the changes to this compiler which are necessary for supporting the CULPRIT POINTER METHOD.

4.1 Compiling Prolog into Lisp

As the PROLOG compiler is implemented on top of LISP, PROLOG programs will be represented as LISP S-expressions in the conventional manner. PROLOG constants will be represented as LISP atoms. PROLOG variables will be represented as LISP symbols whose print-name begins with "?". PROLOG terms will be represented as LISP lists where the first element of the list is the functor and the subsequent elements are the arguments. PROLOG clauses will be represented again as LISP lists where the first element of the list is the clause head and the remaining elements are the goals which constitute the clause body.

Classifying Variables The WAM stores bindings for variables appearing in a clause in the stack frame corresponding to the choice-point which invoked that clause. Analogously, the LISPWAM represents the variables appearing in a clause as parameters to the choice-point function generated for that clause. These parameters then create slots in the choice-point stack frame created when the clause function is called. These slots can then store the bindings of the variables as produced via unification.

To mimic the WAM, unification is compiled away as much as possible into simple operations of parameter passing, type checking, structuring, and destructuring. This requires classifying variables so that different code can be generated for different types of variable and their use. For a given clause, the set of all distinct variables appearing in that clause is called its set of *clause variables*. Variables appearing in the head are called *head variables* while variables appearing in the body are called *body variables*. Clause variables can be classified into one of three distinct types:

1. Clause variables that appear in the body but not in the head are called *logic variables*.

2. Clause variables that appear in the head but not in the body are called *template variables*.

3. Clause variables that appear both in the head and in the body are called *argument variables*.

Variables that appear in the body of a query are called *query variables*. The first occurrence in the head (from left to right) of given head variable is called its *primary occurrence*. Any further occurrences are termed *secondary occurrences*.

Compiling Clauses Each non-fact clause in a Prolog program is compiled into a separate Lisp function called a *clause function*. Clauses which are facts do not generate clause functions. The following is a template for such a clause function:

```
(defun CLAUSE-i (next-goal ⟨argument variables⟩ &aux ⟨logic variables⟩)
  (declare (special *trail*))
  ⟨code for initializing logic variables to be unbound⟩
  (let* ((next-goal
           (lambda () (declare (sys:downward-function)) ⟨code for last goal⟩))
          ⋮
          (next-goal
           (lambda () (declare (sys:downward-function)) ⟨code for third goal⟩))
          (next-goal
           (lambda () (declare (sys:downward-function)) ⟨code for second goal⟩)))
    ⟨code for first goal⟩))
```

The clauses are numbered to given each clause a unique index. The function name, `clause-`i is generated to include that index i.

Calling a clause function creates a choice-point stack frame with slots for both the argument variables and the logic variables in the clause. The bindings for the argument variables are passed to the clause function by the code generated for the goal which selected this clause as an alternative for satisfying the goal. Code generated for the goal, actually performs the unification of the goal with the clause head and then passes the result of this unification to the clause function through the argument variables. In order to correctly pass the results of the unification to the right argument variables in the clause function, the argument variables must be listed in some canonical order. Note that the template variables, those clause variables which appear only in the head, need not be allocated slots in the choice-point frame created by the clause function, as they can never be accessed by its descendants. Furthermore, note that the logic variables, those clause variables which appear only in the body, are not be passed as parameters to the clause function, but rather are auxiliary variables initialized within that function, as they cannot be bound by the unification of the parent goal with the clause head.

In the LispWAM, goal failure and backtracking correspond to a return of a choice-point function which pops its associated choice-point off the stack. On the other hand, when one goal of a clause or query succeeds, and a subsequent goal must be pursued, the return mechanism cannot be used since the choice-point corresponding to the first goal, along with its variable bindings, must be preserved while executing all subsequent goals. This mechanism is handled by a convention whereby each choice-point function is passed a continuation as its first argument. This, argument is always called `next-goal`. While a choice-point function fails by returning, its succeeds by calling the continuation in `next-goal`.

The `next-goal` parameter to the clause-function is the continuation to call when the entire clause, i.e. its last goal, succeeds. The code generated for the last goal in a clause sits in a lexical context where that `next-goal` parameter is visible. The code generated for each goal except the last sits in a lexical context where a local variable named `next-goal` is successively bound to a continuation containing the code for pursuing the next goal (and thus all subsequent goals) of the clause. These continuations for each goal in a clause other than the first, are called *goal continuations*. When the code for some goal must call a clause function to pursue the body of a clause whose head matches the goal, it passes the the current lexical value of `next-goal` as the first argument to the clause function so that when the clause function succeeds, the goal subsequent to the current goal is pursued. When a goal matches a fact, no clause function is called since there is no body to satisfy. In this situation the code for a goal simply calls the continuation for the next goal in the clause. If the last goal in a clause matches a fact then the code generated calls the continuation passed as the `next-goal` to the clause function. The top level continuation passed as the `next-goal` to the query function is called the *solution handler* which will typically print the bindings of top-level query variables and backtrack by returning if another solution is desired.

At the beginning of the function, all of the logic variables are initialized to be unbound. An unbound variable is represented as a locative which points to itself. Each logic variable is initialized by a copy of the following code fragment:

```
    (setf ⟨logic variable⟩ (locf ⟨logic variable⟩))
```

Argument variables receive their value from the goal code generated as part of either a query function or another clause function, which calls this clause function.

PROLOG compilers for the Warren Abstract Machine typically perform a number of optimizations which are not incorporated into the code fragment illustrated here. I mention these optimizations to stress that they do not interfere with the CULPRIT POINTER METHOD. First, argument and logic variables are popped off the stack when there are no further alternatives which can reference them. A clever LISP compiler can accomplish the same optimization on the code generated by the LISPWAM compiler by simply generating code which reclaimed space used by function parameters and local variables after they can no longer be accessed. Second, most PROLOG compilers delay the allocation of storage for a logic variable in a clause until it is needed by the first goal which references it. A somewhat different compilation strategy could incorporate this optimization into the LISPWAM. Finally, the WAM can allocate several logic variables and make them unbound with a single instruction. This optimization has no counterpart in the LISPWAM but would not interfere with the incorporation of the CULPRIT POINTER METHOD to be discussed.

The LISP code generated by the LISPWAM compiler makes use of a number of auxiliary functions. In most cases, calls to these functions can be open-coded without affecting the correctness of the implementation. These cases are noted by a comment on the first line of such functions. Sometimes however, the auxiliary function must access the choice-point stack and relies on the fact that it is *called* by a choice-point function and therefore assumes that the most recent choice-point is the stack frame *previous* to the current stack frame. This assumption would no longer be true if the call to this auxiliary function were open-coded. The two functions which exhibit this property are so indicated by a comment on the their first line. They can however, be modified slightly to revise the assumption so that the most recent choice-point is the *current* stack frame rather than the one previous to it, and thus be suitable for open-coding.

Compiling Goals Each goal in a clause or query can potentially match the head of several clauses. Each potentially matching head, which isn't ruled out by a compile-time unification failure, is called an *alternative*. For each goal in a clause or query, the following code template is generated and incorporated into the clause or query function:

```
(let ((trail *trail*))
   (sys:with-data-stack
    ⟨code for first alternative⟩
    (unwind trail))
   (sys:with-data-stack
    ⟨code for second alternative⟩
    (unwind trail))
   ⋮
   (sys:with-data-stack
    ⟨code for next to last alternative⟩
    (unwind trail)))
 ⟨code for last alternative⟩
 (values)                                    ;does note return any values
```

This code fragment tries in succession, each of the alternative clauses for satisfying a given goal. An alternative fails by simply returning, so that the next alternative is tried in succession. Note that the code fragment ends with the form `(values)`. This is because goal continuations and clause function are choice-point functions which return only on failure, when no value is needed.

One of the optimizations performed by the WAM architecture is that storage allocated during an attempt to satisfy a goal is reclaimed without garbage collection when that goal fails. This is handled in the LispWAM by allocating all storage on the data stack. Each alternative, except the last, is wrapped in a `sys:with-data-stack`. This sets up a data stack frame for allocating any storage created during the unifications performed during that alternative. This storage allocated in this data stack frame is automatically reclaimed, without any garbage collection, when the alternative fails and returns, exiting the data stack frame created by `sys:with-data-stack`. One disadvantage of this scheme is that there is no capability for doing garbage collection on the data stack to reclaim inaccessible space prior to backtracking.

The Trail Before attempting the first alternative, the state of the global variable `*trail*` is recorded. This variable maintains a list of all logic variables which have been bound until this point. As mentioned previously, an unbound logic variable is a Lisp variable bound to a locative pointing to itself. (We will see later that unification also can produce a cons cell whose slots are logic variables; locatives which also point to themselves.) During unification, a logic variable is bound by the following code fragment which records the fact that a variable has been bound on the trail:

```
(push ⟨logic variable⟩ *trail*)
 (setf (location-contents ⟨logic variable⟩) ⟨value⟩)
```

When an alternative fails, the trail is unwound to its previous state by unbinding all of the logic variables bound during the execution of the alternative. This unwinding is accomplished by the following function:

```
(defun UNWIND (trail)                        ;can be open-coded
  (declare (special *trail*))
  (loop until (eq *trail* trail)
        for x = (pop *trail*)
        do (setf (location-contents x) x))
  (values))                                  ;does not return any values
```

Note that the trail need not be explicitly unwound after the last failure as when the code for the last alternative returns, the entire fragment and thus the entire clause or query function returns, and the trail will be unwound by the code for the parent alternative. Also note that the last alternative need not be wrapped in a `sys:with-data-stack` for analogous reasons. Additionally, the lexical variable `trail`, which stores the state of of the global variable `*trail*` can be discarded prior to the last alternative. These three optimizations correspond to some of the tail recursion merging performed by the WAM architecture on the choice-point stack.

Compiling Alternatives For each alternative clause j which can match a given goal in either a given clause i or in the query, one of the following two code fragments is generated and incorporated into either the clause function for clause i or the query function. In the simple case, if the matching clause j is a fact then the following code fragment is generated:

```
(tagbody
    ⟨code for unification⟩
    (funcall next-goal)
 fail)
```

This first performs the unification between the goal in clause i (or the query) and the head of clause j. Since clause j is a fact, the alternative proceeds by simply calling the continuation `next-goal` in the context of that unification. If however, the matching clause j is not a fact, then the following code fragment is generated:

```
(tagbody
    ⟨code for unification⟩
    (clause-j next-goal ⟨arguments⟩)
 fail)
```

Here, after the unification is performed, the clause function for clause j is called to pursue the goals in its body. This clause function is passed the continuation `next-goal` to be called when all of the direct and indirect goals in clause j are satisfied. In each case, the fragment commences with code for performing the unification between the goal in clause i (or the query) and the head of the current alternative, clause j. This unification code does two things. First, it checks whether the unification is possible. If not, a branch is taken to the tag `fail` and the alternative returns. Then, if the alternative is a non-fact clause, the requisite values of the argument variables of the target clause j are computed from the values of the body variables of the source clause i. These values are passed onto the clause function for clause j as arguments.

Code Generated for Unification The code generated for unification consists of a sequence of macro-instructions. This code makes use of a set of auxiliary registers `r1`, ..., `rn` to avoid recomputing common subexpressions. Each macro-instruction generates a binding environment for some registers. The remaining macro-instructions in the unification code for that alternative, as well as the call to either the target clause function or the goal continuation, are done within this context.

Unifying a Goal Constant with a Head Variable When unifying a goal constant with a head variable, the macro-instruction

```
(constant ⟨register⟩ ⟨goal constant⟩)
```

is generated. This macro-instruction expands into the following code:

```
(let (⟨register⟩ '⟨goal constant⟩)
   ⟨remaining code for alternative⟩)
```

The contents of ⟨*register*⟩ are then passed as an argument to the target clause function.

Unifying a Goal Variable with a Head Variable When unifying a goal variable with a head variable, the macro-instruction

```
(parameter ⟨register⟩ ⟨goal variable⟩)
```

is generated. This macro-instruction expands into the following code:

```
(let ((⟨register⟩ (dereference ⟨goal variable⟩))
   ⟨remaining code for alternative⟩)
```

This code dereferences the goal variable and leaves its value in a register. The contents of this register are then passed as an argument to the target clause function.

Dereferencing is accomplished by the following function:

```
(defun DEREFERENCE (x)             ;can be open-coded
  (loop for z = x then (location-contents z)
        until (or (not (locativep z)) (eq z (location-contents z)))
        finally (return z)))
```

Unifying a Goal Variable with a Head Variable When unifying a goal variable with a head constant, the macro-instruction sequence

```
(parameter ⟨register⟩ ⟨goal variable⟩)
 (insure-constant ⟨register⟩ ⟨head constant⟩)
```

is generated. The `insure-constant` macro-instruction expands into the following code:

```
(cond ((locativep ⟨register⟩)
        (push ⟨register⟩ *trail*)
        (setf (location-contents ⟨register⟩) '⟨head constant⟩))
       ((not (eql ⟨register⟩ '⟨head constant⟩)) (go fail)))
```

The `insure-constant` macro-instruction first checks to see whether the register is bound and has the desired value. If it does then the unification succeeds. If the register is unbound then the register is trailed and bound to the value and the unification succeeds. If the register is bound to an undesired value the unification fails.

Unifying a Goal Constant or Cons with a Head Constant or Cons If a goal constant is unified with a head constant the unification either trivially succeeds or fails. If it fails then that clause is not considered as an alternative for the goal being compiled. This is a form of static compile time clause indexing. If it succeeds then no code is generated for this unification. Additionally, the unification of a goal cons with a head constant or of a goal constant with a head cons always fails at compile time as well and the clause is not considered as an alternative for the goal being compiled.

When unifying a goal cons with a head cons, two subsequences of unification macro-instructions are generated—one to unify the corresponding cars and and one to unify the corresponding cdrs. If either one fails at compile time, the whole unification fails at compile time as well, and the alternative is not considered. Otherwise, the two unification subsequences are concatenated to form the complete sequence.

Structuring: Unifying a Goal Cons with a Head Variable When unifying a goal cons with a head variable, two subsequences are first generated to get the dereferenced values of both the car of the goal cons, and its cdr, into registers. If the car of the goal cons is a constant then macro-instruction

```
(constant ⟨car register⟩ ⟨car constant⟩)
```

is generated. If it is a variable then the macro-instruction

```
(parameter ⟨car register⟩ ⟨car variable⟩)
```

is generated. If it is itself is a cons then this procedure is applied recursively. The same procedure is likewise applied to the cdr of the goal cons. The following macro-instruction is appended to the concatenation of the two code subsequences thus generated:

```
(cons ⟨cons register⟩ ⟨car register⟩ ⟨cdr register⟩)
```

This macro-instruction expands into the following code:

```
(let ((⟨cons register⟩ (cons-stack ⟨car register⟩ ⟨cdr register⟩)))
  ⟨remaining code for alternative⟩)
```

This conses the two values together inside the frame created by the enclosing `sys:with-data-stack` so that the storage generated is reclaimed when the frame is exited upon backtracking.

Remember that Prolog terms are represented in the LispWAM as Lisp S-expressions which are in turn composed of Lisp cons cells. During the execution of Prolog programs on the LispWAM, the Prolog terms created by unification are constructed using an alternate representation for cons cells. Execution time Prolog cons cells are represented as two element vectors rather than directly as Lisp cons cells. They are generated by the following function.

```
(defun CONS-STACK (car cdr)          ;can be open-coded
  (let ((cons (sys:make-stack-array 2)))
    (setf (aref cons 0) car)
    (setf (aref cons 1) cdr)
    cons))
```

This is done for two reasons. First, the Symbolics architecture does not have the ability to allocate Lisp cons cells on the data stack and thus does not have a `cons-stack` instruction. It can only allocate arrays on the data stack using the primitive `sys:make-stack-array`. Second, the implementation needs to distinguish between Prolog cons cells which represent structures and locatives which represent (possibly unbound) variables. In the Symbolics architecture, the form `(locf (cdr x))`, unlike `locf` of anything else, returns `(cdr x)` itself, rather than an object whose data type is locative. Although this correctly behaves like a locative as far as the function `location-contents` is concerned, it is indistinguishable from a Lisp cons cell and does not behave like a locative as far as `locativep` is concerned. In particular, the implementation may need to construct a Prolog cons cell whose cdr contained an unbound logic variable. If Prolog cons cells were directly represented as Lisp cons cells, the code sequence

```
(setf (cdr x) (locf (cdr x)))
```

which attempts to make `(cdr x)` be unbound does not result in `(cdr x)` being bound to something of data type locative. A subsequent call to `(locativep (cdr x))` will return `nil` which makes it impossible to determine whether `(cdr x)` is a variable or a Prolog cons cell. These problems do not arise when representing Prolog cons cells as two element vectors.

Destructuring: Unifying a Goal Variable with a Head Cons When unifying a goal variable with a head cons, destructuring must take place. This is accomplished by the following macro-instruction sequence:

```
(parameter ⟨cons register⟩ ⟨goal variable⟩)
 (insure-cons ⟨cons register⟩)
 (car ⟨car register⟩ ⟨cons register⟩)
 (cdr ⟨cdr register⟩ ⟨cons register⟩)
```

The `insure-cons` macro-instruction checks to see whether its argument is indeed a cons. If it is, then it succeeds. If it is unbound, then it is trailed and bound to a new PROLOG cons cell, both of whose slots are bound to unbound logic variables, and the unification succeeds. Otherwise, the unification fails. This is accomplished by the following code:

```
(cond ((locativep ⟨cons register⟩)
       (let ((z (fresh-cons-stack)))
         (setf (location-contents ⟨cons register⟩) z)
         (push ⟨cons register⟩ *trail*)
         (setf ⟨cons register⟩ z)))
      ((not (arrayp ⟨cons register⟩)) (go fail)))
```

The routine `fresh-cons-stack` allocates a new PROLOG cons cell on the data stack whose car and cdr are set to unbound logic variables.

```
(defun FRESH-CONS-STACK ()          ;can be open-coded
  (let ((cons (sys:make-stack-array 2)))
    (setf (aref cons 0) (locf (aref cons 0)))
    (setf (aref cons 1) (locf (aref cons 1)))
    cons))
```

The `car` macro-instruction places the dereferenced value of the car of the source register into the destination register by the following code:

```
(let ((⟨car register⟩ (dereference (car-array ⟨cons register⟩)))
   ⟨remaining code for alternative⟩)
```

Since PROLOG cons cells are represented as two element vectors, the car of a PROLOG cons cell is accessed via the following function:

```
(defun CAR-ARRAY (cons)             ;can be open-coded
  (aref cons 0))
```

Analogous code is generated for the `cdr` macro-instruction. Note that if the results of either the `car` or the `cdr` macro-instruction are not needed, as would be the case when unifying with a template variable, then the macro-instruction is eliminated from the unification code sequence.

The General Unifier Finally, when a variable appears more then once in a head, each secondary occurrence is renamed. At the end of the unification code sequence, the following macro-instruction is generated for each pair of primary and secondary occurrences of variables:

```
(insure-equal ⟨primary register⟩ ⟨secondary register⟩)
```

This macro-instruction calls the unifier on the values in the two registers. If the unification fails, a branch to the tag `fail` is taken. Otherwise, the values are unified and the logic variables which become bound in the process are trailed. The code generated is as follows:

```
(if (unify ⟨primary register⟩ ⟨secondary register⟩) (go fail))
```

where the unifier routine is:

```
(defun UNIFY (x y)
  (declare (special *trail*))
  (cond ((eql x y) nil)
        ((locativep x)
         (cond ((locativep y)
                (setf (location-contents x) y)
                (push x *trail*))
               (t (setf (location-contents x) y)
                  (push x *trail*)))
         nil)
        ((locativep y)
         (setf (location-contents y) x)
         (push y *trail*)
         nil)
        ((and (arrayp x) (arrayp y))
         (or (unify-array (dereference (car-array x))
                          (dereference (car-array y)))
             (unify-array (dereference (cdr-array x))
                          (dereference (cdr-array y)))))
        (t t)))
```

Note that just as in the WAM architecture, the general unifier is rarely called. Most unifications correspond to simple structuring and destructuring operations which are open-coded. Also note that other than trailing operations, the general unifier does not allocate any storage. During unification, storage is allocated only by the `cons` and `insure-cons` macro-instructions which are open-coded.

The following example summarizes how unification code is compiled. If the goal term is:

```
(p (f ?x a) ?u (g ?w) ?z)
```

and the head term is:

```
(p (f b ?y) ?v ?v (h ?v))
```

then the following macro-instruction sequence is generated:

```
(parameter r1 ?x)
 (insure-constant r1 b)
 (constant r2 a)
 (parameter r3 ?u)
 (constant r4 g)
 (parameter r5 ?w)
 (constant r6 nil)
 (cons r7 r5 r6)
 (cons r8 r4 r7)
 (parameter r9 ?z)
 (insure-cons r9)
 (car r10 r9)
 (insure-constant r10 h)
 (cdr r11 r9)
 (insure-cons r11)
 (car r12 r11)
 (cdr r13 r11)
 (insure-constant r13 nil)
 (insure-equal r3 r8)
 (insure-equal r3 r12)
```

This then expands into the following LISP code fragment:

```
(let ((r1 (dereference ?x)))
   (cond ((locativep r1)
          (push r1 *trail*)
          (setf (location-contents r1) 'b))
         ((not (eql r1 'b)) (go fail)))
   (let ((r2 'a))
     (let ((r3 (dereference ?u)))
       (let ((r4 'g))
         (let ((r5 (dereference ?w)))
           (let ((r6 'nil))
             (let ((r7 (cons-stack r5 r6)))
               (let ((r8 (cons-stack r4 r7)))
                 (let ((r9 (dereference ?z)))
                   (cond ((locativep r9)
                          (let ((z (fresh-cons-stack)))
                            (setf (location-contents r9) z)
                            (push r9 *trail*)
                            (setf r9 z)))
                         ((not (arrayp r9)) (go fail)))
                   (let ((r10 (car-array r9)))
                     (cond ((locativep r10)
                            (push r10 *trail*)
                            (setf (location-contents r10) 'h))
                           ((not (eql r10 'h)) (go fail)))
                     (let ((r11 (cdr-array r9)))
                       (cond ((locativep r11)
                              (let ((z (fresh-cons-stack)))
                                (setf (location-contents r11) z)
                                (push r11 *trail*)
                                (setf r11 z)))
                             ((not (arrayp r11)) (go fail)))
                       (let ((r12 (car-array r11)))
                         (let ((r13 (cdr-array r11)))
                           (cond ((locativep r13)
                                  (push r13 *trail*)
                                  (setf (location-contents r13) 'nil))
                                 ((not (eql r13 'nil)) (go fail)))
                           (if (unify r3 r8) (go fail))
                           (if (unify r3 r12) (go fail))
                           ⟨call target with r2 and r3⟩)))))))))))))
```

Compiling Queries To use the code thus generated, a query generates and executes the following code fragment as part of a query function:

```
(catch :succeed
   (let* ((*trail* nil)
          ⟨query variables⟩
          (next-goal
           (lambda () (declare (sys:downward-function)) ⟨solution handler⟩)))
     (declare (special *trail*))
     ⟨code for initializing query variables to be unbound⟩
     (let* ((next-goal (lambda ()
                         (declare (sys:downward-function))
                         ⟨code for last goal⟩))
            ⋮
            (next-goal (lambda ()
                         (declare (sys:downward-function))
                         ⟨code for third goal⟩))
            (next-goal (lambda ()
                         (declare (sys:downward-function))
                         ⟨code for second goal⟩)))
       ⟨code for first goal⟩)))
```

Query variables are initialized the same way as logic variables are in a clause function. Likewise, the code for each goal is generated in the same way as for clause functions. This code fragment executes the solution handler form in a context where the query variables are bound to values which satisfy the query. If another solution is desired, the solution handler backtracks by simply returning. If no further solution is needed, the solution handler can issue a `(throw :succeed (values))`.

4.2 Modifications to Support The Culprit Pointer Method

The previous section presented the LispWAM architecture, a number of Lisp code fragments into which Prolog programs can be compiled. This section describes how the LispWAM can be altered slightly to incorporate the Culprit Pointer Method for performing selective backtracking at goal failures which are not selectively-nested. Similar modifications can be made to the code generated by other Prolog compilers including those based more directly on the Warren Abstract Machine.

Most of the modifications to the LispWAM are additions which are needed to support the maintenance of culprit pointers. Updating the culprit pointers upon unification failure requires determining the consistency of various subsets of the unifications which led to that unification failure. (Remember, these subsets are of the form $\{C[0].C, \ldots, C[k].C, C[j].C, \ldots, C[i].C\}$.) Each constraint $C[l].C$ is a unification of some goal with the head of one of the alternative clauses for satisfying that goal. In the LispWAM the attempt to satisfy that goal involves calling a choice-point function which sets up a choice-point on the stack. To support the Culprit Pointer Method, these choice-points need two additional slots. These slots are provided by adding two new parameters to each choice-point function. The first parameter, `culprit`, is the culprit pointer for that choice-point. It contains the address of the stack frame corresponding to the choice-point to continue when that choice-point exhausts. The second new parameter is called `unifier`. This slot will contain a function which performs the unification between the goal and the head of the alternative clause currently chosen for that goal. The routine which updates the `culprit` pointers will isolate the appropriate subsets of choice-points from the stack, access the unification routines via the `unifier` slots of those frames, and call these unification routines in succession to determine the consistency of the associated unification constraints.

Compiling Unification When compiling code which implements the CULPRIT POINTER METHOD, there need to be two copies of each code fragment which unifies a goal with a head. The first, or primary, copy is used during the forward-going component of PROLOG execution as part of its parameter passing mechanism. This copy is integrated directly into the code generated for different alternatives for goals within clause functions. For example, in the following fragment:

```
p(x):-q(x). %clause A
q(y):-r(y). %clause B
```

the code which unifies `q(x)` with `q(y)` is built into the clause function for clause A as part of one alternative for the goal `q(x)` and the way it calls the clause function for clause B. This code cannot be used by the routine which updates the `culprit` pointers because it cannot be accessed independently of the mechanism by which clause A calls clause B. Therefore, a second copy of this unification code is made which is packaged up in a `lambda` expression and stored in the `unifier` slot of the choice-point created for the goal `q(x)`. This second unifier is called the *auxiliary unifier function*.

Why have two copies of the unification code for each goal-head pair? In addition to the added efficiency of having one copy integrated inline into the clause function, the two copies differ somewhat from one another. There are two reasons for this difference. First, the primary and auxiliary unifiers must reference two different copies of the clause variables. The auxiliary unifier is always called in the context of a failure encountered by the primary unifier. The clause variables used by the primary unifier contain the current state of the search and cannot be disrupted by the auxiliary unifier which is only being called to update the `culprit` pointers. Therefore, each clause function is provided with a duplicate set of parameters called the *auxiliary clause variables*. These auxiliary clause variables are given distinct names by associating with each clause variable `?x`, an auxiliary clause variable named `goal-?x`. The auxiliary unification code then operates on these auxiliary clause variables instead of the primary clause variables.

The reason for the second difference is more complicated. In the general case, a unifier must unify two terms, one a goal, the other a head. The variables in these two terms are disjoint. The goal references clause variables for the clause which it is part of, while the head references clause variables for the clause which it is part of. Unifying the goal with the head may create links between these two sets of clause variables. The code for the primary unifier is designed to be used only as part of the ordinary PROLOG function call process. In such circumstances, the clause variables of the clause which contains the head term are known to be unbound prior to the unification. Accordingly, the primary unifier is optimized to handle only this special case and is divided into two components. The first component performs a one-way consistency check on the values of the variables in the goal term to see whether they can unify with the head term under the assumption that all of its variables are unbound. The second component then computes the values for the clause variables of the head clause which result from this unification and passes on these values as arguments when the clause function for the head clause is called. Thus the primary unification code only explicitly references the clause variables of the clause containing the goal term.

The auxiliary unifier cannot allow this optimization as it is called in contexts when the clause variables of both the goal term and the head term are bound. While the primary unifier explicitly references only the clause variables of the goal clause, the auxiliary unifier must reference the auxiliary clause variables of both the goal clause as well as the head clause. The auxiliary clause variables of the head clause are referenced via names of the form `head-?x`. The auxiliary unifier corresponding to the unification of a goal with a head is created in a lexical context where the head auxiliary variables are created and made unbound. These variables are then passed on to become the goal auxiliary variables of the goal clause.

Because the auxiliary unifier is more symmetrical than the primary unifier, the sequence of macro instructions generated for it is different. When unifying a goal variable with a head variable the following macro-instruction sequence is generated:

```
(parameter ⟨register1⟩ ⟨goal variable⟩)
 (parameter ⟨register2⟩ ⟨head variable⟩)
 (insure-equal ⟨register1⟩ ⟨register2⟩)
```

When unifying a goal variable with a head constant, the following macro-instruction sequence is generated:

```
(parameter ⟨register⟩ ⟨goal variable⟩)
 (insure-constant ⟨register⟩ ⟨head constant⟩)
```

Symmetrical code is generated when unifying a head constant with a goal variable. When unifying a goal variable with a head cons, the goal variable is destructured and then code is appended to unify the car and cdr components.

```
(parameter ⟨cons register⟩ ⟨goal variable⟩)
 (insure-cons ⟨cons register⟩)
 (car ⟨car register⟩ ⟨cons register⟩)
 (cdr ⟨cdr register⟩ ⟨cons register⟩)
 ⟨macro-instructions to unify the car register with the car of the head cons⟩
 ⟨macro-instructions to unify the cdr register with the cdr of the head cons⟩
```

Symmetrical code is generated when unifying a goal cons with a head variable. The remaining cases are the same as for the primary unifier. Note that only one `parameter` macro-instruction should be generated for each variable which appears in either the goal term or head term.

The greater symmetry of the auxiliary unifier results in two important differences between the code generated for the primary and auxiliary unifiers. First, the auxiliary unifier never contains `cons` and `constant` macro-instructions. Unification fragments which would generate these macro-instructions instead generate the corresponding `insure-cons` and `insure-constant` macro-instructions. Second, there is no need to rename secondary occurrences and generate `insure-equal` macro-instructions for them. Instead, many more `insure-equal` macro-instructions are generated for the case of unifying a goal variable with a head variable which would not otherwise be generated.

The differences between the code generated for the primary unifier and the auxiliary unifier are highlighted by the following example which is repeated from before. Unifying the goal term

```
(p (f ?x a) ?u (g ?w) ?z)
```

with the head term

```
(p (f b ?y) ?v ?v (h ?v))
```

generates the macro-instruction sequence on the left for the primary unifier and the macro-instruction sequence on the right for the auxiliary unifier:

```
(parameter r1 ?x)
 (insure-constant r1 b)
 (constant r2 a)
 (parameter r3 ?u)
 (constant r4 g)
 (parameter r5 ?w)
 (constant r6 nil)
 (cons r7 r5 r6)
 (cons r8 r4 r7)
 (parameter r9 ?z)
 (insure-cons r9)
 (car r10 r9)
 (insure-constant r10 h)
 (cdr r11 r9)
 (insure-cons r11)
 (car r12 r11)
 (cdr r13 r11)
 (insure-constant r13 nil)
 (insure-equal r3 r8)
 (insure-equal r3 r12)
```

```
(parameter r1 goal-?x)
 (insure-constant r1 b)
 (parameter r2 head-?y)
 (insure-constant r2 a)
 (parameter r3 goal-?u)
 (parameter r4 head-?v)
 (insure-equal r3 r4)
 (insure-cons r4)
 (car r5 r4)
 (insure-constant r5 g)
 (cdr r6 r4)
 (insure-cons r6)
 (car r7 r6)
 (parameter r8 goal-?w)
 (insure-equal r8 r7)
 (cdr r9 r6)
 (insure-constant r9 nil)
 (parameter r10 goal-?z)
 (insure-cons r10)
 (car r11 r10)
 (insure-constant r11 h)
 (cdr r12 r10)
 (insure-cons r12)
 (car r13 r12)
 (insure-equal r10 r13)
 (cdr r14 r12)
 (insure-constant r14 nil)
```

The auxiliary unifier code is more symmetric. Essentially it destructures each term, both in the goal and the head, and unifies each corresponding subterm separately. The primary unifier on the other hand is less symmetric. It only checks to see that the goal term conforms to the template required by the head term. It then computes the values for the head term variables by structuring and destructuring the goal term.

Keeping track of the `*base-frame*` As the choice-point stack is represented via the underlying Lisp function call stack which may contain frames other than choice points, several code fragments which access the choice-point stack need to know its extent. This is done by creating a special variable `*base-frame*` which is bound to the address of the stack frame corresponding to the query function. The `*base-frame*` itself is not a choice-point. Every frame subsequent to the `*base-frame*`, up to the most recent frame are choice-points, however. Among the code fragments to be discussed, the functions `update-culprit-pointers` and `reset-culprit-pointers`, as well as the code generated at the end of each goal for checking whether the `culprit` pointer points to the previous choice-point, are the only three places which rely on the fact that there is a one-to-one correspondence between stack frames and choice-points. Furthermore, while all other non-choice-point functions discussed so far can be safely open-coded, the two aforementioned functions assume that they are not open-coded so that when they are called, the most recent choice-point is not the current stack frame but the one previous to it. Accordingly, they must be modified to change that assumption before they can be open-coded.

Compiling Clauses The code generated for clause functions is modified in three ways. First, the additional parameters `culprit` and `unifier` are added to both the clause function as well as to each goal continuation. Second, additional auxiliary clause variables of the form `goal-?x` are added as parameters to the clause function. Finally, a parameter `parent` is added to the clause function. This parameter maintains a pointer to the choice-point corresponding to the grandparent $\vee$-vertex of each of the $\vee$-vertices corresponding to the goals contained in this clause. As a choice-point is created for each goal in this clause, the `culprit` pointer of that choice-point is initialized to the value of `parent` for that clause. The resulting code fragment for clause functions is shown below.

```
(defun CLAUSE-i (next-goal culprit unifier parent
                 ⟨goal auxiliary clause variables⟩ ⟨argument variables⟩
                 &aux ⟨logic variables⟩)
  (declare (special *trail*))
  ⟨code for initializing logic variables to be unbound⟩
  (let* ((next-goal (lambda (culprit unifier)
                      (declare (sys:downward-function))
                      ⟨code for last goal⟩))
           ⋮
         (next-goal (lambda (culprit unifier)
                      (declare (sys:downward-function))
                      ⟨code for third goal⟩))
         (next-goal (lambda (culprit unifier)
                      (declare (sys:downward-function))
                      ⟨code for second goal⟩)))
   ⟨code for first goal⟩))
```

Compiling Goals The code generated for each goal must be modified in two ways. First, the `culprit` pointer must be initialized to point to the `parent` choice-point before the first alternative for that goal is considered. Second, code must be added to actually perform the selective backtrack after the last alternative fails by returning. Without the CULPRIT POINTER METHOD, after the last alternative for a goal fails by returning, the goal itself fails by returning. When the CULPRIT POINTER METHOD is added, code must be provided to continue a more deeply nested choice-point which is pointed to by the `culprit` pointer. This is done by setting up a `catch` frame around each alternative in a choice-point whose tag is the address of that choice-point. The next-alternative of a nested choice-point is continued by `throw`ing to the `catch` tag which is the address of that choice-point. This is done by the following code fragment:

```
(setf culprit parent)
 (let ((trail *trail*))
   (sys:with-data-stack
    ⟨code for first alternative⟩
    (unwind trail))
   (sys:with-data-stack
    ⟨code for second alternative⟩
    (unwind trail))
    ⋮
   (sys:with-data-stack
    ⟨code for next to last alternative⟩
    (unwind trail)))
 ⟨code for last alternative⟩
 (if (sys:%pointer-lessp culprit
                          (sys:frame-previous-frame (sys:%stack-frame-pointer)))
     (let ((culprit culprit))
       (reset-culprit-pointers)
       (throw culprit (values)))        ;does not return any values
     (values))                          ;does not return any values
```

Note that before issuing the `throw` to the choice-point pointed to by the `culprit` pointer, a check is made to see whether that failure is actually a selective one, i.e. whether it continues a choice-point more deeply nested than the previous one. If not, the goal continuation simply returns since in the Symbolics architecture, a LISP return is more efficient than a `throw`. If it is selective, then the `culprit` pointers for all nested choice-points must be reset to point to the next previous choice-point to prevent nested selective backtracks. This is done by the function `reset-culprit-pointers`. As this function would overwrite the `culprit` pointer for the current stack frame as well, a copy is made to preserve its value as the target of the `throw` after `reset-culprit-pointers` returns. It should be noted that since a selective backtrack can never happen upon the failure of the first goal in a clause or query, the final `if` statement of the above fragment is not included as part of the code fragment generated for the first goal in a clause or query. In its place a simple `(values)` form is generated.

The function which resets the `culprit` pointers is shown below:

```
(defun RESET-CULPRIT-POINTERS ();can NOT be open-coded
  (declare (special *base-frame*))
  (loop for frame = (sys:frame-previous-frame (sys:%stack-frame-pointer))
                then previous-frame
        for previous-frame = (sys:frame-previous-frame frame)
        until (eq frame *base-frame*)
        do (setf (location-contents
                   (sys:%make-pointer-offset sys:dtp-locative frame 1))
                  previous-frame))
  (values))                             ;does not return any values
```

This function relies on the fact that the parameter `culprit` is at offset 1 in the stack frame for every choice-point function. It also must be modified if it is to be open-coded as it assumes that the most recent choice-point is the stack frame previous to the current stack frame.

Compiling Alternatives The code for generated for alternatives is modified in six ways. First, the alternative is wrapped in a `let` to allocate the auxiliary clause variables for the head clause. Second, the head auxiliary clause variables are initialized to be unbound. Third, the `unifier` slot of the current choice-point is initialized to contain the auxiliary unifier for this choice-point. Fourth, the actual code for the alternative is wrapped in a `catch` whose tag is the address of the current choice-point. Throwing to this tag will continue the next alternative for this choice-point. Fifth, calls to choice-point functions must be augmented to supply values for the new `culprit` and `unifier` parameters, as well as the `parent` parameter in the case of calling a clause function. When calling a clause function, the `parent` parameter is initialized to `(sys:%stack-frame-pointer)` which is the choice-point of the grandparent $\vee$-vertex of this newly created choice-point. When calling both clause functions and goal continuations, the `culprit` pointer and `unifier` slots are initialized to a dummy value of `nil`. The `culprit` pointer will be initialized to the value of `parent` which is lexically visible for that choice-point function, before the first alternative for that goal is considered. The `unifier` slot will be filled in later by the code generated for each alternative of that goal. Finally, a call to the routine `update-culprit-pointers` must be generated for the case when the primary unifier fails. The code generated for a fact alternative is given below:

```
(let ⟨head auxiliary clause variables⟩
   (setf unifier (lambda ()
                  (declare (sys:downward-function))
                  (block nil
                   (tagbody
                      ⟨auxiliary unifier code⟩
                      (return t)
                   fail
                      (return nil)))))
   ⟨code for initializing head auxiliary clause variables to be unbound⟩
   (catch (sys:%stack-frame-pointer)
     (tagbody
        ⟨primary unifier code⟩
        (funcall next-goal nil nil)
        (go next)
     fail
        (update-culprit-pointers)
     next)))
```

The code generated for a non-fact alternative has one additional change. The head auxiliary clause variables must be passed on to the head clause function to become its goal auxiliary clause variables. The resulting code fragment is shown below:

```
(let ⟨head auxiliary clause variables⟩
   (setf unifier (lambda ()
                   (declare (sys:downward-function))
                   (block nil
                    (tagbody
                       ⟨auxiliary unifier code⟩
                       (return t)
                    fail
                       (return nil)))))
   ⟨code for initializing head auxiliary clause variables to be unbound⟩
   (catch (sys:%stack-frame-pointer)
     (tagbody
        ⟨primary unifier code⟩
        (clause-j
         next-goal nil nil (sys:%stack-frame-pointer)
         ⟨head auxiliary clause variables⟩ ⟨arguments⟩)
        (go next)
     fail
        (update-culprit-pointers)
     next)))
```

The `culprit` pointers are updated by the following function:

```
(defun UPDATE-CULPRIT-POINTERS ()    ;can NOT be open-coded
  (declare (special *trail* *base-frame*))
  (let ((trail *trail*))
    (loop for j-frame = (sys:frame-previous-frame (sys:%stack-frame-pointer))
                  then j-previous-frame
          for j-previous-frame = (sys:frame-previous-frame j-frame)
          while (funcall (location-contents (sys:%make-pointer-offset
                                              sys:dtp-locative j-frame 2)))
          for trail = *trail*
          for culprit-location =
              (sys:%make-pointer-offset sys:dtp-locative j-frame 1)
          for culprit = (location-contents culprit-location)
          when (sys:%pointer-lessp culprit j-previous-frame)
            do (loop for k-previous-frame = *base-frame* then k-frame
                     for k-frame =
                         (loop for l-frame = (sys:%stack-frame-pointer)
                                         then l-previous-frame
                               for l-previous-frame =
                                   (sys:frame-previous-frame l-frame)
                               until (eq l-previous-frame k-previous-frame)
                               finally (return l-frame))
                     while (funcall (location-contents
                                     (sys:%make-pointer-offset
                                      sys:dtp-locative k-frame 2)))
                     finally (if (sys:%pointer-lessp culprit k-frame)
                                 (setf (location-contents culprit-location)
                                       k-frame)))
               (unwind trail))
    (unwind trail))
  (values))                              ;does not return any values
```

This function implements lines 29 through 37 of figure 3.2. Checking the consistency of a set of unification constraints corresponding to a set of choice-points is done by calling the `unifier`s of each choice-point, in sequence, and seeing if one returns `nil`. Between each consistency check, the trail is unwound to undo the bindings created by the unifications done during that check.

This function relies on the fact the the `culprit` pointer for each choice-point is located at offset 1 of the stack frame, and that the `unifier` for each choice-point is located at offset 2 of the stack frame. Additionally, it assumes that the most recent choice-point is the stack frame previous to the current stack frame; it must be modified to be open-coded.

Compiling Queries The code generated for queries must be modified as well. First, the special variable `*base-frame*` must be initialized to the address of the stack frame of the query. This special variable is used by the routines `update-culprit-pointers` and `reset-culprit-pointers` to determine the extent of the underlying LISP funcall call stack which constitutes the choice-point stack. Second, like all frames which can be the target of some selective backtracking `throw`, a `catch` frame must be set up for this base frame. Third, the additional parameters `culprit`, `unifier` and `parent` must be added to the query function. Likewise, the parameters `culprit` and `unifier` must be added to its embedded goal continuations as well. The `parent` parameter for the query function is initialized to point to the `*base-frame*`. Its `culprit` pointer and `unifier` slot are initialize to a dummy value of `nil`. The `culprit` pointer will be initialized to the `parent` value when the choice-point function is called. The `unifier` slot will be filled in later by each alternative of the first goal. Note that although the `lambda` expression corresponding to the query function is immediately `funcall`ed, it is not possible to β-reduce this expression because doing so would eliminate the stack frame for the query function. Fourth, auxiliary query variables of the form `goal-?x` are allocated and initialized to be unbound in addition to the normal query variables. Finally, a call to `reset-culprit-pointers` is inserted after the solution handler to maintain soundness. The resulting code generated for query functions is illustrated by the following code fragment:

```
(catch :succeed
   (let ((*base-frame* (sys:%stack-frame-pointer)))
     (declare (special *base-frame*))
     (catch (sys:%stack-frame-pointer)
       (funcall
        (lambda (culprit unifier parent)
          (declare (sys:downward-function))
          (let ((*trail* nil)
                ⟨query variables⟩
                ⟨auxiliary query variables⟩
                (next-goal (lambda (culprit unifier)
                             (declare (sys:downward-function))
                             ⟨solution handler⟩
                             (reset-culprit-pointers))))
            (declare (special *trail*))
            ⟨code for initializing query variables to be unbound⟩
            ⟨code for initializing auxiliary query variables to be unbound⟩
            (let* ((next-goal (lambda (culprit unifier)
                                (declare (sys:downward-function))
                                ⟨code for last goal⟩))
                      ⋮
                   (next-goal (lambda (culprit unifier)
                                (declare (sys:downward-function))
                                ⟨code for third goal⟩))
                   (next-goal (lambda (culprit unifier)
                                (declare (sys:downward-function))
                                ⟨code for second goal⟩)))
              ⟨code for first goal⟩)))
        nil
        nil
        *base-frame*)))
```

4.3 Additional Details of the Compilation Process

4.3.1 Tail Recursion

The WAM normally includes an optimization which is akin to tail merging done for tail recursive Lisp code. When the last alternative for some goal is called, that call is deterministic. The choice-point for that goal is no longer needed and may be popped off the choice-point stack before the final alternative is called. The code generated by the LispWAM has the property that if the underlying Lisp compiler performs tail merging then the PROLOG code inherits this as tail merging of final alternatives for choice-points. The ability to perform this tail merging disappears when the LISPWAM is extended to incorporate the CULPRIT POINTER METHOD because the call to the final alternative is no longer the last fragment executed for a goal. Code is inserted after the last alternative of every goal except the first goal in each clause or query, to be called when it returns. This code is what actually performs the selective backtrack if it is possible.

This limitation is not just a property of the LISPWAM. No implementation of the CULPRIT POINTER METHOD can incorporate tail merging on the choice-point stack. This is because even when the last alternative for a goal is being pursued, the `culprit` pointer and unifier slots for the choice-point created for that goal must be retained. Even if the `culprit` pointer already points to the previous choice-point on the stack and thus rules out any selective backtracking, the unifier slot is still needed to allows updating nested `culprit` pointers. In fact, the particular implementation of the CULPRIT POINTER METHOD discussed in this chapter will fail if it is run on a LISP implementation which supports tail merging. Such a LISP implementation will try to perform tail merging on the last alternative of the first goal in a clause or query, which has no subsequent code for performing selective backtracks. This tail merging will elide the stack frame, and its associated `unifier` slot, wreaking havoc with subsequent attempts to update the `culprit` pointers. Thus the CULPRIT POINTER METHOD is not suitable for applications where tail merging is tantamount to successful execution.

4.3.2 Clause Indexing

Most PROLOG implementations perform clause indexing whereby unifications are not attempted for alternatives which are known to fail given the current variable bindings. A straightforward combination of clause indexing with the CULPRIT POINTER METHOD can result in unsound backtracking behavior as illustrated by the following example:

```
?- gen(X),gen(Y),test(Y,X).
gen(a).
gen(b).
gen(c).
gen(d).
gen(e).
test(a,b).
test(b,a).
```

When `X` and `Y` are both bound to `a`, clause indexing implies that only the first alternative, `test(a,b)`, is tried for the goal `test(Y,X)`. When that alternative fails, the `culprit` pointer for `test(Y,X)` is set to point to the goal `gen(X)` because the constraint set

$$\{\langle \mathtt{gen(X)}, \mathtt{gen(a)}\rangle, \langle \mathtt{test(Y,X)}, \mathtt{test(a,b)}\rangle\}$$

is inconsistent. The goal `test(Y,X)` now exhausts, since it has no further alternatives, and the search backtracks selectively to `gen(X)` skipping over `gen(Y)`. In doing so, the solution `X=a,Y=b` is missed. This happens because clause indexing eliminated not only the consideration of the second alternative for `test(Y,X)` but the updating of the `culprit` pointers that would have occurred upon that unification failure as well. In this case without clause indexing, the second unification failure would have updated the `culprit` pointer for `test(Y,X)` to point to `gen(Y)` and no solution would have been missed.

One might try to develop a mechanism to determine the choice-point which caused the pruning of alternatives by clause indexing, and update the `culprit` pointer to point to this choice-point at the earliest. For example, in the above example, the alternative `test(b,a)` was pruned by clause indexing, because `Y` was bound to `a`. As this binding was created by the current alternative for the choice-point corresponding to the goal `gen(Y)`, the `culprit` pointer for the goal `test(Y,X)` should be updated to point to this goal. Although this will probably always lead to sound backtracking behavior, it can reduce the effectiveness of the Culprit Pointer Method. Consider the following example:

```
?- gen(X1),gen(X2),gen(X3),gen(X4),gen(Y),test(Y,X1,X2,X3,X4).
gen(a).
gen(b).
gen(c).
gen(d).
gen(e).
test(a,b,b,b,b). % alternative 1
test(b,b,_,_,_). % alternative 2
test(b,_,b,_,_). % alternative 3
test(b,_,_,b,_). % alternative 4
test(b,_,_,_,b). % alternative 5
test(b,_,_,_,_). % alternative 6
```

Here, when `Y=a`, alternatives two through six are pruned by clause indexing. The safe clause indexing method discussed above would require updating the `culprit` pointer for `test(Y,X1,X2,X3,X4)` to point to `gen(Y)`. If alternative six was not present, the above method would still backtrack only to `gen(Y)`, even though it would be sound to backtrack to `gen(X4)`. The full Culprit Pointer Method would detect this. Furthermore, if alternatives five and six were not present, it would be sound to backtrack to `gen(X3)`. Likewise, deleting alternatives four through six, licenses backtracking to `gen(X2)` while deleting alternatives three through six, licenses backtracking to `gen(X1)`. Therefore, there seems to be no way to perform as well as the full Culprit Pointer Method without doing the full updating of `culprit` pointers on every unification failure.

To counterbalance the loss of clause indexing, the LispWAM incorporates a static compile-time form of clause indexing, which although is not as powerful as full dynamic run-time clause indexing, is nonetheless compatible with the Culprit Pointer Method. The conventional WAM groups together clauses whose heads have the same functor and number of arguments, and compiles them into one function. Any goal term with this same functor and number of arguments calls this function, even if it contains alternatives which could not possibly unify with this particular goal. The LispWAM instead compiles a separate function for each clause rather than groups of related clauses. A goal then compiles into a non-deterministic call to only those clauses where the unification is not ruled out at compile time. This is sound in the context of the Culprit Pointer Method because no `culprit` pointers would ever be updated at a unification failure where that single unification was inconsistent independent of context. This technique has the disadvantage that code generated to handle the non-determinism of choosing alternative clauses has moved from the target of the call, to the source of the call, and results in many more code variants being generated. On the other hand, this technique has an additional side benefit that in some cases the amount of unifier code which needs to be generated can be reduced. For example, when compiling the following code fragment:

```
...,p(X,f(Y,Z)),...
p(U,f(V,W)):-...
```

in the conventional WAM architecture, the calling goal must create the structure `f(Y,Z)` which will immediately be destructured by the target clause. Because in the LispWAM, the unification code is moved from the target to the source of a call, that code can be customized to generate code which is particular to each unification which can eliminate needless structuring and destructuring.

4.3.3 Is

The `is` builtin predicate poses special problems for the CULPRIT POINTER METHOD. To deal with the `is` construct we assume that an `is`-goal creates its own choice-point which has only a single alternative and whose `unifier` slot contains a new type of atomic constraint. In order to update the `culprit` pointers, the CHECK function must be able to determine the consistency of sets containing two types of atomic constraints: conventional unification constraints between two terms, and an `is`-constraint between a variable and an expression which is evaluated relative to some variable bindings. The following example demonstrates some of the problems with this approach:

```
?- gen(X),gen(Y),gen(Z),Z is X+Y.
gen(1).
gen(2).
gen(3).
gen(4).
gen(5).
```

An `is`-constraint of the form x `is` e can fail for one of two reasons. Either x is bound to a value which does not unify with the value produced by e, or some variable referenced by e is unbound which prevents e from being evaluated. We will call the first failure type an *unbound* `is`*-failure* and the later failure type a *bound* `is`*-failure*. In the above example, the goal `Z is X+Y` causes a bound `is`-failure. When attempting to update the `culprit` pointer for this goal, the consistency of the following subsets of atomic constraints must be checked in order:

$$\{\langle \texttt{Z is X} + \texttt{Y}\rangle\}$$
$$\{\langle \texttt{gen(X)}, \texttt{gen(1)}\rangle, \langle \texttt{Z is X} + \texttt{Y}\rangle\}$$
$$\{\langle \texttt{gen(X)}, \texttt{gen(1)}\rangle, \langle \texttt{gen(Y)}, \texttt{gen(1)}\rangle, \langle \texttt{Z is X} + \texttt{Y}\rangle\}$$
$$\{\langle \texttt{gen(X)}, \texttt{gen(1)}\rangle, \langle \texttt{gen(Y)}, \texttt{gen(1)}\rangle, \langle \texttt{gen(Z)}, \texttt{gen(1)}\rangle, \langle \texttt{Z is X} + \texttt{Y}\rangle\}$$

The problem here is that although the original failure is a bound `is`-failure, the first two subsets above are inconsistent due to unbound `is`-failures. If these inconsistencies were allowed to stop the process of updating the `culprit` pointer for the goal `Z is X+Y`, that `culprit` pointer would be left at zero causing the search to terminate unsoundly upon the exhaustion of the `is`-goal, thus missing the solution `X=1,Y=1,Z=2`.

The solution is to ignore unbound `is`-failures while updating the `culprit` pointers. As the following example shows, this alone is not enough to alleviate the problem.

```
?- gen(X),gen(Y),gen(Z),gen(W),Z is X+Y.
gen(1).
gen(2).
gen(3).
gen(4).
gen(5).
```

Updating the `culprit` pointers upon the failure of the `is`-goal will require checking the consistency of various subsets of the following atomic constraint set:

$$\{\langle \texttt{gen(X)}, \texttt{gen(1)}\rangle, \langle \texttt{gen(Y)}, \texttt{gen(1)}\rangle, \langle \texttt{gen(Z)}, \texttt{gen(1)}\rangle, \langle \texttt{Z is X} + \texttt{Y}\rangle\}$$

This is normally done incrementally by starting with the empty set and adding in atomic constraints one by one until the consistency of the entire set is checked. The code fragments given in section 4.2, perform this consistency check by first adding in the constraint `Z is X+Y` and then adding in the atomic constraints for `X`, `Y`, and `Z` in that order. In general, when checking the consistency of subsets of the form

$$\{C[0].C, \ldots, C[k].C, C[j].C, \ldots, C[i].C\}$$

the constraints are added in from i down to j and then from 0 up to k. This is done so that the work done to add in the atomic constraint $C[j].C$ in order to update the `culprit` pointer for choice-point j can be reused while updating `culprit` pointers for choice-points prior to j. The problem that is illustrated by the above example is that when adding in the atomic constraint $\langle$`Z is X` $+$ `Y`$\rangle$, an unbound `is`-failure results which is ignored. Then the remaining atomic constraints are added in after ignoring this `is`-constraint and the atomic constraint set is determined to be consistent when it should be inconsistent. This problem can be avoided by always adding in atomic constraints from least recent to most recent when checking the consistency of an atomic constraint set. This however, is in conflict with the aforementioned optimization and requires more work running the CHECK function while updating the `culprit` pointers upon every unification failure.

One final note about `is`-constraints. The CULPRIT POINTER METHOD is sound only for bound `is`-failures. When an unbound `is`-failure occurs, only a non-selective backtrack to the most recent choice-point should occur. This is because `is`-constraints violate the monotonicity requirement for the CHECK function. A given atomic constraint set may be inconsistent because of an unbound `is`-failure and yet a superset of that set may be consistent. Even though the previous choice-point might be for a goal which appears not to bind the variables which cause the unbound `is`-failure, they may be linked, however indirectly, to other variables via remaining alternatives for the previous choice-point and eventually become bound and lead to a solution.

It should be pointed out that the ideas in this subsection are preliminary and have not been tested via implementation.

Chapter 5

Experimental Results

In order to assess the value of the Culprit Pointer Method, the implementation presented in the previous chapter was tested on two standard benchmarks often used to test other intelligent backtracking schemes presented in the literature. These two benchmarks are the N-Queens problem and the map coloring problem given by Bruynooghe and Pereira[8].

A problem arises when attempting to do a more thorough comparison between the different published schemes. A number of the published benchmarks, namely the circuit design program used by Kumar and Lin[51], and the N-Queens program used by Bruynooghe and Pereira, are not written in pure Prolog. Because the implementation of Prolog incorporating the Culprit Pointer Method discussed in chapter 4 currently does not include cut, arithmetic, and other extra-logical primitives, benchmarks such as these can not be run as written. As the circuit design program is heavily based on extra-logical primitives, no attempt was made to modify and perform the benchmark. For the N-Queens program, however, a modified version was written in pure Prolog which uses Peano arithmetic to perform consistency checks. The results of benchmarking this version are given in this chapter. These results are misleading however, since performing Peano arithmetic entails a significant increase in the number of unification failures with the associated increase in time spent updating the culprit pointers on each such failure. Accordingly, the N-Queens problem was benchmarked a second time, being encoded as a constraint satisfaction problem being solved by a dedicated program which incorporates the Culprit Pointer Method. The remainder of this chapter discusses the three benchmarks performed: the map coloring example, the Prolog N-Queens example, and the N-Queens problem run as a dedicated constraint satisfaction problem.

5.1 The Map Coloring Example

Bruynooghe and Pereira[8] present the problem of finding a 4-coloring of a particular graph which has 13 vertices and 31 edges. They give two Prolog programs for the task which differ from each other only in the order of the goals in one clause. Figure 5.1 gives the Prolog code for both the "good" and "bad" ordering.

Both the "good" ordering and the "bad" ordering were run twice; first to compute just a single solution, and then to compute all 1176 solutions. Each of these four runs were performed both with conventional chronological backtracking as well as with selective backtracking as performed by the Culprit Pointer Method. The results of these runs are given in table 5.1. The upper half of this table gives the data for chronological backtracking while the lower half gives the corresponding data for selective backtracking. The statistics gathered during selective backtracking distinguish between unifications per-

```
?-good_goal(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13).
?-bad_goal(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13).
good_goal(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13):-
 next(R1,R13),next(R1,R2),next(R2,R13),next(R2,R4),
 next(R4,R10),next(R6,R10),next(R8,R13),next(R6,R13),
 next(R2,R3),next(R3,R4),next(R3,R13),next(R3,R5),
 next(R5,R6),next(R5,R13),next(R4,R5),next(R5,R10),
 next(R1,R7),next(R7,R13),next(R2,R7),next(R4,R7),
 next(R7,R8),next(R4,R9),next(R9,R10),next(R8,R9),
 next(R9,R13),next(R6,R11),next(R10,R11),next(R11,R13),
 next(R9,R12),next(R11,R12),next(R12,R13).
bad_goal(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13):-
 next(R1,R2),next(R2,R3),next(R3,R4),next(R4,R5),
 next(R5,R6),next(R6,R11),next(R11,R12),next(R12,R13),
 next(R9,R13),next(R9,R10),next(R4,R10),next(R4,R7),
 next(R7,R8),next(R2,R7),next(R6,R10),next(R2,R13),
 next(R6,R13),next(R2,R4),next(R8,R13),next(R4,R9),
 next(R3,R5),next(R8,R9),next(R1,R13),next(R3,R13),
 next(R5,R13),next(R7,R13),next(R11,R13),next(R9,R12),
 next(R5,R10),next(R10,R11),next(R1,R7).
next(blue,yellow).
next(blue,red).
next(blue,green).
next(yellow,blue).
next(yellow,red).
next(yellow,green).
next(red,blue).
next(red,yellow).
next(red,green).
next(green,blue).
next(green,yellow).
next(green,red).
```

Figure 5.1: Prolog code for the map coloring benchmark.

	good ordering		bad ordering	
	first solution	all 1176 solutions	first solution	all 1176 solutions
# goal failures	12	48,746	89,218	7,282,310
# unifications	320	584,941	1,070765	87,387,709
time (seconds)	0.2	51.5	93.5	7419.0
# goal failures	9	37,610	10	76,556
# selective goal failures	3	7,704	9	36,454
# unifications	300	520,837	638	2,564,738
# auxiliary unifications	2,831	4,374,103	10,552	38,361,881
time (seconds)	3.4	2,283.0	9.7	21,774.9
selective goals failures	33%	20%	90%	48%
decrease in # goal failures	25%	23%	99%	99%
decrease in # unifications	-878%	-737%	99%	53%
decrease in time	-1,600%	-4,333%	90%	-194%

Table 5.1: Statistics gathered when running the map coloring benchmark. The upper portion of the table is for chronological backtracking while the lower portion is for selective backtracking using the Culprit Pointer Method.

formed as part of normal execution ("# unifications") and unifications performed in order to update the culprit pointers ("# auxiliary unifications"). While the Culprit Pointer Method offers a consistent reduction in both the number of goal failures as well as the number of primary unifications, only for the bad ordering is there a decrease in the total number of unifications performed. Even then, there is a reduction in running time only when computing the first solution for the bad ordering. For the remainder of the benchmarks, the Culprit Pointer Method performs worse than chronological backtracking.

5.2 The N-Queens Example

Because the Prolog implementation discussed in chapter 4 does not support cut, arithmetic, and other extra-logical primitives, the standard N-Queens program which is often used to benchmark intelligent backtracking implementations was rewritten in pure Prolog. The code for this benchmark is given in figure 5.2.

This code was run to find all solutions for $N = 1, \ldots, 6$ both with and without the Culprit Pointer Method. The results of these runs are given in table 5.2. Like before, the upper half of the table gives the results for chronological backtracking, while the lower half gives the results for selective backtracking. Again, while the Culprit Pointer Method affords a consistent decrease in the number of goal failures, the cost of updating the culprit pointers on unification failure is prohibitive, both in terms of the number of auxiliary unifications as well as running time.

5.3 The N-Queens Example as a Constraint Satisfaction Problem

The running times for the Culprit Pointer Method on the Prolog N-Queens example are skewed because using Peano arithmetic, instead of some builtin form of arithmetic, causes a significant increase in the number of unification failures. These unification failures require additional calls to the unifier to update the culprit pointers even though such updating can not contribute to any selective backtracking.

```
listBelow(0,[]).
listBelow(s(N),[s(N)|L]):-listBelow(N,L).
append([],Y,Y).
append([A|X],Y,[A|Z]):-append(X,Y,Z).
permute([],[]).
permute([A|X],Y):-permute(X,Z),append(U,V,Z),append(U,[A|V],Y).
sum(X,0,X).
sum(X,s(Y),s(Z)):-sum(X,Y,Z).
diff(X,Y,Z):-sum(Y,Z,X).
check2(R1,C1,R2,C2):-diff(R1,R2,D),diff(C1,C2,E),diff(D,E,s(F)).
check2(R1,C1,R2,C2):-diff(R1,R2,D),diff(C1,C2,E),diff(E,D,s(F)).
check2(R1,C1,R2,C2):-diff(R1,R2,D),diff(C2,C1,E),diff(D,E,s(F)).
check2(R1,C1,R2,C2):-diff(R1,R2,D),diff(C2,C1,E),diff(E,D,s(F)).
check1(R1,C1,0,[]).
check1(R1,C1,s(R2),[C2|Cs]):-check2(R1,C1,s(R2),C2),check1(R1,C1,R2,Cs).
check(0,[]).
check(s(N),[C1|Cs]):-check1(s(N),C1,N,Cs),check(N,Cs).
nQueens(N,S):-listBelow(N,L),permute(L,S),check(N,S).
```

Figure 5.2: Prolog code for the N-Queens benchmark.

N	1	2	3	4	5	6
# goal failures	10	92	480	3,268	23,978	195,178
# unifications	19	167	879	6,039	44,675	366,275
time (seconds)	0.779	0.422	0.912	3.830	26.931	235.711
# goal failures	10	68	341	2,281	16,471	132,126
# selective goal failures	0	8	36	208	1,340	9,746
# unifications	19	143	733	5,017	36,769	298,833
# auxiliary unifications	34	1,364	14,044	168,936	1,846,561	20,368,068
time (seconds)	0.512	2.401	22.991	288.219	3,372.749	39,919.563
selective goals failures	0%	12%	11%	9%	8%	7%
decrease in # goal failures	0%	26%	29%	30%	31%	32%

Table 5.2: Statistics gathered when running the Prolog version of the N-Queens benchmark. The upper portion is for chronological backtracking while the lower portion is for selective backtracking using the Culprit Pointer Method.

N	7	8	9	10	11	12	13
# solutions	40	92	352	724	2,680	14,200	73,712
# exhaustions	512	1,965	8,042	34,815	164,246	841,989	4,601,178
# calls to check	3,584	15,720	72,378	348,150	1,806,706	10,103,868	59,815,314
time (seconds)	0.426	1.620	7.857	41.881	237.750	1,430.163	9,127.861
# exhaustions	410	1,557	6,379	26,107	119,503	601,138	3,238,681
# selective exhaustions	63	249	1,044	5,110	24,392	124,458	676,754
# calls to check	3,129	13,593	62,957	293,459	1,482,732	8,162,135	47,762,689
time (seconds)	0.696	3.012	15.885	80.363	455.313	2,737.494	17,549.289
selective exhaustions	15%	16%	16%	20%	20%	20%	21%
decr. in # exhaustions	20%	21%	21%	25%	27%	29%	30%
decr. in # calls to check	13%	14%	13%	16%	18%	19%	20%

Table 5.3: Statistics gathered when running the CSP version of the N-Queens benchmark. The upper portion of the table is for chronological backtracking while the lower portion of the table is for selective backtracking using the CULPRIT POINTER METHOD.

Running a version of the N-Queens benchmark on an implementation of PROLOG which supported builtin arithmetic should afford less catastrophic results when the CULPRIT POINTER METHOD is added. In an attempt to quantify how such a hypothetical system would perform, the N-Queens problem was encoded as a constraint satisfaction problem and solved using both DEPTH-FIRST SEARCH and the CULPRIT POINTER METHOD as given in chapter 2. The results for finding all solutions for $N = 7, \ldots, 13$ are given in table 5.3. The upper half of the table gives the results for DEPTH-FIRST SEARCH while the lower half gives the results for selective backtracking. Here, the CULPRIT POINTER METHOD fairs somewhat better than the previous example. It offers a consistent reduction in both the number of exhaustion failures as well as the number of calls to CHECK but still require roughly twice the time as DEPTH-FIRST SEARCH. Much of the reduction in the number of calls to CHECK stems from the fact that since N-Queens is a binary CSP, no auxiliary calls to CHECK are needed to update the culprit pointers after each CHECK failure.

Chapter 6

Conclusion

Your heart's pride has deceived you;
you who dwell in the clefts of the rock,
whose habitation is high,
you say in your heart:
"Who shall bring me down to the ground?"
Though you fly as high as an eagle,
and make your nest among the stars,
from thence I will bring you down,
saith the Lord.

זדון לבך השיאך שכני בחגוי–סלע
מחם שבתו אמר בלבו מי יורידני ארץ:
אם–תגביה כנשר ואם–בין כוכבים שים קנך
משם אורידך נאם–ה':

עבדיה

Ovadia

This thesis has taken over three and a half years to complete. During that time, at least three fairly complex PROLOG implementations have been constructed which all have shared a common goal: to incorporate different search pruning techniques which would afford them better problem solving performance than the simple chronological backtracking associated with typical PROLOG implementations. Despite that long effort, no positive results have been obtained. Disappointingly, no provable negative results have been obtained either. Each of the three implementations has worked. Furthermore, it has been informally shown that no technique dominates any other one. For any given pair of techniques, whether they be chronological backtracking, one of the previously published techniques, or one of the techniques which I have developed, examples can be created where irrespective of the overhead involved, one technique performs arbitrarily better than the other. Unfortunately, these examples are contrived ones. The overhead required by each of these systems makes them completely impractical for any real use.

It seems that this thesis, as well as most of the related research into search-pruning techniques, is ill-motivated. Much effort has been expended into finding ever more sophisticated techniques which can prune the search space in certain rare cases. In most cases however, it is the simpler less powerful techniques—and even the brute force ones—which outperform the more sophisticated techniques in all but contrived examples. It is difficult to make a formal case for this. Examples can usually be created that allow any one given technique to outperform any other given technique by an arbitrary amount. And yet the painful experience gained by the repeated effort expended during this thesis is that it is nearly impossible to make any of the search pruning techniques, both well-known and novel, competitive with brute force approaches. A simple review of the benchmarks given in chapter 5 is evidence of that fact. This sounds inelegant but unfortunately appears to be the truth.

Much of the work I have done during this thesis project has not made it into this document. The algorithms underlying the implementations constructed prior to the one discussed here, as well as the in-depth comparison between the different search pruning techniques which have been studied will probably never be published. The effort needed to flesh out these ideas and understandings into a coherent and complete document is not worth the benefits. In its place, the next section gives a brief overview of some of the implementations I have built and the rationale that was used at the time to justify their construction. After discussing that history, I will conclude with a few open problems that this thesis has not succeeded in solving.

6.1 History

The original research objective for this thesis was to construct an implementation of Prolog which incorporated dependency-directed backtracking[83]. Dependency-directed backtracking, however, is not a well-defined term. Many authors have used the term dependency-directed backtracking to describe different collections of search pruning techniques. Realizing this discrepancy, efforts were then directed to merging a particular set of search pruning techniques into Prolog, namely those found in truth-maintenance systems[39, 38] such as Rup[61, 62, 63, 64] and Cake[80, 41]. Simply stated, these systems incorporate three primary search pruning techniques. Each of these techniques are orthogonal and attempt to solve a different search anomaly as discussed by de Kleer[29]. First, they attempt to avoid *futile backtracking* by analyzing the cause of failures and backtracking only to a choice which has contributed to the current failure. This choice may not be the most recent one made. This capability has been called *selective backtracking*. Second, they attempt to avoid *rediscovering the same contradiction* by saving inconsistent sets of choices. As each future choice is made, it is checked against that saved set and abandoned it if it can be shown to lead to failure. This capability has been called *lateral pruning* in the literature. Finally, truth-maintenance systems such as Rup attempt to avoid the problem of *incorrect ordering* by doing a limited form of dynamic variable reordering in the form of *constraint propagation*. Constraint propagation reduces to a simple variable reordering rule whereby choices with no remaining alternatives are pursued with highest priority and choices with only one remaining alternative are pursued with next highest priority. The combination of these three techniques namely, selective backtracking, lateral pruning, and constraint propagation constitute one definition of dependency-directed backtracking.

When truth-maintenance systems (TMSs) such as Rup are applied to search problems, they can be viewed as solving propositional satisfiability problems (SAT)[61]. This observation motivated the design of the first system constructed as part of this thesis project. This system operated by incrementally unraveling the Prolog program into an AND/OR-tree and converting this AND/OR-tree into conjunctive normal form in linear time via the addition of new propositional variables. This set of clauses was given to the TMS along with additional clauses created to represent the unification constraints annotating the atomic vertices of the AND/OR-tree. This system can be viewed as two coroutining processes. One process unraveled the Prolog program into larger and larger SAT problems, and the other attempted to solve these SAT problems using the dependency-directed backtracking techniques inherent in the TMS.

Although this implementation worked, it proved completely unusable. Even the smallest Prolog programs unraveled into vary large SAT problems. Conceptually, the inefficiencies inherent in this approach are apparent. First, while Depth-First Search need only represent the current AND-subtree at any point during the search, the TMS approach represents the entire AND/OR-tree, or at least some initial subtree of it. The difference in space requirements between the two approaches can be exponential. Second, the constructed SAT problem contains two kinds of clauses; those which represent the AND/OR-tree, and those which encode the unification constraints. Computing the clauses which encode the unification constraints is tantamount to computing all possible unification failures and finding the minimal sets of choices which entail those failures. As any actual search path generated by DEPTH-FIRST SEARCH will encounter only some of these failures, the TMS approach is doing the maximal amount of work possible, and far more work than DEPTH-FIRST SEARCH. Thus while the TMS component of the system was indeed using dependency-directed backtracking and supported a PROLOG implementation which exhibited all three search pruning characteristics namely, selective backtracking, lateral pruning, and constraint propagation, the cost of unraveling the AND/OR-tree into the SAT representation far outweighed the savings of search pruning.

Most of the overhead in the previous system was involved with constructing the SAT clauses that enforce the unification constraints. A second implementation was begun which utilized an equational reasoning system similar to that found in RUP as a basis for enforcing the unification constraints instead of the SAT representation of that information. Before that implementation was completed, a shortcoming was discovered in the dependency analysis performed by RUP. When RUP discovers an equational contradiction it produces a single nogood which in fact might not be minimal. Using a single non-minimal nogood can lead to less effective selective backtracking behavior than is possible if all (or at least the "right") minimal nogoods were computed and used.

This realization allowed an understanding of a major difference between two distinct approaches to search pruning, namely the dependency-directed backtracking/TMS approach and the variety of intelligent backtracking schemes proposed for PROLOG. One one hand, dependency-directed backtracking incorporates all three search pruning techniques: selective backtracking as well as lateral pruning and constraint propagation. The intelligent backtracking schemes proposed for PROLOG typically attempt only selective backtracking and do not attempt lateral pruning and constraint propagation. On the other hand, some of the intelligent backtracking schemes do more sophisticated and complete dependency analysis of unification failures than is done by the equational reasoning component of truth-maintenance systems. In the limit, all minimal nogoods are computed—not just one non-minimal one. In addition, some intelligent backtracking schemes such as the MULTIPLE NOGOOD algorithm produce new nogoods from previous ones by hyperresolution on exhaustion failures. This is not normally done by truth-maintenance systems and dependency-directed backtracking. While dependency-directed backtracking and intelligent backtracking schemes share much in common, each incorporates some sophistication which the other lacks thus making them incomparable.

It is possible to conceive of a system which combined all of the sophistication of both dependency-directed backtracking and intelligent backtracking. No attempt was made to construct such a system as it was felt that it would perform even more poorly than the previous system described. In fact, a result due to Wolfram[93] shows that there can be an exponential number of minimal nogoods produced at a unification failure so a straightforward implementation could be extremely inefficient. This raised an open question. If search pruning is limited to selective backtracking, does an algorithm exist which makes the same backtracking decisions as the MULTIPLE NOGOOD algorithm but which can make each such decision in polynomial time.

At first it was thought that such an algorithm was found. Although Wolfram shows that there can be an exponential number of nogoods produced at a unification failure, it might be possible to represent that set with a polynomial sized data structure. An implementation was constructed which built such a representation by analyzing paths in the unification graph. Unfortunately, it was later discovered that this representation was indeed polynomial sized only for unification problems without function symbols. Secondly, it was also discovered that although this representation could be constructed, manipulated, and queried in polynomial time for function-symbol-free PROLOG, an exponential number of unification failures could still lead to a single goal failure. The data structure representing the nogoods computed for that goal failure could be exponential in size.

This implementation, while correct, proved unusable as well. In addition, it failed at its goal of determining whether there exists a polynomial time equivalent to the MULTIPLE NOGOOD algorithm. It did show, however, that if such an algorithm does not exist, then something stronger than Wolfram's results will be needed to demonstrate this fact. Significant effort was expended without success in trying to prove that such an algorithm does not exist.

During this effort, the Culprit Pointer Method was developed. When the method was first developed and implemented, we did not realize that it was unsound when applied to selectively-nested exhaustions. Accordingly, as we were applying the method to all exhaustion failures, we thought that the Culprit Pointer Method was a polynomial equivalent to the Multiple Nogood algorithm. Only four months later was the unsoundness discovered by a fortuitous accident. The unsoundness did not show up in the N-Queens problem until $N = 11$. Even then there is only a slight discrepancy in the number of solutions found. Only because of a competition to see who could compute all solution to N-Queens faster, was the Culprit Pointer Method applied to larger problems and the unsoundness discovered. The disappointment upon discovering that the Culprit Pointer Method was not equivalent to the Multiple Nogood algorithm is indicative of the whole history of this project. There have been many false hopes and subsequent setbacks. We even had a soundness proof for the Culprit Pointer Method only to discover that the proof was indeed valid; it was the algorithm that did not meet the preconditions required by the theorem.

6.2 Future Work

This thesis leaves a number of questions unanswered. The most prominent of these is whether there exists a polynomial time algorithm which is equivalent to the Multiple Nogood algorithm. This seems like a difficult question to answer. A second question is whether the Culprit Pointer Method is equivalent to the Multiple Nogood algorithm for non-selectively-nested exhaustion failures. I conjecture that the answer to this question is yes but have not been able to prove this conjecture. The third question is how the Culprit Pointer Method compares formally with other techniques for solving constraint satisfaction problems. The comparison of two techniques can be formalized via the following definitions. A technique α *does not dominate* a technique β if an infinite class of problems can be demonstrated for which β requires time polynomial in the size of the problem but for which α requires time exponential in the size of the problem. Two techniques are *incomparable* if neither dominates the other. I believe that the Culprit Pointer method is incomparable to most other search pruning techniques for solving CSPs, including k-consistency and rearrangement search, but do not have complete proofs of this fact.

On one hand the above open questions are intriguing. On the other hand, they probably do not warrant significant effort to find their solution. When uninformed search can not be avoided, brute-force techniques, or the simplest search pruning techniques, seem to outperform more sophisticated ones. And in general, it is always better to avoid search whenever possible.

Bibliography

[1] Maurice Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. Report CW-16, Afdeling Toegepaste Wiskunde en Programmatie Katholieke Universiteit Leuven, Belgium, 1978.

[2] Maurice Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. In *Mathematical Logic in Computer Science*, Salotarjan, Hungary, 1978. Colloquia Mathematica Societatis Janos Bolyai.

[3] Maurice Bruynooghe. *Naar een betere beheersing van de uitvoering van programma's in the logika der Horn-uitdrukkingen*. PhD thesis, Afdeling Toegepaste Wiskunde en Programmatie Katholieke Universiteit Leuven, Belgium, 1979.

[4] Maurice Bruynooghe. Analysis of dependencies to improve the behaviour of logic programs. In W. Bibel and R. Kowalski, editors, *Proceedings of the Fifth International Conference on Automated Deduction*, pages 293–305, Les Arcs, France, 1980. Springer-Verlag. Also available as Lecture Notes in Computer Science #87.

[5] Maurice Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. In B. Dömölki and T. Gergely, editors, *Mathematical Logic in Computer Science*, pages 215–258. North-Holland, 1981. From Colloquium on Mathematical Logic in Programming, Salgotarjan, Hungary 1978.

[6] Maurice Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, February 1981.

[7] Maurice Bruynooghe and Luis Moniz Pereira. Revision of top-down logical reasoning through intelligent backtracking. CIUNL 8/81, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1981.

[8] Maurice Bruynooghe and Luis Moniz Pereira. Deduction revision by intelligent backtracking. In J. A. Campbell, editor, *Implementations of* PROLOG, chapter 3, pages 196–215. Ellis Horwood, Chichester, 1984.

[9] Jung-Herng Chang and Alvin M. Despain. Semi-intelligent backtracking of PROLOG based on static data dependency analysis. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 10–21, July 1985.

[10] Jung-Herng Chang, Alvin M. Despain, and Doug DeGroot. AND-parallelism of logic programs based on a static data dependency analysis. In *Proceedings of the 30th IEEE Computer Society International Conference*, pages 218–226, February 1985.

[11] David Chapman. Dependency-Directed Lisp. Unpublished manuscript received directly from author.

[12] T. Y. Chen, J-L. Lassez, and Graeme S. Port. Minimal non-unifiable subsets. Technical Report 84/16, Department of Computer Science, The University of Melborne, Austrailia, May 1985.

[13] T. Y. Chen, J-L. Lassez, and Graeme S. Port. Maximal unifiable subsets and minimal nonunifiable subsets. *Journal of New Generation Computing*, 4:133–152, 1986.

[14] C. Codognet and P. Codognet. *Formalizing and Implementing Intelligent Backtracking in Logic Programming*. PhD thesis, Mathématiques et Informatique, Univ. de Bordeaux I, 1988. forthcomming.

[15] C. Codognet, P. Codognet, and G. Filé. A very intelligent backtracking method for logic programs. Technical Report 8527, Mathématiques et Informatique, Univ. de Bordeaux I, 1985.

[16] C. Codognet, P. Codognet, and G. Filé. A very intelligent backtracking method for logic programs. In *Proceedings ESOP 1986*, pages 315–326, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1986. Springer-Verlag. Also available as Lecture Notes in Computer Science #213.

[17] C. Codognet, P. Codognet, and G. Filé. Depth-first intelligent backtracking. Technical report, Mathématiques et Informatique, Univ. de Bordeaux I, 1987.

[18] C. Codognet, P. Codognet, and G. Filé. Yet another intelligent backtracking method. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 447–465, Cambridge, Massachusetts and London, England, August 1988. The M. I. T. Press.

[19] Philip T. Cox. *Deduction Plans: A Graphical Proof Procedure for the First-Order Predicate Calculus*. PhD thesis, Universty of Waterloo, Ontario, Cannda, 1977. Also available as Research Report CS-77-28.

[20] Philip T. Cox. Locating the source of unification failure. In *Proceedings of the Second National Conference of Canadian Society for Computational Studies of Intelligence*, pages 20–29, Toronto, Ontario, 1978.

[21] Philip T. Cox. Representational economy in a mechanical theorem prover. In *Proceedings of the Fourth Workshop on Automated Deduction*, pages 122–128, Austin, Texas, 1979.

[22] Philip T. Cox. On determining the causes of nonunifiability. Report 23, Department of Computer Science, University of Auckland, New Zealand, 1981.

[23] Philip. T. Cox. Finding backtrack points for intelligent bactracking. In J. A. Campbell, editor, *Implementations of* Prolog, chapter 3, pages 216–233. Ellis Horwood, Chichester, 1984.

[24] Philip T. Cox. On determining the causes of nonunifiability. Technical Report 8601, Technical University of Nova Scotia, Halifax, 1986.

[25] Philip T. Cox. On determining the causes of nonunifiability. *Journal of Logic Programming*, 4:33–58, 1987.

[26] Philip T. Cox and Tomasz Pietrzykowski. Deduction plans: A basis for intelligent backtracking. Research Report CS-79-41, Universty of Waterloo, Ontario, Canada, December 1979.

[27] Philip T. Cox and Tomasz Pietrzykowski. Deduction plans: A basis for intelligent backtracking. *IEEE Transactions on Patern Analysis and Machine Intelligence*, 3(1):52–65, January 1981.

[28] Philip T. Cox and Tomasz Pietrzykowski. Surface deduction: A uniform mechanism for logic programming. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 220–227. IEEE Computer Society Technical Committee on Computer Languages, The Computer Society Press, July 1985. ISBN 0-8186-0636-3.

[29] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, March 1986.

[30] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28(2):163–196, March 1986.

[31] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28(2):197–224, March 1986.

[32] Johan de Kleer, Jon Doyle, Charles Rich, Guy Lewis Steele Jr., and Gerald Jay Sussman. AMORD: A deductive procedure system. A. I. Memo 435, M. I. T. Artificial Intelligence Laboratory, January 1978.

[33] Johan de Kleer, Jon Doyle, Guy Lewis Steele Jr., and Gerald Jay Sussman. Explicit control of reasoning. A. I. Memo 427, M. I. T. Artificial Intelligence Laboratory, June 1977.

[34] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 29–38, July 1985.

[35] W. Dilger and A. Janson. Unifikationsgraphen für intelligentes backtracking in deduktionssystemen. In *Proceedings of GWAI-83*, Dassel, Federal Republic of Germany, 1983.

[36] W. Dilger and A. Janson. Intelligent backtracking in deduction systems by means of extended unification graphs. *Journal of Automated Reasoning*, 2:43–62, 1986.

[37] W. Dilger and H. A. Schneider. ASSIP-T a theorem proving machine. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1194–1200, Los Angeles, 1985.

[38] Jon Doyle. A truth maintenaince system. *Artificial Intelligence*, 12:231–272, 1979.

[39] Jon Doyle. A truth maintenance system. A. I. Memo 521, M. I. T. Artificial Intelligence Laboratory, June 1979.

[40] Bernd-Jürgen Falkowski and Lothar Schmitz. A note on the queens' problem. *Information Processing Letters*, 23:39–46, July 1986.

[41] Yishai A. Feldman and Charles Rich. The interaction between truth maintenance, equality, and pattern-directed invocation: Issues of completeness and efficiency. Unpublished manuscript received directly from author, 1987.

[42] Nicholas S. Flann, Thomas G. Dietterich, and Dan R. Corpron. Forward chaining logic programming with the ATMS. In *Proceedings of AAAI-87*, pages 24–29, 95 First Street Los Altos, California 94022, July 1987. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc.

[43] D. R. Forster. GTP: A graph based theorem prover. Master's thesis, Universty of Waterloo, Ontario, Canada, 1982.

[44] K. Forsythe and S. Matwin. Implementation strategies for plan-based deduction. In R. E. Shostack, editor, *Proceedings of the Seventh International Conference on Automated Deduction*, Napa, California, 1984. Springer-Verlag. Also available as Lecture Notes in Computer Science #170.

[45] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\Re$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 675–703, Cambridge, Massachusetts and London, England, May 1987. The M. I. T. Press.

[46] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Unknown*, pages 111–119, 1987.

[47] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 196–218, Cambridge, Massachusetts and London, England, May 1987. The M. I. T. Press.

[48] Kenneth M. Kahn and M. Carlsson. How to implement PROLOG on a LISP machine. In J. A. Campbell, editor, *Implementations of* PROLOG, chapter 2, pages 117–134. Ellis Horwood, Chichester, 1984.

[49] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.

[50] Vipin Kumar and Yow-Jian Lin. An intelligent backtracking scheme for PROLOG. In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 406–414, September 1987.

[51] Vipin Kumar and Yow-Jian Lin. A data-dependency-based intelligent backtracking scheme for PROLOG. *Journal of Logic Programming*, 5:165–181, 1988.

[52] C. Lasserre and H. Gallaire. Controlling backtracking in horn clauses programming. In *Proceedings of the Logic Programming Workshop*, pages 286–292, 1980.

[53] C. Lasserre and H. Gallaire. Controlling backtracking in horn clauses programming. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 107–114. Academic Press, 1982.

[54] Peyyun Peggy Li and Alain J. Martin. The sync model: A parallel execution method for logic programming. In *Unknown*, pages 223–234, August 1986.

[55] Yow-Jian Lin and Vipin Kumar Clement Leung. An intelligent backtracking algorithm for parallel execution of logic programs. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 55–68, Berlin, Heidelberg, New York, London, Paris, Tokyo, July 1986. Springer-Verlag. Also available as Lecture Notes in Computer Science #225.

[56] Heikki Mannila and Esko Ukkonen. The set union problem with backtracking. In *Proceedings of the Thirteenth International Colloquium on Automata, Languages, and Programming*, Rennes, France, July 1986.

[57] Heikki Mannila and Esko Ukkonen. Timestamped term representation for implementing PROLOG. In *Unknown*, pages 159–165, August 1986.

[58] S. Matwin and Tomasz Pietrzykowski. Exponential improvement of exhaustive backtracking: Data structure and implementation. In *Proceedings of the Sixth International Conference on Automated Deduction*, pages 240–259. Springer-Verlag, 1982. Also available as Lecture Notes in Computer Science #138.

[59] S. Matwin and Tomasz Pietrzykowski. Intelligent backtracking for automating deduction. In *Proceedings of the Logic Programming Workshop*, pages 186–191, 1983.

[60] S. Matwin and Tomasz Pietrzykowski. Intelligent backtracking in plan-based deduction. *IEEE Transactions on Patern Analysis and Machine Intelligence*, 7(6):682–692, November 1985.

[61] David Allen McAllester. Solving SAT problems via dependency directed backtracking. Unpublished manuscript received directly from author.

[62] David Allen McAllester. A three valued truth maintenance system. A. I. Memo 473, M. I. T. Artificial Intelligence Laboratory, May 1978.

[63] David Allen McAllester. An outlook on truth maintenance. A. I. Memo 551, M. I. T. Artificial Intelligence Laboratory, August 1980.

[64] David Allen McAllester. Reasoning utility package user's manual version one. A. I. Memo 667, M. I. T. Artificial Intelligence Laboratory, April 1982.

[65] Luis Moniz Pereira. Backtracking intelligently in AND/OR trees. Technical report, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1979.

[66] Luis Moniz Pereira and Antonio Porto. Intelligent backtracking and sidetracking in horn clause programs—the implementation. CIUNL 13/79, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1979.

[67] Luis Moniz Pereira and Antonio Porto. Intelligent backtracking and sidetracking in horn clause programs—the theory. CIUNL 2/79, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1979.

[68] Luis Moniz Pereira and Antonio Porto. An interpreter of logic programs using selective backtracking. CIUNL 3/80, Dept. de Informática, Universidade Nova de Lisboa, Portugal, July 1980.

[69] Luis Moniz Pereira and Antonio Porto. An interpreter of logic programs using selective backtracking. In *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary, 1980.

[70] Luis Moniz Pereira and Antonio Porto. Selective backtracking at work. Technical report, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1980.

[71] Luis Moniz Pereira and Antonio Porto. Selective backtracking for logic programs. In W. Bibel and R. Kowalski, editors, *Proceedings of the Fifth International Conference on Automated Deduction*, pages 306–317, Les Arcs, France, 1980. Springer-Verlag. Also available as Lecture Notes in Computer Science #87.

[72] Luis Moniz Pereira and Antonio Porto. Selective backtracking for logic programs. CIUNL 1/80, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1980.

[73] Luis Moniz Pereira and Antonio Porto. Selective backtracking. UNL/FCT 11/80, Dept. de Informática, Universidade Nova de Lisboa, Portugal, 1981.

[74] Luis Moniz Pereira and Antonio Porto. Selective backtracking. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 107–114. Academic Press, 1982.

[75] Tomasz Pietrzykowski and S. Matwin. Exponential improvement of efficient backtracking: A strategy for plan-based deduction. In *Proceedings of the Sixth International Conference on Automated Deduction*, pages 223–239. Springer-Verlag, 1982. Also available as Lecture Notes in Computer Science #138.

[76] Graeme S. Port. Efficiently computing source information. Technical Report 88/3, Department of Computer Science, University of Melbourne, Australia, 1988. forthcomming.

[77] Graeme S. Port. *Parallel Logic Programming Using Source Information.* PhD thesis, Department of Computer Science, University of Melbourne, Australia, 1988. forthcomming.

[78] Graeme S. Port. A simple approach to finding the cause of non-unifiability. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 651–665, Cambridge, Massachusetts and London, England, August 1988. The M. I. T. Press.

[79] Matthias Reichling. A simplified solution of the N queens' problem. *Information Processing Letters*, 25:253–255, June 1987.

[80] Charles Rich. The layered architecture of a system for reasoning about programs. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 540–546, Los Angeles, 1985.

[81] Willem Rosiers and Maurice Bruynooghe. Empirical study of some constraint satisfaction algorithms. Report CW 50, Department of Computer Science, Katholieke Universiteit Leuven, July 1986.

[82] T. Sato. An algorithm for intelligent backtracking. In S. Goto et al., editors, *Proceedings of the RIMS Symposia of Software Science and Engineering*, pages 88–98. Springer-Verlag, 1983. Also available as Lecture Notes in Computer Science #147.

[83] Richard M. Stallman and Gerald Jay Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977. Also available as M. I. T. Artificial Intillegence Laboratory Memo 380.

[84] Guy Lewis Steele Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, Massachusetts Institute of Technology, August 1980. Also avilable as M. I. T. VLSI Memo 80-32 and as M. I. T. Artificial Inteligence Laboratory Technical Report 595.

[85] Harold S. Stone and Janice M. Stone. Efficient search techniques—an empirical study of the n-queens problem. RC 12057, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, August 1986. Also available as technical report 54343.

[86] Gerald Jay Sussman and Guy Lewis Steele Jr. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980. Also available as M. I. T. Artificial Intelligence Laboratory Memo 502A.

[87] J. Toh and K. Ramamohanrao. Failure directed backtracking. Technical Report 86/9, Department of Computer Science, University of Melbourne, Australia, 1986.

[88] P. Van Hentenryck and M. Dincbas. Forward checking in logic programming. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 229–256, Cambridge, Massachusetts and London, England, May 1987. The M. I. T. Press.

[89] P. E. Vasey. A logic-in-logic interpreter. Master's thesis, Imperial College of Science and Technology, University of London, 1980.

[90] David H. D. Warren. An abstract PROLOG instruction set. Technical Note 309, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025, October 1983.

[91] H. S. Wilf. An $O(1)$ expected time algorithm for the graph coloring problem. *Information Processing Letters*, 18:119–121, 1984.

[92] W. Winsborough. Semantically transparent selective reset for AND parallel interpreters based on the origin of failures. In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 134–152, September 1987.

[93] David A. Wolfram. Intractable unifiability problems and backtracking. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 107–121, Berlin, Heidelberg, New York, London, Paris, Tokyo, July 1986. Springer-Verlag. Also available as Lecture Notes in Computer Science #225.

[94] Nam Sung Woo and Kwang-Moo Choe. Selecting the backtrack literal in the AND/OR process model. In *Proceedings of the Third IEEE Symposium on Logic Programming*, pages 200–210, August 1986.

[95] Ramin D. Zabih. Dependency-directed backtracking in non-deterministic SCHEME. Master's thesis, Massachusetts Institute of Technology, January 1987.

[96] Ramin D. Zabih and David Allen McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artifical Intelligence*, pages 155–160, August 1988.

[97] Ramin D. Zabih, David Allen McAllester, and David Chapman. Non-deterministic LISP with dependency-directed backtracking. In *Proceedings of AAAI-87*, pages 59–64, 95 First Street Los Altos, California 94022, July 1987. American Association for Artificial Intelligence, Morgan Kaufmann Publishers, Inc.

[98] Ramin D. Zabih, David Allen McAllester, and David Chapman. Dependency-directed backtracking in non-deterministic LISP. *Artificial Intelligence*, 1988. Submitted for publication.