NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

THE UNIVERSITY OF ALBERTA

Intelligent Backtracking in Prolog

by

Brian Wong

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1989

ISBN 0-315-52792-7

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Brian Wong

TITLE OF THESIS: Intelligent Backtracking in Prolog

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1989

(Signed) .....

Permanent Address:
27D, 6/F, Nassau St.,
Mei Foo Sun Chuen,
Hong Kong.

Date: Dec 24, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Intelligent Backtracking in Prolog** submitted by **Brian Wong** in partial fulfillment of the requirements for the degree of **Master of Science.**

Jia-Huai You
Supervisor

Date: .....

To my parents

Abstract

Prolog's execution mechanism relies heavily on backtracking. When a failure is encountered, the interpreter returns to the most recent choice point and starts searching an untried alternative path from that point. This simple backtracking mechanism entails much futile computation and makes Prolog programs unduly expensive to run. In recent years, much research has been done on intelligent backtracking: the interpreter should be able to avoid backtracking to those choice points whose alternative solutions cannot possibly prevent the repetition of the failures which caused backtracking. In this dissertation we will describe the design and implementation of an intelligent backtracking scheme, called context resolution, for the Waterloo Unix Prolog interpreter. Another objective of this thesis is to study and compare some of the intelligent backtracking scheme proposed recently.

Acknowledgements

I would like to express my sincere gratitude towards Dr. J. You, my supervisor, for his guidance and support throughout this research. I would also like to thank my committee members: Dr. D. Szafron, Dr. R. Goebel and Dr. R.W. Toogood for their careful reading and constructive criticisms. I also thank Miss B. Lee who has contributed help in many ways, and Mr. C.S. Li for many valuable discussions. Finally, I should thank the Department of Computing Science for the excellent facilities and financial support which make this research possible.

Table of Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

1.1. Motivation and Objectives

Prolog, an acronym for **pro**gramming in **log**ic, is a programming language based on logic programming [Kowalski 74, Lloyd 84]. One of the main advantages offered by Prolog over conventional programming languages is its separation of program logic from the control component, which allows a programmer to compose a program just by describing the logical structures, rather than specifying explicitly how the computer is to go about solving it. During the search of solution(s) to a problem, Prolog's execution mechanism relies heavily on *backtracking*. When a failure is encountered, the interpreter returns to the most recent choice point and starts searching an untried alternative path from that point. This simple *chronological* backtracking mechanism entails much futile computation and makes Prolog programs unduly expensive to run. In recent years, much research has been done on *intelligent backtracking*: the interpreter should be able to avoid backtracking to those choice points that will definitely not contribute to the solution. However, the previously proposed schemes entail too much computational overhead [Cox 84, Bruynooghe&Pereira 84] and most of them have ignored the non-logical constructs, which are present in most real-life Prolog programs. These factors make the scope of their practical application rather limited.

In [You&Wang 88], the authors proposed an intelligent backtracking scheme — *context resolution*, which incorporates backtracking information into resolution in a natural way. In this thesis, we will present an improved version of context resolution

and lock into the design and implementation of this scheme. Other well-known schemes in this area will be studied and analyzed and their performance will be compared with context resolution.

1.2. Overview of Prolog

Prolog is based on the procedural interpretation of Horn clause predicate logic formulated by Kowalski [Kowalski 74]. The inference system of Horn clauses is linear resolution with selection function (SLD-resolution [Kowalski&Kuehner 71]) with unification of Horn clauses [Robinson 65, Lloyd 84]. The language was developed and first implemented by Roussel, Colmerauer et al. at the University of Aix-Marseille in 1972. Since then there has been a considerable proliferation of Prolog implementations, ranging from machine coded interpreters, to compilers, to special-purpose Prolog machines. The adoption of Prolog as the core language for the Japanese Fifth Generation Project in the fall of 1981 has further stimulated world-wide interest in Prolog and logic programming. However, owing to its deviation from typical von Neumann machine behavior, Prolog had the reputation of being difficult to be implemented efficiently. Fortunately, over the past decade, many techniques have been devised to improve the implementation of Prolog [Warren 77, Bruynooghe 82, Bruynooghe 82a, Mellish 82]. These contributions make Prolog implementations better understood.

1.3. Prolog Programs

1.3.1. Syntax

A Prolog program comprises a set of procedure declarations known as *clauses*. A clause is an implication of the form:

$$B \leftarrow A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_l;\ (l \geq 0)$$

where B and each A_i (called *predicates* or *atoms*) is an expression of the form:

$$r(T_1, T_2, ..., T_n)\ (n \geq 0)$$

where r is an n-adic relation called the *predicate symbol* and each argument T_i is called a *term*. A term is a variable or a structured term. A structured term is a construct of the form:

$$f(C_1, C_2, ..., C_k)\ (k \geq 0)$$

where f is called a k-ary *function symbol* and the C_i's are again terms. (Throughout this thesis, we will use functor(T) to denote the function symbol of a term T and arity(T) to denote the number of arguments in the term T. For example, functor($f(C_1,C_2,...,C_k)$) = f and arity($f(C_1,C_2,...,C_k)$) = k.) B is called the *head* of the clause and the A_i's constitute the *body* of the clause.

A goal statement has the form:

$$?\ D_1 \,\&\, D_2 \,\&\, \cdots \,\&\, D_m;\ (m \geq 1)$$

and the D_i's are called *subgoals*. Throughout this thesis, the syntactic conventions of Waterloo Unix Prolog (WUP) [Cheng 84] will be followed. A function or predicate symbol starts with a lower case letter. A variable starts with an upper case letter. A list is a structured term which is constructed using the binary concatenate function '*concat*', for example, a list with three elements a, b and c is represented as $concat(a, concat(b, c))$. A list is more commonly represented by the infix notation $[A\,|B\,]$ where A is called the head of the list and B the body of the list, and they are again terms. The empty list is represented by [] and the don't care variable is denoted by _. The list example above is also represented as $[a\,|[b\,|c\,]]$. In addition, constants are

considered 0-ary function symbols. A sample Prolog program is shown in Program 1.1.

---

father (*john* , *tom*);
father (*tom* , *sam*);

grandfather (*X* , *Z*) ← *father* (*X* ,*Y*) & *father* (*Y* , *Z*);

? *grandfather* (*john* , *Who*);

Program 1.1: An Example of a Prolog Program

1.3.2. Program Execution

1.3.2.1. Unification

Unification in a Prolog program plays the role of parameter passing and assignments in traditional languages. During the execution of a logic program, the *unifiability* of two atoms, F and G, where $F = P_1(t_1, t_2, \ldots, t_n)$, $G = P_2(l_1, l_2, \ldots, l_m)$, is determined by the following steps, which are based on Robinson's Unification Algorithm [Robinson 65]. Note that the variables in G are renamed before the unification so that they are different from the variables in F.

1. If the predicate symbols are different (that is, $P_1 \neq P_2$) or the number of arguments in F is not equal to the number of arguments in G (that is, $n \neq m$), then the unification of F and G fails.
2. If each argument in F is unifiable with each corresponding argument in G then the algorithm succeeds, otherwise F and G are not unifiable.

In addition, the unifiability of two terms t and l is conducted by the procedure *naive_unify* (Figure 1.1), which is modified from a unification algorithm given in [Roberts 77].

```
naive_unify(t, l) {

  dereference(t);
  dereference(l);

  occurs_check(t,l);

  if (t is a variable) {
    θ ← θ ∪ {t / l};
    return(SUCCESS);
  }
  else if (l is a variable) {
       θ ← θ ∪ {l / t};
       return(SUCCESS);
     }
     else {
       if (functor(t) ≠ functor(l) or arity(t) ≠ arity(l))
         return(FAILURE);

       for i in 1 to arity(t) do {
         t_i ← i-th argument of t;
         l_i ← i-th argument of l;
         if (not naive_unify(t_i, l_i))
           return(FAILURE);
       }
       return(SUCCESS);
     }
}
```

Figure 1.1: The Ordinary Unification Algorithm

The global variable θ (called a *substitution*) is the set of all variable assignments taken place during the unification of F and G such that no variable will be assigned more than one term. θ is initially set to the empty set before the unification routine is entered. When a variable X is bound to a term t, written X/t, this binding is recorded in θ. This record of the bindings is kept so that the bindings can be undone during backtracking. When θ is applied to a predicate P, written $P\theta$, all the variables in P are

replaced by the terms which θ recorded. For example, when P = *father* (X, Y) and θ = $\{X/john, Y/tom\}$, $P\theta$ = *father* $(john, tom)$.

The two terms are first dereferenced before unification is attempted. This means that if a term is a bound variable, it is replaced by its value which is looked up in θ. The routine *occurs_check* determines if t occurs in l or vice versa to prevent self referencing infinite structures. Like many existing Prolog systems, our interpreter will not perform occurs check since this checking is expensive to perform. Thus this operation will be omitted from all subsequent algorithms in this thesis.

If one of the terms is an unbound variable, then the algorithm binds this free variable to the other term. Otherwise both terms are structured terms. If they have different function symbols or arities, the unification algorithm fails with no output. This means that the two terms t and l cannot be unified. In the *for* loop, the algorithm tries to unify each argument in t with the corresponding argument in l. If any one of these pairs of arguments cannot be unified, then the unification of t and l fails. If each pair of arguments can be unified, *SUCCESS* is returned and θ is the *most general unifier* [Lloyd 84] of the two terms such that $t\theta = l\theta$. Otherwise, *FAILURE* is returned which indicates that the two terms cannot be unified. An example of unification is shown in Figure 1.2.

The following definition is useful later in the thesis:

Definition: [Lloyd 84]

Let $\theta = \{u_1/s_1, ..., u_m/s_m\}$ and $\sigma = \{v_1/t_1, ..., v_n/t_n\}$ be substitutions. Then the composition $\theta\cdot\sigma$ of θ and σ is the substitution obtained from the set $\{u_1/s_1\sigma, ..., u_m/s_m\sigma, v_1/t_1, ..., v_n/t_n\}$ by deleting any binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, ..., u_m\}$. □

It can be shown that for any predicate P and substitutions θ and σ, $(P\theta)\sigma = P(\theta\cdot\sigma)$.

t: [1|W]
l: [X|[Y|Z]]

θ = {X/1, W/[Y|Z]}

Figure 1.2: An Example of Unification

1.3.2.2. Interpreter Cycle

Consider a logic program with a goal statement G:

$$? \ A_1 \ \& \ A_2 \ \& \ \cdots \ \& \ A_m ; \ (m \geq 1).$$

During the k-th step of program execution, a subgoal A_i $(1 \leq i \leq m)$ is selected for activation. Then from the set of available candidate clauses (that is, the set of clauses whose heads have the same predicate symbol and arity as A_i), a clause $B \leftarrow B_1 \ \& \ B_2 \ \& \ \cdots \ \& \ B_n ;$ is selected. The most general unifier θ_k is then determined for A_i and B. The new goal statement (called a *derived goal*) in the k+1-th step of execution becomes G':

$$? \ (A_1 \ \& \ A_2 \ \& \ \cdots \ \& \ A_{i-1} \ \& \ B_1 \ \& \ \cdots \ \& \ B_n \ \& \ A_{i+1} \ \& \ \cdots \ \& \ A_m)\theta_k$$

which is the new state of the computation. This cycle is repeated until we reach an empty goal statement, that is, all the subgoals have been solved. The set of variable assignments obtained by applying the composition $\theta_1 \bullet \theta_2 \bullet \cdots \bullet \theta_l$ of all the most general unifiers $\theta_1, \theta_2, \cdots, \theta_l$ of this computation to the variables in G is the output of the program. For example, the output of Program 1.1 is {*Who/sam*}.

If A fails to unify with the head of a candidate clause, the head of the next avail-

able clause is tried. This is called *shallow backtracking*. If A_i fails to unify with all the heads of the candidate clauses, then a *deep backtracking* occurs. This corresponds to resetting a previous solved subgoal (the *backtrack subgoal*) and undoing all the variable bindings since that subgoal is activated. Intelligent backtracking is the process of choosing this backtrack subgoal when a deep backtracking occurs.

A subgoal selection rule is often called a *computation rule*. The *standard computation rule* always selects the leftmost unsolved subgoal in the current goal. The selection of responding clause to the active subgoal is governed by the *search rule*. The *standard search rule* sequentially chooses responding clause in the order of their appearance in the program text. Like most conventional Prolog implementations, the standard computation rule and standard search rule are adopted in this thesis.

1.3.3. Search Tree

The execution of a logic program is best explained by an *OR*-tree, also known as a *search tree*. The goal is at the root. At each node, a subgoal is selected for activation. A downward branch is extended from that node for each responding clause to the active subgoal. A node in the tree is *deterministic* if it has only one downward branch, while a node with more than one branch is a *non-deterministic* node. A node with no branch is called a *leaf*. There are three possible outcomes for the execution:

1. when an empty goal is reached, the computation is successful and a solution is found. This is represented by a ○ at a leaf of the search tree. This leaf is called a success leaf.
2. if a clause fail to unify with the active subgoal, the computation fails. A ∅ at the leaf of a search tree is used to denote failure. This leaf is called a failure leaf.
3. when the computation is trapped into an infinite derivation, it is represented by an infinite branch in the search tree.

1.3.4. Non-Determinism

The interpreter cycle presented in 1.3.2.2 is *non-deterministic* in two ways. The first non-determinism results from the fact that any subgoal can be selected for activation during a cycle. The second non-determinism arises since more than one clause will respond to the currently activated subgoal. Each of these clauses leads to an alternative branch in the search tree. Prolog relies on backtracking to explore all these branches.

The search tree is typically searched depth-first. That is, the current node is derived until the interpreter reaches a success leaf or a failure leaf or is trapped into an infinite computation. New computations are developed only when the current one terminates, that is, when it reaches ○ or Ø. In this way, the whole computation space is explored systematically. If a success leaf is reached, a solution is reported. If a failure leaf is reached, the interpreter tries the next available candidate clause. If all the candidate clauses fail to unify with the current subgoal, the interpreter backtracks upward through the path just developed to the most recent node with unexplored branch(es). If no further unexplored branch exists then the execution fails. Otherwise the interpreter searches the first alternative branch and explores in a depth-first fashion again. An example of a search tree for Program 1.2 is shown in Figure 1.3.

$p(a)$;
$p(b)$;

$q(m)$;
$q(W) \leftarrow q(W)$;

$r(b)$;

$? \; p(X) \; \& \; q(Y) \; \& \; r(X)$;

Program 1.2

The selection of a subgoal to be activated and the responding clause greatly determine the outcome and efficiency of a computation. Unless an intelligent scheduler is devised or a parallel system is employed which explores all the available branches in parallel, a user must decide upon the textual ordering of the declarations and the subgoals to make sure that the computation of his program will terminate efficiently (will not go into an infinite derivation).

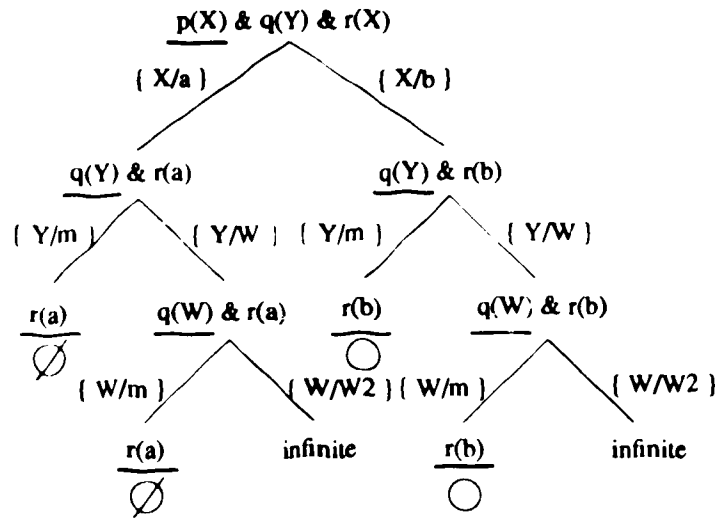


Figure 1.3: The Search Tree for Program 1.2
(Active subgoal at each node is underlined.)

1.4. Previous Work

Most intelligent backtracking schemes can be broadly classified into two categories. The backtracking schemes in the first category are based on unification failure analyses. Cox [Cox 84, Cox 87] and Bruynooghe and Pereira [Bruynooghe&Pereira 84] use *deduction trees* to analyze the possibility of unification of a subtree. Cox's algorithm is based on finding the maximal subtree such that unification is possible. Bruynooghe-Pereira's algorithm is based on finding the minimal subtree such that unification is impossible. Once the subtree has been found, the nodes that belong to a non-unifiable subtree should be avoided. The method based on *modifying goals* by Pereira and Porto

[Pereira&Porto 82] is another example of unification analysis. However, as indicated in [Wolfram 86], the computation of all maximal unifiable subsets or minimal non-unifiable subsets is intractable. Even Cox confessed that these methods can only be used in some restricted applications because of the intensive computation involved. Based on the method by [Bruynooghe&Pereira 84], a more efficient intelligent backtracking algorithm is presented in [Codognet,Codognet&Filè 88]. This scheme exhibits substantial speed up and can handle most system predicates intelligently.

The backtracking schemes in the second category are based on variable consumption analysis. This approach analyzes the generator-consumer dependency relationship of variables appearing in a clause either statically (during compilation) [Chang&Despain 85] and/or dynamically (during program execution) [Kumar&Lin 88]. Using this approach, a better backtrack point can be discovered from variable sharing information without performing a possibly intensive failure analysis. However, generator-consumer analysis, which is based on the computations of graphs, can also be expensive. In particular, when bindings introduced by unification include non-ground terms, reconstruction of the dependency graphs is often needed [Lin,Kumar&Leung 86]. The ground binding restriction was later eliminated in an improved scheme [Kumar&Lin 88] by a technique called *tagging*. Their simulation results indicate that this scheme entails a small amount of overhead among several known schemes for a number of typical programs. However, a problem with the generator-consumer approach is that it sometimes fails to locate the cause of failure *directly*, and is thus sometimes less intelligent than desired.

1.5. Organization of the Thesis

In Chapter two, we will look at the resolution-based procedure — context resolution. In Chapter three, we will examine some of the design problems encountered, and a modified unification algorithm will be proposed. Then we will present the implementation details of the intelligent backtracking scheme in Chapter four. In Chapter five, our scheme will be compared to some other well-known schemes. Some execution statistics of our scheme, together with the performance results of the other schemes will be presented in Chapter six. Finally, a summary of the thesis and some suggestions for extensions and future work will be given in Chapter seven.

Chapter 2

Context Resolution

In this chapter, we will introduce context resolution, an intelligent backtracking scheme for Prolog programs. An example is first given to illustrate the inefficiency of a naive Prolog interpreter. After the definitions for context and context term are given, the unification algorithms and resolution procedure which incorporate context information are presented. Finally, the data structure which supports context resolution and the two types of failed subgoals are discussed.

2.1. Introduction

In the traditional implementation of a Prolog interpreter, the backtracking scheme is uninformed. When a derivation fails, the interpreter backtracks to the most recent choice point in the search tree. Although the conventional scheme has little overhead and greatly simplifies the needed run time structures [Bruynooghe 82], it can easily lead to thrashing. The interpreter cannot detect that some earlier branches in the search tree will never lead to solutions and those subtrees will be blindly searched. The same failure will come up many times before the correct backtrack point which leads to a solution is found. A number of intelligent backtracking schemes have been developed to avoid this kind of redundant backtracking. While those schemes which are based on unification failure analyses seem difficult to implement on a conventional Prolog system, the other schemes which are based on variable sharing information have a lower degree of intelligence†. Context resolution is a scheme which combines ease of implementation and a high degree of intelligence.

† A backtracking scheme is said to be more intelligent if it can eliminate more irrelevant backtrack

$p(a)$;
$p(b)$;

$q(m)$;
$q(n)$;
$q(o)$;

$r(b)$;

$? \; p(X) \; \& \; q(Y) \; \& \; r(X)$;

Program 2.1: A Program to Illustrate the Problem with a Naive Interpreter

Before we formally introduce context resolution, we look at the efficiency problem with a naive interpreter. Consider Program 2.1 where the first time the interpreter needs to backtrack is at the subgoal $\leftarrow r(a)$; since it cannot unify with $r(b)$. The interpreter will backtrack to q and get another binding for Y. The same failure will be encountered again and this whole cycle will repeat. Eventually the interpreter exhausts all the declarations for q, then it backtracks to p, which will give a binding that leads to a solution. This entails a lot of redundant computation. The situation is even worse when there are many alternative clauses for q, or when there are a lot of subgoals between the predicate where a term is introduced and the failed subgoal which contains that term. Although a naive interpreter will eventually find the right backtrack point, it will, in the mean time, search irrelevant parts of the search tree. An intelligent interpreter should be capable of avoiding backtracking into those subgoals which definitely lead to failure later in the derivation.

points. That is, it can compute a backtrack point which is further away from the failed subgoal.

2.2. Context Resolution

Context resolution is an intelligent backtracking method. To incorporate the information required for intelligent backtracking, the intelligent interpreter keeps track of some extra information with each term. The idea is to include a *context* during the unification process and resolution procedure. Every term introduced in the current subgoal is associated with a context, which is a unique number indicating the current subgoal where this term is introduced. This term, together with the context information, will be used in the derivation of new subgoals and thus is propagated to the other parts of the search tree. Later in the derivation, if a unification failure is detected, the context which is associated with the term involved in the failure is returned — piece of information can directly indicate the backtrack subgoal where the failu — ssibly cured. We will use the following definitions throughout the rest of this th

Definition:

A *context* is a number *#i* associated with a term t, which indicates the subgoal G_i † at which t is introduced. □

Definition:

If *#i* is a context, a context term is defined as follows:

- An ordinary term is a context term.
- If f is an n-ary function symbol and $t_1, t_2, \cdots, t_n$ are context terms, then $f(t_1, t_2, \cdots, t_n)$ is a context term.
- If t is a context term, then [t,*#i*] is a context term.

A context term is called a *context variable* if it is either a variable, or a variable associated with a context, or inductively, a context variable associated with a context.

A substitution is now defined as a mapping from context variables to context terms, extended to an endomorphism of the set of context terms. □

---

† The initial goal is G_0. After k inference steps, the derived goal is G_k.

Definition:

We use *context*(f) to denote the list of contexts associated with the function symbol f. In a possibly nested context term of the form $[[...[f(t_1, t_2, ..., t_{arity(f)}), \#c_1], \#c_2], ..., \#c_m]$, we define:

$\text{list}(f) = \{\#c_1, \#c_2, ..., \#c_m\}$,

$context(f) = \text{list}(f)$,

$context(t_i) = \text{list}(t_i) \cup \text{list}(f)$.

For an ordinary term t (that is, t has not been associated with any contexts), $context(t) = \varnothing$. □

Intuitively, the contexts associated with a function symbol f are chosen without considering any contexts which are associated with the arguments of f. However, those contexts which are associated with the function symbols which take f as one of their arguments have to be considered. The implication of a nested context term is that a symbol therein has been "introduced" more than once, possibly along with some other symbols. Each of the contexts indicates the subgoal where the symbol was introduced.

2.2.1. Context Unification

The unification mechanism which incorporates context information is called *context unification*. Note that context information will not affect the outcome of a unification. When a unification is successful and a context variable is bound to a context term, the current context is introduced into the context term. When a subgoal fails to unify with the head of a candidate clause, the largest context associated with the context term which causes the unification failure is returned. The largest context is chosen since it indicates the subgoal where the function symbol which causes the unification failure was *last* introduced.

```
unify_I(t, l) {

  dereference(t);
  dereference(l);

  if (t is a context variable) {
    Θ ← Θ ∪ {t / [l,#i]};
    return(SUCCESS);
  }
  else if (l is a context variable) {
        Θ ← Θ ∪ {l / [t,#i]};
        return(SUCCESS);
      }
      else {
        if (functor(t) ≠ functor(l) or arity(t) ≠ arity(l)) {

          if (context(t) ≠ ∅ or context(l) ≠ ∅)
            failure_context ← max(context(t) ∪ context(l));
          else
            failure_context ← context associated with the context variable
                                    which caused the disagreement;

          return(FAILURE);
        }
        for i in 1 to arity(t) do {
          t_i ← i-th argument of t;
          l_i ← i-th argument of l;
          if (not unify_I(t_i, l_i))
            return(FAILURE);
        }
        return(SUCCESS);
      }
}
```

Figure 2.1: Unification Algorithm I

The context unification mechanism is represented by the procedure *unify_1* in Figure 2.1. The variable *failure_context* will store the largest of all the contexts associated with a pair of function symbols which causes a failure during unification, or the largest context associated with the context variable which causes a failure. Since occurs check is omitted, disagreement of terms can only arise because of symbol disagreement. *failure_context* is initialized to $\perp$, where $\perp$ is considered smaller than any other contexts which it may encountered during the execution of a Prolog program. The procedure *max* will return the maximum number from its argument, which is a list of contexts. We assume two context terms, t in the current subgoal G_i and l in the head of a candidate clause, are to be unified, where l is a standard term without any context. It is assumed that the variables in these two terms are renamed so that the two given terms do not share common variables.

Note the similarity of the mechanism between context unification and the conventional unification mechanism given in Section 1.3.2.1. Thus context information can be naturally incorporated into the conventional unification mechanism without much overhead. The main differences are the introduction of the current context in the unification step and the retrieval of the failure context for backtracking purpose.

In the algorithm, when none of the conflicting function symbols are associated with any context, the context associated with the context variable that caused the disagreement is returned. As an example, suppose the current context is #5, and $p([X,\#1],[X,\#1])$ and $p(a,b)$ are to be unified, the context associated with X (#1) is returned since X caused the disagreement between a and b.

Several examples on context unifications are now given. Assume that the current goal is G_5:

1. $t = p([f([X,\#2]),\#4])$

   $l = p(f(a))$

   SUCCESS is returned, $\Theta = \{[X,\#2]/[a,\#5]\}$

2. $t = p([f([b,\#2]),\#4])$

   $l = p(f(a))$

   FAILURE is returned, failure_context = $max(\text{context}(b) \cup \text{context}(a)) = max(\{\#2,\#4\} \cup \emptyset) = \#4$

3. $t = p([X,\#1],[X,\#1])$

   $l = p(a,b)$

   FAILURE is returned, failure_context = #1

You and Wang also presented a second context unification algorithm (*unify_II*) which is shown in Figure 2.2. The first context unification algorithm (*unify_I*) stops when the first failure is encountered and a failure context is returned. In *unify_II*, the unification routine will try to collect *all* the failure contexts which are associated with different function symbols which cause unification failures, and from them the *smallest* context is chosen for backtracking. It can be easily seen that, when compared to the original algorithm, this approach requires extra work should unification fail. The idea is to backtrack further up the search tree whenever a unification failure occurs. As an example, suppose we want to unify the subgoal $p([d,\#i],[e,\#j],[f,\#k])$ with $p(a,b,c)$, where $p(a,b,c)$ is the only available candidate clause. The first algorithm, upon reaching the first pair of terms which cannot be unified, (that is, a and $[d,\#i]$) will report failure immediately and return the failure context $\#i$. In the second algorithm, the interpreter will try to gather all the failure contexts, in this case, $\{\#i, \#j, \#k\}$, and return the smallest for backtracking. Now if context $\#k$ happens to be the smallest, the interpreter will backtrack further up the search tree, bypassing the nodes pointed to by $\#i$ and $\#j$. While *unify_II* may sometimes give a better backtrack point, too much overhead is involved to

justify its usefulness. We have chosen *Unify_I* for our implementation.

```
unify_II(t, l) {

  dereference(t);
  dereference(l);

  if (t is a context variable) {
    Θ ← Θ ∪ {t / [l,#i]};
    return(SUCCESS);
  }
  else if (l is a context variable) {
      Θ ← Θ ∪ {l / [t,#i]};
      return(SUCCESS);
    }
    else {
      if (functor(t) ≠ functor(l) or arity(t) ≠ arity(l)) {

        if (context(t) ≠ ∅ or context(l) ≠ ∅)
          max_context ← max(context(t) ∪ context(l));
        else
          max_context ← context associated with the context variable
                                  which caused the disagreement;

        failure_context ← min(max_context, failure_context);

        return(FAILURE);
      }
      flag ← SUCCESS;
      for i in 1 to arity(t) do {
        t_i ← i-th argument of t;
        l_i ← i-th argument of l;
        if (not unify_II(t_i, l_i))
          flag ← FAILURE;
      }
      return(flag);
    }
}
```

Figure 2.2: Unification Algorithm II

2.2.2. Context Resolution

Context resolution is like standard resolution, except that it uses context unification and the context information is carried along the derivations. The procedure *resolve* described below, which is modified from the resolution procedure given in [Lloyd 84], gives the actions to be performed in a resolving step. Note that the leftmost computation rule is used.

resolve(G_i, Θ)

> G_i is of the form $\leftarrow l_1 \,\&\, \cdots \,\&\, l_k$, and Θ is the most general unifier obtained by context unification for l_1 and p, p being the head of a clause $p \leftarrow q_1 \,\&\, \cdots \,\&\, q_m$. Since the terms in the clause $p \leftarrow q_1 \,\&\, \cdots \,\&\, q_m$ are also introduced in the current resolving step, they will be associated with the current context $\#i$. Let q'_j, $1 \le j \le m$, denotes the subgoal obtained from $q_j\Theta$ by associating the terms in q_j with the current context. The next goal derived is:
>
> G_{i+1}: $\leftarrow q'_1 \,\&\, \cdots \,\&\, q'_m \,\&\, l_2\Theta \,\&\, \cdots \,\&\, l_k\Theta$ □

As an example, the goal G_3 is derived from G_2 by resolving with $p(f(Y), b) \leftarrow r(Y, c)$.

G_2: $\leftarrow p(f([a, \#1]), X) \,\&\, q(X)$;

G_3: $\leftarrow r([[a, \#1], \#2], [c, \#2]) \,\&\, q([b, \#2])$;

2.3. The Intelligent Backtracking Mechanism with a *B*-list

A global data structure called a *B*-list is maintained through program execution to keep track of potential backtrack points. When a unification failure occurs and a failure context is returned, the interpreter will backtrack directly to the backtrack point specified by the failure context if no more candidate clauses are available for the current goal. If other candidate clauses are available, the current failure context is temporarily stored into the *B*-list, and another clause is tried. After all the available clauses are tried,

unifiable clause is found, the largest context in the *B*-list is chosen for backtracking. The operations required so far are insertion, retrieval and sorting. As we will see in the next chapter, by using a good data structure for the *B*-list and making use of the run time properties of a Prolog interpreter, the overhead of insertion and retrieval will be greatly reduced and the sorting operation can be eliminated.

2.4. Direct and Indirect Failures

In [You&Wang 88], two types of failed subgoals are discussed. A *direct failed* subgoal is a subgoal which fails to unify with the head of any candidate clause. An *indirect failed* subgoal is one which unifies with a candidate clause at least once, but finally fails. The ideas are very similar to Type-I and Type-II failures in [Chang&Despain 85] and [Kumar&Lin 88].

```
p(a,a);
p(a,b);

q(a);
q(b);

r(a,b);
r(a,c);

s(a,c);
s(b,b);

t(a,d);
t(b,b);

? p(X,Y) & q(Y) & r(X,Z) & s(Y,Z) & t(Y,Z);
```

Program 2.2: An Example to Illustrate Direct and Indirect Failures

---

When a subgoal fails directly, the failure context returned indicates the node to which backtracking is done. If backtracking is done to an indirect failed subgoal, however, all the contexts in the failed subgoal are obtained and the largest one is chosen for backtracking. This step is implemented as *Get_All_Contexts* (g_i) [You&Wang 88] where g_i is an indirect failed subgoal. In terms of a search tree, a direct failed subgoal is a node whose children are all failure leaves, while an indirect failed subgoal is a node in the tree with no success leaf and at least one child which is not a leaf.

Consider Program 2.2 and its partial execution trace in Figure 2.3. At step 4, when $s([a,\#0],[b,\#2])$ fails to unify with all the candidate clauses, we have a direct failed subgoal. The context returned is #2, thus backtracking is done to #2 in the next step. A conventional Prolog interpreter will backtrack to #3 since it is the most recent choice point with untried alternative clause. At step 7, $t([a,\#0],[c,\#2])$ fails to unify with the available clauses and the context returned is #2. At step 8, backtracking is done to #2 and we run out of clauses for r. This is an indirect failed subgoal since $r([a,\#0],Z)$ has succeeded before. Now we get all the contexts in the subgoal $r([a,\#0],Z)$ (in this case the only context returned is #0), and we backtrack to the largest context (#0). Notice that the backtrack point #1 is also skipped over. In the next step, the interpreter will try to unify $p(X,Y)$ with the next available clause for p, $p(a,b)$. After a few more steps, the interpreter will come up with the answer $\{X/a, Y/b, Z/b\}$. The complete search tree for Program 2.2 is shown in Figure 2.4. For easy reference, the step numbers (the circled number in the diagram) are put beside the variable bindings created at each step. The indirect failed node $r([a,\#0],Z)$ & $s([a,\#0],Z)$ & $t([a,\#0],Z)$ is surrounded by a rectangle.

A Partial execution trace:

step: context:

1	#0:	p(X,Y) & q(Y) & r(X,Z) & s(Y,Z) & t(Y,Z); { Y/[a,#0] , Y/[a,#0] }
2	#1:	q([a,#0]) & r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z);
3	#2:	r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z); { Z/[b,#2] }
4	#3:	s([a,#0],[b,#2]) & t([a,#0],[b,#2]); direct failure, goto #
5	#2:	r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z); { Z/[c,#2] }
6	#3:	s([a,#0],[c,#2]) & t([a,#0],[c,#2]);
7	#4:	t([a,#0],[c,#2]); direct failure, goto #2
8	#2:	r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z); indirect failure

.
.
.

Figure 2.3: A Partial Trace for Program 2.2

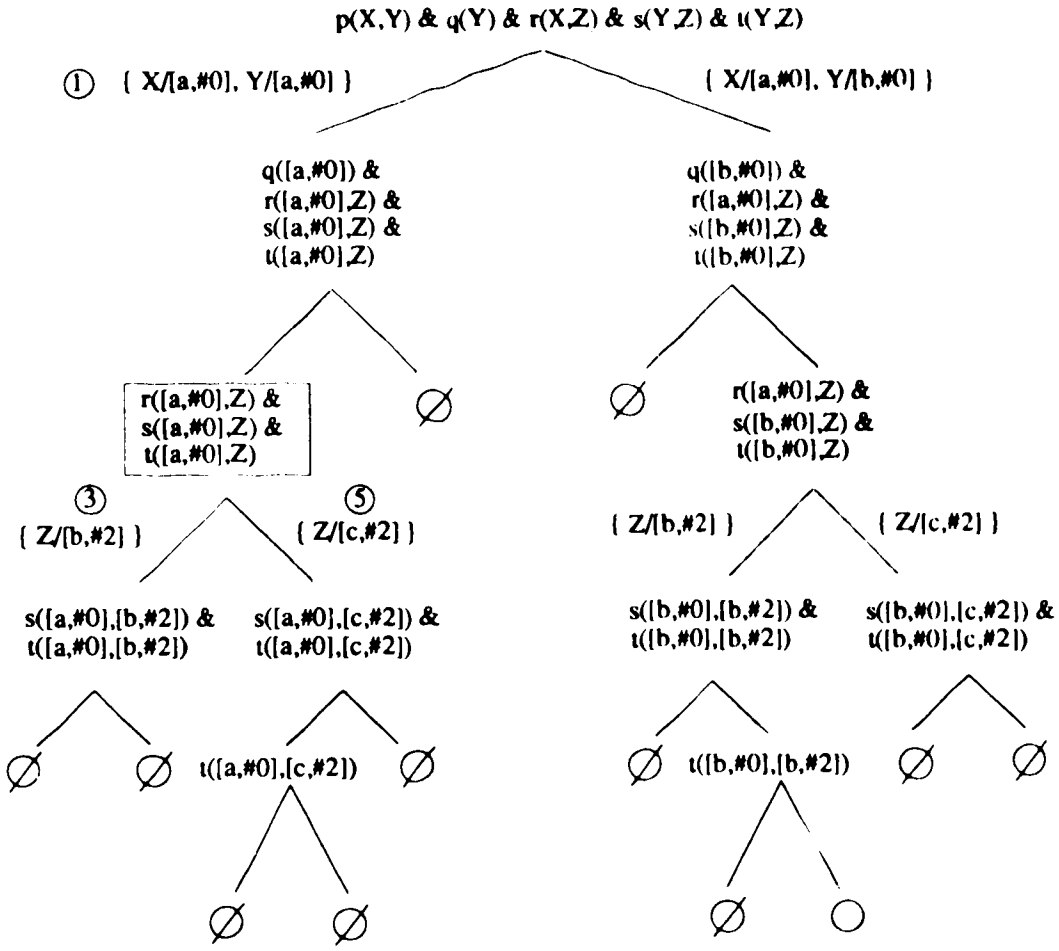


Figure 2.4: The Search Tree for Program 2.2

2.5. Summary

In this chapter, we have seen why naive backtracking can be very inefficient. Context resolution is introduced which incorporates backtracking information (contexts) into the unification and resolution procedures. When a context variable is bound to a context term, the context which indicates the current subgoal where the context term is

introduced is inserted into the context term. In doing so, the context information is embedded into the newly derived goal and will be propagated to further derived goals. Now when a subgoal fails to unify with the head of a candidate clause, the largest context associated with the context term which causes the unification failure is returned. This context indicates the subgoal where the current failure can be possibly cured. In addition to the modified resolution procedure, two context unification algorithms with different intelligence and overhead are introduced. In addition, two kinds of failed subgoals are presented which require different treatment from an intelligent interpreter.

Chapter 3

Design of the Implementation

This chapter is concerned with the details of the design of the implementation of context resolution. The problems in the handling of indirect failure are discussed. Then a revised unification scheme is proposed, which is more intelligent and more efficient than the original scheme. This new unification algorithm will be incorporated into a Prover algorithm, which shows the major steps taken by an interpreter incorporated with intelligent backtracking. Finally the methods for the triggering of intelligent backtracking will be presented.

3.1. Handling of Indirect Failure

When a subgoal fails indirectly, all the contexts in the arguments in the failed current subgoal must be examined to extract the most recent context for backtracking. Examining just the outermost context is not adequate because the outer context may be smaller than the inner context. This can happen when a variable is instantiated to a structured term with an uninstantiated variable, which is *later* instantiated. For example, X is first bound to $[f(Y),\#3]$ and Y is later bound to $[a,\#6]$. Thus X is now bound to $[f([a,\#6]),\#3]$.

In general, the largest context may be embedded deep inside a structured term. Thus each of the contexts must be examined to retrieve the largest one. This imposes an efficiency problem in terms of execution time. The problem of indirect failure is caused by introducing an irrelevant context during unification. This context will lead the interpreter into backtracking to a place where there is no more alternative matching clauses for the backtrack subgoal. The remedy to this problem is to adapt the context

unification algorithm in the following way: during a unification step, whenever a context term t is assigned to a context variable X where the current context is $\#i$, we have the substitution $\{X/[t,\#i]\}$ when there is one or more alternative clauses for the current subgoal. Otherwise the current unification is deterministic and the substitution is $\{X/[t,\perp]\}$. The idea is not to include any context which corresponds to a deterministic node during unification. Thus every context corresponds to a node in the search tree where there is one or more alternative clauses. In this way, the context $\#c$ which is associated with the introduced symbol t will not be *blocked* by an irrelevant context which occurs *later* in the derivations and corresponds to an indirect-failed node with no alternative clause. During a unification failure which involves t, the interpreter can directly backtrack to $\#c$, without going through the deterministic indirect-failed node.

$p(a)$;
$p(e)$;

$q(b)$;
$q(c)$;

$r(Z,\_) \leftarrow Z = c$;
$r(\_,c)$;

? $p(X)$ & $q(Y)$ & $r(Y,X)$;

Program 3.1

A serious problem arises when we eliminate the context which refers to a deterministic node. Consider Program 3.1 where $r(Y,X)$ is invoked with X bound to $[a,\#p]$ and Y to $[b,\#q]$. (We use context $\#x$ to refer to the choice point created by the activation of the subgoal with predicate name x.) The head of the corresponding clause $r(Z,\_)$

unifies successfully with $r(Y,X)$ and Z is bound to $[Y,\#r]$, which is $[[b,\#q],\#r]$. The next call is $Z = c$, which fails and a failure context $\#r$ is returned. Thus backtracking is done into the subgoal $r(Y,X)$, and unification is attempted between $r(Y,X)$ and $r(\_,c)$. The unification fails because of the conflict between the value of X and c. The failure context returned is the largest context associated with X ($\#p$) and backtracking is done into p where X will get another binding. It is obvious that the solution $\{X/a, Y/c\}$ is missed because the intelligent interpreter skips the backtrack point q.

During the execution of the call $Z = c$ where Z is bound to $[[b,\#q],\#r]$. The failure context returned is $\#r$ (since $\#r > \#q$) and $\#r$ is put into the B-list. The context $\#q$ should also be put into the B-list at this point since it is also a possible choice point which may 'correct' the current failure. The idea is to store *all* the failure contexts associated with a conflict function symbol in the B-list so that when one of the choice points (for example $\#r$) does not help in resolving the failure, we can try the other backtrack points (for example $\#q$) later.

The second modification to the unification algorithm is as follows: during a unification failure, *all* the contexts which are associated with the function symbol which causes the conflict are put into the B-list. This operation is represented as *Retrieve_All*(f) which returns a list of contexts which is associated with f, where f is the function symbol which causes a unification failure. *Retrieve_All* is very different from *Get_All_Contexts* although both operations involve retrieving all the contexts associated with a term. Since *Get_All_Contexts* is performed on all the arguments of an indirect-failed subgoal and *Retrieve_All* is performed on a function symbol, which may be an argument of a failed subgoal or a subterm within an argument of the failed subgoal, the set of contexts gathered by *Get_All_Contexts* is a superset of the set of con-

texts gathered by *Retrieve_All*. Hence the backtrack point selected by the scheme with *Retrieve_All* is at least as good as the backtrack point selected by the scheme with *Get_All_Contexts*. This makes the scheme with *Retrieve_All* more intelligent. In addition, in terms of execution time, *Retrieve_All* is more efficient since less contexts have to be manipulated during a unification step.

We have proposed two modifications to the unification algorithm: that is, exclusion of a context which refers to a deterministic node and the introduction of the *Retrieve_All* operation during a unification failure. The unification routine (*new_unify*) which incorporates the two ideas is now presented in Figure 3.1. *new_unify* is very similar to *unify_1* except for the following:

1. When unification is successful, we introduce the current context (*#i*) in the context term which is assigned to the context variable when there are other candidate clauses available for the current subgoal. Otherwise, we introduce the context $\perp$ in the context term to denote that the current unification is deterministic.
2. During a unification failure, we return a list of contexts associated with the context term(s) which cause(s) the conflict in the variable *failure_contexts*. This list will be merged with the *B*-list.

3.2. The Prover Algorithm

The unification routine is incorporated into a revised version of a Prover algorithm (Figure 3.2), which is originally presented in [You&Wang 88]. This new version treats direct-failed and indirect-failed subgoals uniformly and is invoked by *Prover*(G_0, *program*) where G_0 is the list of subgoals to be solved and *program* is the list of clauses which is available for solving the subgoals. The algorithm uses the following functions:

leftmost

returns the leftmost unsolved subgoal from a list of subgoals to be solved;

candidate_clause
returns an untried candidate clause for the current subgoal;

head returns the head of a clause;

new_unify
the modified unification algorithm presented in Figure 3.1;

resolve
the context resolution procedure presented in Section 2.2.2;

max takes a list as argument and returns the maximum element in the list. NIL is returned if the list is empty;

reset_subgoal
takes a backtrack point as argument and restores the subgoal at that backtrack point. That is, undo all the variable bindings since that backtrack point is created.

The Prover algorithm works as follows: given a goal G_i to be solved, we choose the leftmost subgoal to resolve. If there still exists an untried candidate clause, we then try to unify the head of the clause with the chosen subgoal. If unification succeeds, we resolve the subgoals and invoke Prover recursively on the newly-derived goal. Otherwise, we save the failure contexts in the B-list and try the next candidate clause. By the time all candidate clauses have been exhausted, the contexts that are the sources responsible for each unification failure have already been saved in the B-list. We then retrieve the most recent (largest) context in the B-list for backtracking. If the B-list is empty, the interpreter does not benefit from intelligent backtracking and it has to backtrack to the most recent backtrack point. If the most recent backtrack pointer is NIL, we have exhausted all the possibilities and the Prover will return FAILURE, meaning that the proof of G_i is unsuccessful. Otherwise, the execution continues at the restored subgoal pointed to by the selected backtrack point by invoking Prover on that subgoal again.

```
new_unify(t , l) {

  dereference(t);
  dereference(l);

  if (t is a context variable) {
    if (no more candidate clause for the current subgoal)
      Θ ← Θ ∪ {t / [l,⊥]};
    else
      Θ ← Θ ∪ {t / [l,#i]};
    return(SUCCESS);
  }
  else if (l is a context variable) {
      if (no more candidate clause for the current subgoal)
        Θ ← Θ ∪ {l / [t,⊥]};
      else
        Θ ← Θ ∪ {l / [t,#i]};
      return(SUCCESS);
    }
    else {
      if (functor(t) ≠ functor(l) or arity(t) ≠ arity(l)) {

        if (context(t) ≠ ∅ or context(l) ≠ ∅)
          failure_contexts ← Retrieve_All(t) ∪ Retrieve_All(l);
        else
          failure_contexts ← list of contexts associated with the context
                                  variable which caused the disagreement;

        return(FAILURE);
      }
      for i in 1 to arity(t) do {
        t_i ← i-th argument of t;
        l_i ← i-th argument of l;
        if (not new_unify(t_i , l_i))
          return(FAILURE);
      }
      return(SUCCESS);
    }
}
```

Figure 3.1: A Modified Unification Algorithm

```
B-list ← ∅;
            /* the list is initially empty */
Prover(G_i, program) {

  if (G_i = NIL)
    return(SUCCESS);
            /* an answer has been found */

  current_subgoal ← leftmost(G_i);
            /* use the leftmost computation rule */
try:
  clause ← candidate_clause(current_subgoal, program);
            /* get an untried candidate clause */
  if (clause ≠ NIL) {
            /* an untried candidate clause is found */
    if (new_unify(current_subgoal, head(clause))) { /* unification succeeded */
      G_{i+1} ← resolve(G_i, Θ);
      return(Prover(G_{i+1}, program));
    }
    else { /* unification failed */
      B-list ← B-list ∪ failure_contexts;
                  /* save potential backtrack points */
      goto try;
    }
  }
 /* no more candidate clause */
  backtrack_point ← max(B-list);

  if (backtrack_point = NIL)
    backtrack_point ← most recent backtrack point;
  else
    B-list ← B-list - {backtrack_point};
     /* update B-list */

  if (backtrack_point = NIL)
    return(FAILURE);

  B-subgoal ← reset_subgoal(backtrack_point);
     /* restore the goal pointed to by backtrack_point */
  return(Prover(B-subgoal, program));
}
```

Figure 3.2: A Prover Algorithm

Partial execution trace:

step: context:

1	#0:	p(X,Y) & q(Y) & r(X,Z) & s(Y,Z) & t(Y,Z); { X/[a,#0] , Y/[a,#0] }
2	#1:	q([a,#0]) & r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z);
3	#2:	r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z); { Z/[b,#2] }
4	#3:	s([a,#0],[b,#2]) & t([a,#0],[b,#2]); direct failure, goto #2
5	#2:	r([a,#0],Z) & s([a,#0],Z) & t([a,#0],Z); { Z/[c,⊥] }
6	#3:	s([a,#0],[c,⊥]) & t([a,#0],[c,⊥]);
7	#4:	t([a,#0],[c,⊥]); direct failure, goto #0
8	#0:	p(X,Y) & q(Y) & r(X,Z) & s(Y,Z) & t(Y,Z); { X/[a,⊥] , Y/[b,⊥] }

.
.
.

Figure 3.3: An Execution Trace of Program 2.2.

A partial execution trace of Program 2.2 (page 23) using this algorithm is shown in Figure 3.3. Notice that in step 5, when Z is unified with the constant c, the context *#2* is not included. This is due to the fact that after the second r clause is tried, r is no longer a backtrack point. This piece of context information (*#2*) is irrelevant in the sense that the interpreter will never backtrack into r anyway. Later in step 7 when t([*a*,*#0*],[*c*,⊥])

fails to unify with all candidate clauses, the context returned is #0. Context #0 is now not blocked by context #2 as before and the indirect failure node in step 5 is avoided. After a few more steps, the interpreter comes up with the answer $\{X/a, Y/b, Z/b\}$.

3.3. Completeness of the Modified Scheme

The completeness of the modified scheme can be shown (informally) by considering the following two cases. Recall that in our interpreter, occurs check has been ignored and a unification failure can only be caused by a pair of conflicting function symbols.

1. In the original scheme, all the contexts being inserted into the *B*-list during direct failures are also in the *B*-list in the modified scheme.
2. In the original scheme, all the contexts associated with the function symbols being inserted into the *B*-list during indirect failures can be divided into two groups. The first group of contexts are those which are associated with the symbols which will later fail directly. Thus the associated contexts will be inserted into the *B*-list. The second group of contexts are those which are associated with function symbols which will never fail in later derivations, and the backtrack points which correspond to these associated contexts will never lead to solutions.

From these we can see that the *B*-list in the modified scheme is a subset of the *B*-list in the original scheme and the outstanding contexts are those which correspond to the introduced function symbols which will never fail in unifications. Since the completeness of the original scheme has been proven in [You&Wang 88], we conclude that the modified scheme is complete. That is, all backtracking points being pruned will lead to no solution.

3.4. Finding Subsequent Solutions

[You&Wang 88] does not consider intelligent backtracking for the purpose of generating multiple answers. Since subsequent solutions are often required in Prolog programs†, the interpreter should make use of intelligent backtracking to search the whole

† Most Prolog implementations provide a predicate which returns all the answers to a query prov-

search tree, not just until a success leaf is reached. Since the last unification is successful, no failure context is returned. To carry on with the search process, the interpreter should backtrack to the most recent non-deterministic node. Then the search can resume from that node with intelligent backtracking.

3.5. Triggering of Intelligent Backtracking

The triggering of the intelligent backtracking scheme is controlled by a pair of switches implemented in terms of a pair of high-level Prolog predicates. This allows the user to deactivate the mechanism totally, or trigger intelligent backtracking on a per-clause basis. In this way, context resolution can be enabled during parts of the program where the user knows that intelligent backtracking will speed up the execution. When a user knows that a program will not benefit much from intelligent backtracking, (s)he can turn off the scheme to reduce the amount of overhead incurred.

The operation of intelligent backtracking is composed of two related processes. The first is an *association process*. The second is a *manipulation process*. Both processes occur during unification. The association process takes place when a unification is successful, and a context variable is bound to a context term. A context is associated with that context term and is carried along in later derivations. The manipulation process takes place when a unification failure occurs. The contexts associated with the conflicting symbols are retrieved and put into the B-list. After all available clauses are tried, the maximum context in the B-list is chosen for backtracking.

The triggering of intelligent backtracking is controlled by two switches, $S_{association}$ and $S_{manipulation}$, one for each process. The switches are implemented as two high-level

---

able from the program. This predicate is called *all_of* in WUP.

Prolog predicates, *association*(*X*) and *manipulation*(*X*). When *X* is **on**, the switch is turned on. When *X* is **off**, the switch is turned off. When *X* is uninstantiated, *X* will be bound to the current state of the switch, which is **on** or **off**.

Now when a user wants to ignore intelligent backtracking as a whole, (s)he will turn off both $S_{association}$ and $S_{manipulation}$. In this way, no overhead will entail. When a user wants to enable intelligent backtracking on a per-clause basis, (s)he will always turn on $S_{association}$. In addition, the user will use a pair of $S_{manipulation}$ switches to enclose all the subgoals that should be solved with the intelligent interpreter. Note that when $S_{manipulation}$ is off and backtracking is required, the interpreter backtracks to the most recent backtrack point. For example, the goal statements:

? association (*on*) & *manipulation* (*on*);
? p_1(...) & *manipulation* (*off*) & p_2(...) & *manipulation* (*on*) &
p_3(...) & *association* (*off*) & *manipulation* (*off*) & p_4(...);

make use of intelligent backtracking in solving the subgoal p_1. Then the manipulation switch is turned off, and p_2 is solved without intelligent backtracking. Later, intelligent backtracking is enabled again in solving p_3. Then the scheme is turned off totally in solving p_4. By default, both switches are turned on before execution. Thus a program can run without modification on an intelligent interpreter. Note that once the association switch is turned off, intelligent backtracking cannot be enabled again since some contexts which may lead to solutions will be missing from the derivations.

3.6. Summary

We have proposed a modified version of a unification scheme which is more powerful than the previous schemes. The new scheme is based on the idea that we omit the piece of context information when the derivation is deterministic so that some other embedded contexts will not be blocked. In addition, we have introduced a *Retrieve_All*

operation which is more efficient than a *Get_All_Contexts* operation. Under these modifications, the direct-failed subgoals and indirect-failed subgoals are given uniform treatment and the unification algorithm is simplified. Finally the ways in which the intelligent backtracking scheme is triggered have been presented. This triggering mechanism is versatile and allows the activation of intelligent backtracking on a per-clause basis.

Chapter 4

Implementation of Context Resolution

In this chapter, we will discuss the implementation of context resolution on Waterloo Unix Prolog (WUP). Detailed data and control structures of the implementation will be given. The methods used for handling the impure constructs (for example, 'cut' and 'not') will also be presented. We will see how these constructs are handled without turning off the intelligent backtracking scheme.

4.1. Incorporate Context Resolution into WUP

Since context resolution fits naturally into the unification mechanism of conventional Prolog implementations, intelligent backtracking can be implemented on top of WUP without changing much of the original code of the interpreter. The main problem is to find suitable data and control structures that will show the feasibility of the scheme. Every effort has been attempted to implement the scheme in a clear and concise way. It is our belief that future implementations that fully utilize a particular machine's architectural features and/or a particular language's support will give an even better performance in terms of execution time or memory consumption statistics.

4.2. Overview of Prolog Implementation

This section is not intended as an in-depth analysis of Prolog implementation but rather as a description of the general data structures and control mechanisms which will help in the understanding of the rest of this chapter. Readers who are interested in the details of Prolog implementation should refer to [Hogger 84], where a detailed description of

the implementation of conventional Prolog interpreters and a complete execution control algorithm were given.

In conventional Prolog implementations, the memory is divided into two main data areas. The first area is static in the sense that it will not change throughout program execution. It is called the *input heap* and it stores the codified version of the input logic program. The second area is called the *execution stack* (which consists of several stacks†, to be explained later) which is dynamic and represents the proof tree (also known as the *AND*-tree) during program execution.

The input heap is created before program execution. All the clauses with the same predicate name and arity are linked together corresponding to the textual order in the original input program. They are compacted and codified in a suitable form. During program execution, whenever a predicate with a certain name and arity is requested, the interpreter will look into this heap and retrieve the next available clause in turn.

The execution stack represents both the execution path of a program and the assignment of variables. This is usually implemented in terms of two stacks. The first stack is called the *Run Time Stack*. Whenever a subgoal is unified with the head of a clause, a node is created on this stack. Each node consists of the space allocated for variable assignment and the space for all the control information needed for execution. If other candidate clauses are available for unification with the current subgoal, a non-deterministic node is created. Otherwise, a deterministic node is created. Another stack called the *Trail* is used which keeps track of those variables whose instantiations have to be undone (reset to 'uninitialized') during backtracking.

---

† In the following discussion, we assume that a stack grows upwards.

The pointers required for control are:

- A *Return* pointer which points to t' ext subgoal to be solved. When the current subgoal is solved successfully, the interpreter will look into this pointer to activate the next subgoal.
- A *Parent* pointer which points to the parent† subgoal of the current subgoal. This maintains the structure of the proof tree. In addition, when the Return pointer is NIL (the current subgoal is the last subgoal in the body of a clause), the Parent pointer gives access to the parent of the current subgoal. The Return pointer of the parent of the current subgoal gives the next subgoal to be activated.

In addition to the Return and Parent pointers, the following pointers are required for non-deterministic nodes:

- If more than one cl · can unify with the active subgoal, the next available clause is indicate·l by · *Next Clause* pointer. This pointer essentially keeps track of a list of ·:e clauses which have not been tried by the interpreter. Later if bac· ng is done to this subgoal, this pointer is consulted to look for alternative clauses which will be used to unify with the subgoal.
- A global register *MB* is used to indicate the most recent backtrack point. When a failure is encountered, all the nodes from the current one down to that indicated by MB are discarded, and the *Next Clause* pointer in the node indicated by *MB* indicates the next clause to be tried for unification. A *Previous Back* pointer is used to indicate the previous backtrack point. When a non-deterministic node is created, the content of MB is copied to Previous Back and MB points to the current non-deterministic node. When backtracking is done to a node *N*, the content of Previous Back in *N* is copied into MB then all the nodes from the most recent one down to *N* are discarded. This ensures that the execution path is properly preserved when backtracking occurs.
- Whenever the variable cells of the nodes under the most recent backtrack point are assigned, these assignments are recorded in the *Trail*. On backtrack-

† A subgoal *q* is the *parent* of a subgoal *p* if *p* is an instance of a subgoal occurring in the body of the clause whose head unifies with *q*.

ing, these assignments have to be undone so that the execution is returned to the previous state. The *Reset* pointer indicates the segment of the Trail to be popped. When the interpreter backtracks, all the variable assignments indicated by the top of the *Trail* until *Reset* are reset.

The principal data structure in Prolog programs is a term, which can be a simple constant, a variable or a complex nested structure. During program execution, when a constant or a variable is assigned to a variable *X* by unification, a pointer to that constant† or variable is placed in the variable cell of *X*. For variable-to-structured-term assignments, the simplest scheme is to place a pointer to a block of memory cells representing the term in the variable's cell. This scheme is called *structure copying* [Mellish 82]. Whenever a new structured term is created, a new copy is made of the original code. Thus term construction is slow. However, the code for a term is readily available, thus accessing is fast.

Structure sharing is an alternative to structure copying in the construction of new terms. In order that all the instances of a structured term share the code of that term (for space efficiency reason), the concept of binding environment is introduced. Now a variable is defined by two pointers: one to the code (or skeleton) of the structure, which defines the general "shape" of the term, and another to the binding environment, which is the context in which the skeleton is used. A variable is always accessed in the context of a binding environment. For example, the assignments *X* / [*U*|*V*] where *U*/*a* and *V*/*b*, and *Y* / [*U*|*V*] where *U*/*c* and *V*/*d*, are represented by storing the code for [*U*|*V*] in both *X* and *Y*'s structure pointers. In addition, the bindings where *U* is bound to *a* and *V* is bound to *b* is stored in *X*'s binding environment and the binding environment where *U* is bound to *c* and *V* is bound to *d* is stored in *Y*'s binding environment. This scheme

† Generally speaking, a small constant can be stored in the variable's cell directly.

allows rapid construction of terms and great saving in space if a complex structure is shared by many variables. However, a long chain of dereferencing is necessary to retrieve a variable thus accessing is slower. In addition, instead of using one pointer per variable as in structure copying, structure sharing uses two.

4.3. Overview of WUP

WUP is written in the C language under the UNIX† operating system [Cheng 84]. It provides an integrated environment in which a user can create, maintain and execute Prolog programs. Its concept of modular programming and separate compilation, together with a large library of built-in predicates, allow the development of Prolog programs of reasonable size. It employs the method of structure copying in the construction of new terms and it recognizes tail recursion optimization [Hogger 84]. Its main features are summarized in the following six subsections.

4.3.1. Storage Organization

It uses the usual 2-stack representation of run time structure. In addition, a third stack, called the *Copy Stack*, is used for storing the constructed terms during execution. Two kinds of nodes are stored on the Run Time Stack: control node and environment node. A control node stores all the control information required for execution. It is created, together with a possibly empty environment node, whenever the active subgoal unifies successfully with the head of a clause. The size of a control node depends on whether the current node is deterministic or non-deterministic. An environment node is the storage allocated for the variables. Its size depends on the number of unique variables in the matched clause. Each environment node is divided into a number of slots, one for

† UNIX is a trademark of AT&T Bell Laboratories.

each variable.

When a structured term is assigned to a variable, the term is constructed in the Copy Stack. A pointer is oriented from the slot of that variable to the term in the Copy Stack. When a short constant (for example, an integer, a real number or a character) is assigned to a variable, no copying is made. The constant is assigned to the slot of the variable directly. When two uninstantiated variables are unified, a pointer is oriented from the slot of the first variable into the slot of the second one.

During backtracking, all the stack nodes already created down to the most recent non-deterministic node will be discarded. We have to make sure that no pointers are referring to variable slots in these nodes which have already been discarded. Otherwise these pointers will become dangling. To avoid this problem, all the pointers in the Run Time Stack are oriented downwards. This is enforced by the following rules:

- The Copy Stack is placed *under* the Run Time Stack. When a structured term is referred to by a variable, that pointer is downward.
- When two free variables are unified, a pointer is oriented from the slot of the variable higher in the stack to the slot of the other variable.

Under this arrangement, a node to be discarded will never be referred to by any nodes which are still in the Run Time Stack. The structures of the Run Time Stack and Copy Stack are summarized in Figure 4.1.

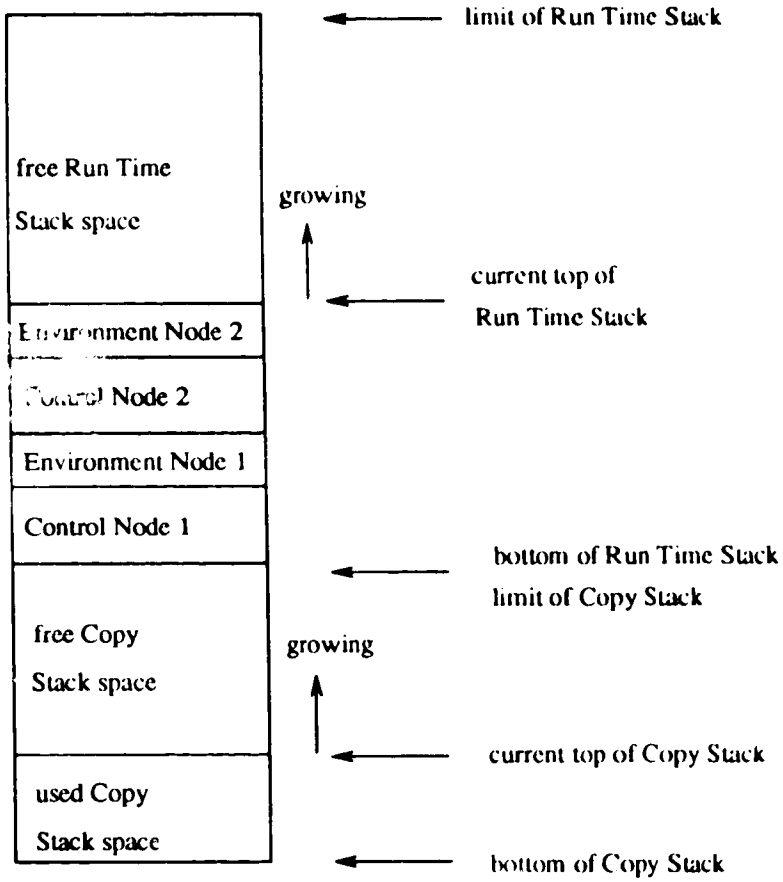


Figure 4.1: Run Time Stack and Copy Stack

```
typedef struct pc_word {
    int tag;
    union word {
        int ival;
        float fval;
        struct pc_word *ptr;
        char *sval;
    }
} PC_WORD;
```

Figure 4.2: The declaration statements in C for a PC_WORD

4.3.2. Structure Representation

The basic building block of the run time structures (clauses and stacks) is a *PC_WORD*, which is a record consisting of two fields. The first field specifies the type of this record and the second field is an integer word which can be used to store any appropriate object depending on the record's type. For example, it can store an integer, a real number, a character constant, an address of another PC_WORD or the address of a string of characters. The usage of this field depends on the context where this record is used. The declaration statements in C for a PC_WORD is given in Figure 4.2. As an example, a non-deterministic control node is made up of 7 PC_WORDs and a deterministic control node is made up of 3 PC_WORDs. The structures of the control nodes are shown in Figure 4.3. The function of each field has been described in Section 4.2, except for the following:

Module A pointer to the module containing the matched clause of the current subgoal.

Copy A pointer to the Copy Stack. It records the top of the Copy Stack at the

time of creation of the current node and it is used to indicate the segment of the Copy Stack to be popped upon backtracking.

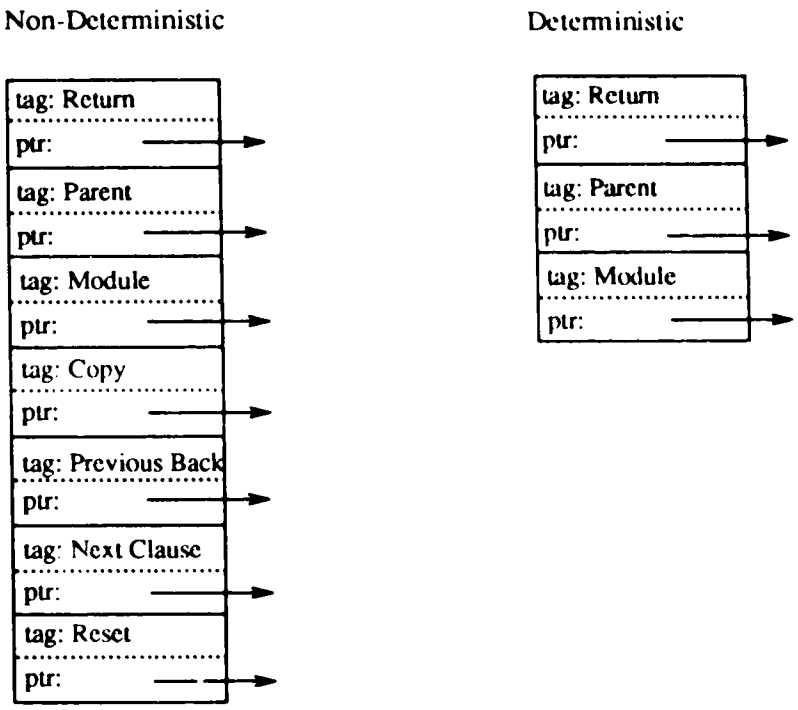


Figure 4.3: Control Nodes

4.3.3. The Interpreting Algorithm

The interpreting algorithm used in WUP is van Emden's ABC Algorithm [van Emden 82]. It is a simple, non-recursive algorithm for depth-first, left-to-right traversal of a tree. The algorithm assumes that the proof tree is implemented as a stack where the sequence of nodes from the root up to the current node is kept, and the algorithm specifies the list of actions to be performed on subgoal selection, candidate call selection and backtracking based on this assumption. The ABC algorithm is the first published description of a complete interpreting algorithm.

4.3.4. Table-Driven Unification

The unification routine is table-driven. The unification table is a two-dimensional array which has as its dimensions all the possible arguments that can appear during the unification process. The first dimension is the type of the caller, and the second dimension is the type of the object being called. The entries in the table are predefined constants, which are the possible outcomes for the given arguments. This arrangement allows a systematic treatment of various kinds of arguments to be unified and makes the unification routine more manageable and extensible. The unification table is shown in Table 4.1.

The entries in the unification table determine the actions taken during the course of unification. In the point of view of programming, this table-driven set-up eliminates deeply nested *if-then-else* constructs which are error-prone and difficult to modify.

Table 4.1: Unification Table

calls → ↓ heads	Free Var.	Void Var.	Int. Const.	Float Const.	Atom Const.	Char Const.	Empty List	Const. List	Var. List	Const. Funct.	Var Funct.	Stream
Free Variable	*CF*	*S*	*AC*	*AC*	*AC*	*AC*	*AC*	*TC*	*PC*	*TC*	*PC*	*FSP*
Void Variable	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*
Integer Constant	*All*	*S*	*SC*	*IF*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*
Float Constant	*All*	*S*	*FI*	*SC*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*F*
Atom Constant	*All*	*S*	*F*	*F*	*SC*	*F*	*F*	*F*	*F*	*F*	*F*	*F*
Char Constant	*All*	*S*	*F*	*F*	*F*	*SC*	*F*	*F*	*F*	*F*	*F*	*F*
Empty List	*All*	*S*	*F*	*F*	*F*	*F*	*S*	*F*	*F*	*F*	*F*	*LSP*
Constant List	*TII*	*S*	*F*	*F*	*F*	*F*	*F*	*UL*	*UL*	*F*	*F*	*LSP*
Variable List	*PII*	*S*	*F*	*F*	*F*	*F*	*F*	*UL*	*UL*	*F*	*F*	*LSP*
Constant Functor	*TII*	*S*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*UF*	*UF*	*F*
Variable Functor	*PII*	*S*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	*UF*	*UF*	*F*
Stream	*SPF*	*S*	*F*	*F*	*F*	*F*	*SPL*	*SPL*	*SPL*	*F*	*F*	*SPSP*

F	always fails	*S*	always succeeds
CF	case free var	*AC*	head = call
All	call = head	*UL*	unify lists
UF	unify functors	*PC*	head = copy(call)
PII	call = copy(head)	*SC*	simple comparison
TII	if test then call = head	*TC*	if test then head = call
IF	unify integer and float	*FI*	unify float and integer
FSP	free and stream ptr	*SPF*	stream ptr and free
LSP	list and stream ptr	*SPL*	stream ptr and list
SI SP	stream ptr and stream ptr		

4.3.5. 1-Clause Lookahead

Since a non-deterministic node takes up more space than a deterministic node, it is a good idea to differentiate between the two kinds of nodes. One way to do this is by Warren's indexing scheme [Warren 77] which classifies parameters into different groups. In WUP, the first argument in the head of the next available clause is checked against the first argument in the current subgoal. If they do not match, then the clause can be skipped and the next candidate clause is checked. Otherwise, a non-deterministic backtrack node is established and the execution proceeds. Consider Program 4.1, the clause *append* will append its second argument to the end of the first argument to form the third argument. In WUP, no non-deterministic node will be set up because the first argument in the first clause and the first argument in the second clause are mutually exclusive. On a conventional implementation without lookahead, all the stack nodes are non-deterministic (except the last one†) since there are always two matching clause heads for *append*.

---

append([*X* | *L* 1], *L* 2, [*X* | *L* 3]) ← *append*(*L* 1, *L* 2, *L* 3);
append([ ], *X* , *X*);

? *append*([1,2], [3,4], *List*);

Program 4.1: A Program to Illustrate Clause Lookahead

---

† During the last invocation of *append*, the first argument is an empty list, which fails to unify with the first argument of the head of the first clause ([*X*|*L*1]). Thus the last stack node is deterministic.

4.3.6. Module Concept

In a programming project, it is often desirable to break a large program into smaller, more manageable parts. In WUP, a programmer can break his program into *modules*, which can be distributed over different files in different directories. The programmer can explicitly export a module, which makes it accessible to other modules. Otherwise, the routines in a module are local to that module. This scheme provides an excellent programming environment whereby large logic programming projects can be accomplished.

The modules are organized into a hierarchical tree-like structure, which is very similar to the file system of UNIX. A predicate is searched for in the current module first. If it is not found then all the children of the current module are checked to see if the predicate has been exported. If it is still not found then the parent of the current module is searched. This process continues until we reach the root (the library of build-in 'standard' predicates), where a failure will be reported if the required predicate is still not found.

4.4. Data Structures

To incorporate context resolution into WUP, we need two main data structures. The first data structure is used to hold the contexts which are associated with a term. This structure can be implemented using a list of records; each is called a *context record*, which consists of two fields: the first field (*Context*) is the context associated with the term, and the second field (*Link*) is a link to the next record. Two more fields are added to a PC_WORD to accommodate this arrangement. The first field is a pointer to the list of contexts, and the second field is a flag which indicates whether this PC_WORD is associated with a list of contexts. For example, the internal representation of the context

term [*f*([*a*,#0],#1],*b*),#2] is shown in Figure 4.4. Notice that by definition, context #2 is associated with *f*,*a* and *b*, and contexts #0 and #1 are associated with *a* only.

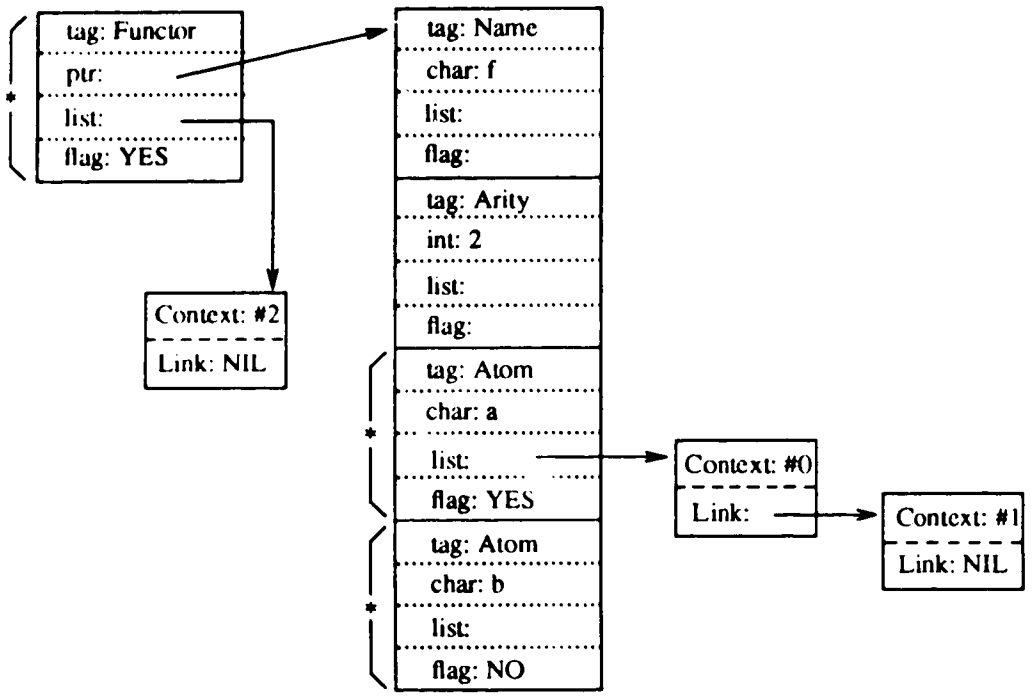


Figure 4.4: Internal Representation of a Functor

In Figure 4.4, five PC_WORDs are used by WUP to represent the structured term. However, only three of these PC_WORDs (as marked by '*') will ever be associated with a list of contexts, and the space allocated for the context list pointer and the flag in the other two PC_WORDs is wasted. This suggest a possible optimization: instead of storing the actual value of an atom or a pointer to a structured term, the pointer field of a PC_WORD now stores an address of a record, which has the term's value, a pointer to the list of contexts and the flag as before. This arrangement allows a great saving in memory since a large part of the execution stack is made up of PC_WORDs which will never be associated with a list of contexts. However, one more dereferencing is required

to access the term or the associated contexts and this makes accessing slower.

The second main data structure is the *B*-list used in the Prover algorithm for storing the failure contexts. It is implemented as an array (the *context array*, or *CA* for short), where each of its element is a context. In the implementation, the address of the Run Time Stack is used as a context for tagging. Therefore, a context is just a pointer to a PC_WORD, which makes up the stacks.

4.5. Control Structure

The mechanism of intelligent backtracking is composed of the association process, manipulation process and backtracking step (Section 3.5). We will present the major actions performed at each step. In the following, CA[i] represents the i-th element of the context array.

Association Process

When a variable X is bound to a term t, one of the following cases applies:

1) t is a short constant

the tag and the value of the constant is copied directly into X's PC_WORD;

2) t is a structured term

the structured term is constructed in the Copy Stack and a pointer to the structure is put in the PC_WORD of X;

3) t is a free variable

a pointer is constructed from the PC_WORD of the variable higher in the Run Time Stack (suppose it is X) to the PC_WORD which represents the other variable.

Suppose the current stack node, with address *addr*, is non-deterministic. The intelligent interpreter has to create a context record (with Context field initialized to *addr*) and add this to the list of contexts already associated with X. If X is not associated with any contexts (the flag is off), the interpreter will turn the flag on. Furthermore, if t has already been associated with some contexts before unification, that list of contexts has to be shared by X. In general, a list of contexts

is shared by multiple variables in order to gain a higher efficiency in memory consumption.

Manipulation Process

We assume that a global variable LC (largest context) is used to keep track of the largest failure context encountered during unification. It is initially set to ⊥. In addition, all the elements of the context array have been initialized to ⊥ before the unification routine is entered. Whenever a term fails to unify with another term, the interpreter performs the following steps on both terms:

for each context *addr* in the list of contexts associated with the term do:

```
i ← hash(addr);
CA[i] ← addr;
if (CA[i] > LC)
    LC ← CA[i];
```

At the end of the Manipulation Process, the variable LC stores the largest context in the context array, and all the contexts in the array are sorted in ascending order of the array index.

Backtracking Step

If no more candidate clause is available for the current subgoal, the interpreter will do the following:

```
if (LC = ⊥)
    backtrack to MB;
else {
    i ← hash[LC];
    CA[i] ← ⊥;
    Temp ← LC;
    LC ← largest context in CA;
    backtrack to Temp;
}
```

If the context array is empty, we backtrack to the point indicated by MB. Otherwise we remove the largest context from the context array and update LC to the next largest context in the array. This is done by scanning the array sequentially from the top until an element is found whose value is not ⊥.

This implementation adds a little overhead to the interpreter since only a few simple operations are necessary during each unification and backtracking step. The most time-consuming operation, the sorting of contexts, is eliminated by hashing each failure context into the context array and the variable LC always holds the largest failure context, which is immediately available for backtracking. The hashing step in the manipulation process hashes an address into an index by first subtracting the address of the beginning of the Run Time Stack from the stack address, then dividing the number obtained by a constant to scale it down. The constant is chosen so that no two addresses will collide into the same index. In the implementation, right shift operation is used to replace the division operation for faster execution time.

Since the size of a non-deterministic node is 7 PC_WORDs, the current non-deterministic stack address is at least 7 PC_WORDs away from the previous non-deterministic stack address. Thus we define:

$$\text{hash}(current) = (current - start) \gg 2$$

to get an index into the context array without collision, where *current* and *start* are the address we want to hash and the address of the beginning of the Run Time Stack expressed in number of PC_WORDs respectively, and $\gg$ is the right shift operator in C. Note that when the binary representation of an integer is shifted n bits to the right, the integer obtained is the same as when the original integer is divided by 2^n.

4.6. Handling of Special Constructs

In this section, the implementations of some of the predicates which need special treatment are discussed. Although these are all extra-logical constructs, there is general agreement as to the need of these constructs to make Prolog a more practical programming language for efficiency and convenience reasons. Care should be exercised when

programs with these impure constructs are executed on an intelligent interpreter. We will see how some of these constructs are handled without sacrificing intelligence.

4.6.1. Cut

The most controversial impure construct in Prolog is the 'cut' operator. When a 'cut' operator is encountered, all the backtrack points set up by the subgoals in the same clause to the left of the 'cut', down to the backtrack point for the parent predicate whose body contains the 'cut', are discarded, and the global register MB will be updated accordingly. Later when a unification failure occurs and a maximum failure context from the context array is returned which indicates a node which has already been discarded, the interpretive process may be wrecked. This problem can be handled with negligible overhead in the following way: whenever a context *#c* is used for backtracking, we just compare it to MB. If *#c* is more recent than MB (*#c* > MB), we backtrack to MB. Otherwise backtracking is done to *#c*. Context resolution has been proven correct without considering any impure constructs. If now a 'cut' indicates an earlier backtrack point, the interpreter can simply backtrack to that point.

4.6.2. Fail

The 'fail' predicate always fails. It is mostly used in failure-driven loops. Since 'fail' has no argument, its failure will give no failure context. Thus whenever a 'fail' predicate is encountered, backtracking is done to the most recent non-deterministic node with untried alternatives (as indicated by the MB register). In general, we cannot just backtrack to the largest context in the *B*-list whenever 'fail' is executed.

4.6.3. Prove

Prove(X) is a *meta-predicate* which succeeds if its argument X is provable. X is called a *meta-variable* since it will bind to different pred[illegible] dynamically. When prove(X) is executed, the term X is proved as if it is an or[illegible]redicate. The 'prove' predicate is required since WUP differentiates between functors and predicates and they cannot be used interchangeably. The clause:

$$and(X,Y) \leftarrow X \ \& \ Y;$$

under minor syntactic change is prefectly acceptable to C-Prolog [Pereira 87]. WUP, however, will give a parsing error when the program is being read into the database. The clause should be modified as follows:

$$and(X,Y) \leftarrow prove(X) \ \& \ prove(Y);$$

The argument of the 'prove' predicate must be instantiated to a term, possibly with arguments. The WUP interpreter will internally translate the argument of 'prove' from a term representation to a predicate representation, which is then executed as usual. Bindings are possibly created. No special treatment from the intelligent interpreter is necessary.

4.6.4. Not

Negative information is often required in situations where Prolog is used to model a real-world phenomenon. The most commonly used tactic is to insert the negative facts in the database directly or to implement the *negation as failure* rule [Lloyd 84]. While the first approach is a more straightforward solution, the natural relationship between the negative subgoal and its positive counterpart is lost. For example, how do you relate the predicates *is_tall*(*tom*) and *is_not_tall*(*tom*)? In addition, it is redundant to represent two closely related facts. Most Prolog implementations employ the second approach

which uses the following interpretation to represent negation: if a system fails to prove a predicate $p(X)$, then it infers not($p(X)$).

The 'not' predicate can be handled when 'cut', 'fail' and 'prove' are handled properly since 'not' in WUP is defined as:

$not(X) \leftarrow prove(X)\ \&\ cut\ \&\ fail$;
$not(\_)$;

When X is provable, not(X) will fail since 'cut' removes the choice point set up by 'not' and the predicate 'fail' always fails. If X is not provable, backtracking will be done into the second clause, which will always succeed. However, some vital information is lost during this process. This is revealed by Program 4.2.

$p(a)$;
$p(b)$;

$q(d)$;
$q(e)$;
$q(f)$;

$eq(Z,Z)$;

? $p(X)\ \&\ q(Y)\ \&\ not(eq(X,a))$;

Program 4.2: A Program with the Not Predicate

After the subgoals p and q are invoked, the variable bindings are $X/[a,\#p]$, $Y/[d,\#q]$. The next subgoal to be activated is not($eq([a,\#p],a)$) which unifies with the first clause in the definition of 'not'. When prove($eq([a,\#p],a)$) is executed, it will succeed and give no failure context. Then the 'cut' predicate is executed, which eliminates the choice point set up by 'not'. Finally the 'fail' predicate is executed and the

interpreter will backtrack to the most recent choice point q. The next clause of q is tried and this procedure is repeated. Clearly, the change in the bindings in Y has nothing to do with the predicate not($eq(X,a)$). The interpreter should be able to backtrack intelligently to p for alternative bindings for X.

The problem is that 'prove' and 'fail' are two independent predicates which can be used separately. However, when they are used in conjunction as in the body of 'not', one of the contexts which appears in the two *unifiable* constants to be unified is in fact a failure context. Without a lookahead mechanism for subgoals, the interpreter does not know whether the contexts should be inserted in the B-list when unification of the two context terms is successful. In order to handle this case, we can modify the definition for 'not' as follows:

$$not(X) \leftarrow \#push\ \&\ prove(X)\ \&\ \#pop\ \&\ cut\ \&\ \#fail;$$
$$not(\_) \leftarrow \#pop;$$

Notice that the system predicates are preceded with a # to distinguish them from user-defined predicates. The interpreter will create a new empty context array by pushing all the contexts and their respective indexes onto a stack whenever the #push predicate is executed. In addition, a *Get_All_Contexts* operation is performed on X, the argument of 'not', and the largest context is put in a variable V, whose value is also pushed onto the stack. When X is being proved, intelligent backtracking can be employed as usual using the new context array. When the proof succeeds, the interpreter will execute the #pop predicate, which will restore the variable V and all the contexts in the context array. Then the 'cut' removes the backtracking point created by the 'not' predicate and #fail is executed. #fail is similar to 'fail', except that #fail will backtrack to V instead of MB. When the proof of X fails, the interpreter backtracks into the second 'not' clause. The #pop predicate will be invoked and the next subgoal to be activated is the one following

the 'not' subgoal.

Using the previous example, when $eq([a,\#p],a)$ is executed and the execution succeeds, the #pop predicate will update V to $\#p$, which is the correct choice point of p to backtrack into later when the #fail predicate is executed.

4.6.5. Assert & Retract

These two constructs are used to change the database of the program. While 'assert' creates program segments dynamically, 'retract' removes a clause from the database. The arguments to 'assert' and 'retract' are also meta-variables which can bind to different clauses dynamically during the execution of the program. These two constructs give Prolog program a self-modifying capability, which may mislead an intelligent interpreter. The problem is revealed using Program 4.3.

$p(a)$;
$p(c)$;

$q(a,w)$;
$q(a,a)$;

$r(b)$;

$s(X,Y) \leftarrow p(X)\ \&\ q(X,Y)\ \&\ assert(r(Y))\ \&\ r(X)$;

? $s(X,Y)$;

Program 4.3: A Program with the Assert Predicate

After the execution of the subgoals p and q, the variables X and Y will be bound to $[a,\#p]$ and $[w,\#q]$ respectively. After the 'assert' clause is executed, $r(w)$ is added to the database. Then the subgoal $r([a,\#p])$ will fail to unify with the sole candidate clause and

the failure context *#p* is returned. Backtracking is done into the clause *p* and finally the program will fail with no output. It can be easily seen that if backtracking is done into *q* instead, the output of the program will be {*X/a*, *Y/a*}.

This solution is missed because when a bound variable fails to unify with all the heads of the candidate clauses, the intelligent interpreter will backtrack into the most recent subgoal where the failed variable gets the binding. In the example, when *r*(*X*) where *X*/[*a*,*#p*] fails to unify with the head of the candidate clause, the interpreter backtracks into *p*, the subgoal where *X* gets bound, to get another value for *X*. The interpreter oversees the fact that backtracking into *q* may later *add* another clause to the database. The new term in that clause may be able to unify with the value of *X*, without actually changing the value of *X*.

The solution is to include the context information in the clause when that clause is asserted. This context indicates the place where the terms in the asserted clause are introduced. Later when unification with any terms in the asserted clause fails, this context will give backtracking information to the interpreter. As in the previous example, *r*([*w*,*#q*]) will get asserted instead of *r*(*w*). When the subgoal *r*([*a*,*#p*]) fails to unify with *r*(*b*) and *r*([*w*,*#q*]), the largest failure context is *#q*, which indicates the correct backtrack point.

Note that this approach relies on the fact that an 'assert' has been encountered before a backtracking occurs. In this way, the occurrence of the 'assert' can be recognized by the intelligent interpreter. If this is not the case (for example, the 'assert' in the example above is moved into the body of the second *q* clause), the user has to turn off the scheme for correct execution.

The 'retract' predicate poses a less serious problem to an intelligent interpreter than 'assert' since a previously failed subgoal is less likely to be successfully unified with a clause if some clauses are removed from the database. The handling of 'retract' is similar to the way we deal with 'cut'. When the interpreter backtracks into a non-deterministic node which now becomes deterministic after some clauses at that node are removed by 'retract', the interpreter just tries the backtrack point indicated by the MB register.

$s(A,B) \leftarrow p(A)\ \&\ q(A,B)\ \&\ r(A);$

$p(a);$
$p(b);$

$q(a,b);$
$q(b,a);$

$r(b);$

$?\ s(X,Y);$

Program 4.4

Step	Subgoal	Candidate	Unifier/State	Figure
1	s(X,Y)	s(A,B)←···	{X/[A,⊥],Y/[B,⊥]}	4.5b
2	p(X)	p(a)	{X/[a,#10]}	4.5c
3	q([a,#10],Y)	q(a,b)	{Y/[b,#17]}	4.5d
4	r([a,#10])	r(b)	fails	4.5b
5	p(X)	p(b)	{X/[b,⊥]}	4.5e
6	q([b,⊥],Y)	q(b,a)	{Y/[a,⊥]}	4.5f
7	r([b,⊥])	r(b)	succeeds	

Table 4.2: Summary of Execution of Program 4.4

4.7. Summary of Intelligent Backtracking Mechanism

To summarize the mechanism of intelligent backtracking in execution, an execution trace for Program 4.4 is presented. Table 4.2 shows the subgoal, the candidate, the unifier and the figure for reference for each step. Figure 4.5a to 4.5f are snapshots of the Run Time Stack during each step. The following should be noted:

- Address Offset is used in the snapshots of the Run Time Stack. It is defined as:

$$Address\ Offset = \frac{(Real\ Address - Address\ of\ start\ of\ Run\ Time\ Stack)}{sizeof\ (PC\_WORD)}$$

  Thus an Address Offset quantity refers to an address by the number of PC_WORDs. In the implementation, real address (that is, the address of the start of the control node) is used for tagging.

- As shown in Figure 4.5a, the first control node and environment node are created before a goal is solved. In subsequent execution, a control node is created whenever a subgoal matched a unifying candidate. An environment node is created immediate above the control node whenever there are variables appearing in the candidate clause and the size of the environment is equal to the number of distinct variables in the candidate clause. (For example, see Figure 4.5b where the environment node for $s(X,Y)$ is created.)

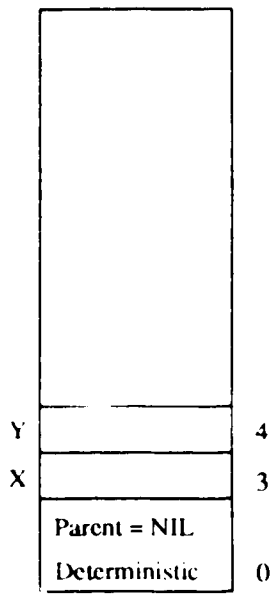
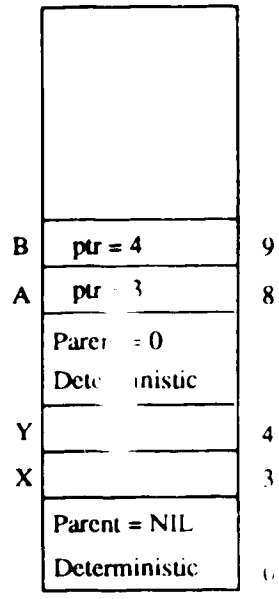
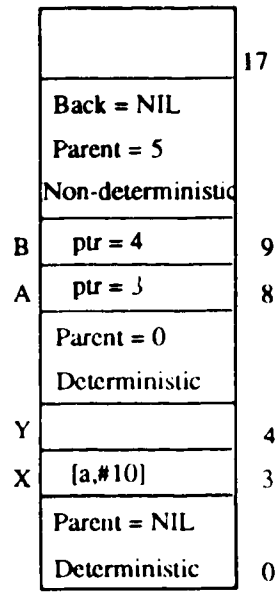


Figure 4.5a



4.5b



4.5c

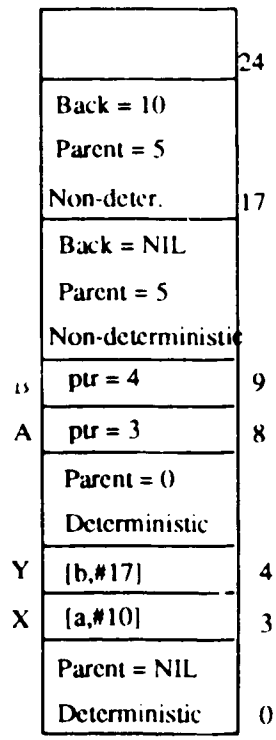
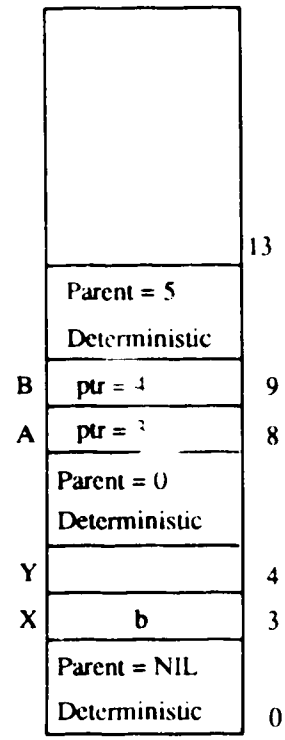
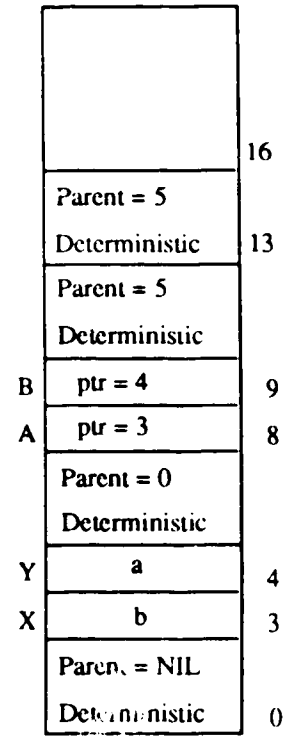


Figure 4.5d



4.5e



4.5f

- For simplicity, only the Parent pointer is shown in each control node. In addition, whenever a node is non-deterministic, the Previous Back pointer is also shown.
- In step 4, when the interpreter fails to unify the context terms $[a,\#10]$ and b, the failure context #10 is returned. Then the interpreter pops all the nodes from the current node up to #10, resets the bindings of the variables X and Y and backtracks to #10. An ordinary interpreter will backtrack to #17, which is the most recent non-deterministic node.
- In steps 1, 5 and 6, the contexts $\perp$ are used to attach to the terms since the corresponding nodes are deterministic. The context $\perp$ is not shown in the diagrams since this step actually corresponds to setting the flag field of a PC_WORD to a predefined constant to indicate that the term is not associated with any contexts.
- The final solution is $\{X/b, Y/a\}$.

4.8. Summary

We have presented an overview of the conventional implementation of Prolog, some special features of the WUP interpreter, and the implementation details of context resolution in WUP. In addition, we have looked at the ways by which some of the impure constructs are handled by the intelligent interpreter without turning off the scheme. Detailed algorithms have been presented which show how the association process, manipulation process and backtracking process are handled. The conciseness and simplicity of the algorithms suggest that this implementation will add only very little overhead to the interpreter. This observation will be justified in Chapter six, where some performance results are given. The main result we draw from this implementation is that context resolution can be easily incorporated into any existing conventional Prolog systems.

Chapter 5

Related Work

In this chapter, we will look at some of the well-known schemes for intelligent backtracking. A brief summary of the conceptual basis of each scheme will be presented and comparisons are made between context resolution and some of the schemes. The execution statistics for context resolution and several other schemes on a number of programs will be provided in the next chapter.

5.1. Static Data Dependency Analysis (SDDA)

This scheme is proposed in [Chang&Despain 85]. It is based on the construction of a set of data dependency graphs, which give the backtrack point for each subgoal at compile time. When a subgoal fails, the backtrack point associated with the failed subgoal can be used for backtracking. Since this piece of information is readily available before execution, it has almost no run time overhead. However, SDDA can only backtrack intelligently within a clause and it cannot react favorably to changing run time conditions.

During SDDA, a data dependency graph (DDG) is generated for each clause in the program. The user has to provide an activation pattern, in the form of a Prolog predicate (called *entry*), which indicates the potential status of the arguments in the top-level goal. Possible activations are '*i*' (independent), '*g*' (grounded) or '*c*' (coupled)†. In the graph, each subgoal in the body of a clause is represented by a node. A directed arc is

† A term is grounded if it does not contain any unbound variable. Two terms are coupled together if they shared at least one common, unbound variable. A term is independent if it is neither a ground term nor a coupled term.

drawn from the generator† of a variable to all the consumers of that variable and finally all duplicate arcs are removed. From this set of graphs, the backtrack point of each subgoal is identified. Since the graphs are constructed before run time, the backtrack subgoals are chosen in a worst-case manner to guarantee completeness during execution.

```
entry(color(i,i,i,i,i));

color(A,B,C,D,E) ←
        next(A,B) & next(A,C) & next(A,D) & next(B,C) &
        next(C,D) & next(B,E) & next(C,E) & next(D,E);

next(X,Y) ← next1(X,Y);
next(X,Y) ← next2(X,Y);

next(green,red);
next(green,yellow);
next(green,blue);
next(red,blue);
next(red,yellow);
next(blue,yellow);

next2(X,Y) ← next1(Y,X);
```

Program 5.1: A Map Coloring Prolog Program

Two types of backtrack paths are distinguished. A Type-I backtrack path is obtained when the end of a forward state of an execution (that is, continuation of deduction steps) is reached. This occurs when a subgoal cannot unify with any available candidate clauses. The Type-I backtrack subgoal of a subgoal is just its closest predecessor

† The *generator* of a variable X in a predicate is the predicate at which X gets bound to a term. All the other predicates which contain X and are not the generators of X are called the *consumers* of X.

node in the DDG. A Type-II backtrack path is obtained during the backward execution of a clause (that is, backtracking into a clause). An algorithm is given to determine the Type-II backtrack subgoals. The idea is that the Type-II backtrack subgoal of a subgoal s is the closest subgoal which lies in any of the possible backtracking paths that go through s. As an example, consider Program 5.1 from [Chang&Despain 85], when backtracking is done into the subgoal *next(B,E)* from *next(C,E)* or *next(D,E)*. In the former case, the backtrack path is *next(B,E)* (generator of E) → *next(A,C)* (generator of C) → *next(A,B)*. In the latter case, the backtrack path is *next(B,E)* (generator of E) → *next(A,D)* (generator of D) → *next(A,B)*. For worst-case consideration, the Type-II backtrack subgoal for *next(B,E)* is *next(A,D)*. Figure 5.1 is the corresponding DDG and Figure 5.2 shows the Type-I and Type-II backtrack paths.

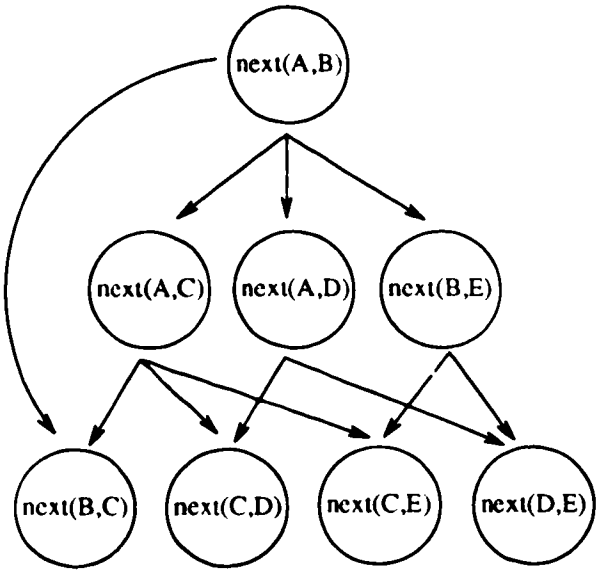
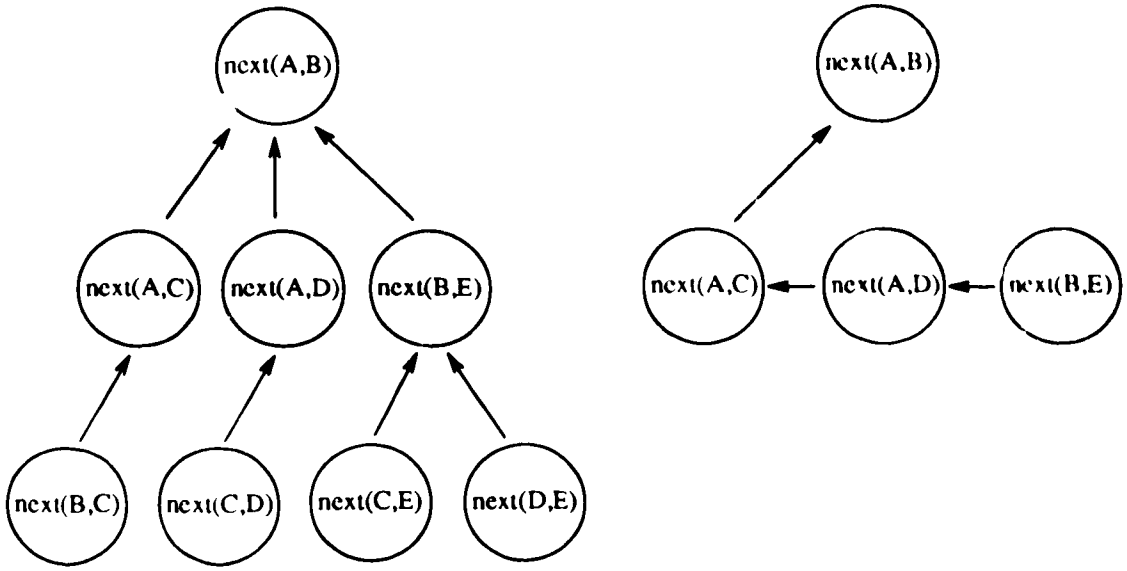


Figure 5.1: A Data Dependency Graph for Program 5.1



Type-I Backtrack Path Type-II Backtrack Path

Figure 5.2: Type-I & Type-II Backtrack Paths

While context resolution allows backtracking to anywhere in the search tree and requires no modification to the Prolog programs, SDDA can only handle intelligent backtracking within the same clause, and it requires the programmer to provide the necessary activation pattern for the generation of DDG. The major advantage of SDDA over all the other schemes is its low run time overhead and ease of implementation on a conventional architecture. The authors have incorporated SDDA into a PLM Prolog machine [Dobry,Patt&Despain 84]. They have also provided the number of PLM instructions executed on a number of benchmark programs. Their results are included in the next chapter.

5.2. Generator-Consumer Analysis (GCA)

This scheme is proposed in [Kumar&Lin 88]. It is based on run time variable consump-

tion analysis. An algorithm is presented which will determine the backtrack subgoal when a subgoal fails. Since this analysis is done during program execution, it can better respond to changing run time situations. A local data structure called a B-list is created for each subgoal P_i. B-list(P_i) is used to store all the potential backtrack points when P_i fails.

The basic idea of GCA is similar to the idea of context resolution. Both are based on tagging. That is, whenever a subgoal P_i is unified with the head of a clause, which results in assignment of a value (which can be a simple constant, a structured term or another variable) to a variable V, a tag is attached to V. There is a set of subgoals, gen(V), which contains all the subgoals which are responsible for the current value of V. If now P_i fails to unify with any clauses, then clearly the current value of one or more of the variables in P_i is not satisfactory. Let n be the number of distinct variables in P_i and X_j, $1 \le j \le n$, be the variables. The binding of X_j will not change unless we backtrack into at least one of the subgoals in *modifying* (P_i) = {parent of P} $\cup$ {gen(X_j) | $1 \le j \le n$}. To guarantee completeness, the most recent subgoal in modifying(P_i) is chosen for backtracking.

The complete algorithm is summarized as follows:

- When a subgoal P_i is first invoked, B-list(P_i) $\leftarrow$ {the parent of P_i}.
- Whenever P_i fails in a derivation, the algorithm performs the following steps:
  1. Temp $\leftarrow$ B-list(P_i) $\cup$ {gen(X_j) | $1 \le j \le n$} where X_j's are the variables that occur in the arguments of P_i.
  2. $P_{max} \leftarrow$ most recent choice point in Temp.
  3. B-list(P_{max}) $\leftarrow$ B-list(P_{max}) $\cup$ Temp $-$ {P_{max}}.
  4. Backtracks to P_{max}.

Consider Program 5.1 again, the Type-I backtrack subgoal is the same as that obtained by SDDA. The Type-II backtrack subgoal for *next*(*B*,*E*) (called this *P*) is *next*(*A*,*C*) (better than SDDA) when backtracking is done into *P* from *next*(*C*,*E*), otherwise backtracking is done into *P* from *next*(*D*,*E*) and the backtrack subgoal is *next*(*A*,*D*) (same as SDDA).

Another advantage of GCA over SDDA is its ability for *across-the-clause* backtracking. Suppose we have the clause (from [Kumar&Lin 88]):

$$s(X,Y) \leftarrow v(X) \;\&\; w(Y);$$

and the goal statement:

$$?\; s(X,Y) \;\&\; t(Y) \;\&\; u(X);$$

When the subgoal $u(X)$ fails, clause-level data-dependency-based backtracking schemes (for example, SDDA) will backtrack to *w*, whereas GCA is able to backtrack intelligently to *v*. The authors mention that their scheme "performed backtracking at the proof tree level", meaning that their scheme can backtrack to any subgoals in the *same* derived goal.

$p(a)$;
$p(b)$;

$q(X) \leftarrow r(X)$;

$r(b)$;

' $p(X)$ & $q(X)$;

Program 5.2

In contrast, context resolution can do even better. When a subgoal fails, context resolution is sometimes able to give a backtrack point which is less recent than the parent of the failed subgoal. Consider Program 5.2, when $r([a,\#p])$ fails, the context returned ($\#p$) indicates a backtrack point which is less recent than the parent of r (that is, q), whereas GCA can only backtrack to q since their algorithm always includes the parent of P whenever a subgoal P is invoked. This is also the reason why they have to keep the choice point for P around even if there is only one applicable clause available for P because their interpreter will sometimes backtrack into a deterministic subgoal. In modified context resolution, the interpreter will never backtrack into a deterministic node thus the choice point for a deterministic node is not required. This saves space (allocated for the choice point) as well as time (required for the manipulation of the extra control information in the deterministic node) during execution.

```
p(a);
p(b);

q(d);
q(e);

r(w);
r(d);

s(b);

? p(X) & q(Y) & r(Y) & s(X);
```

Program 5.3

Sometimes a local B-list can give better run time performance than a global B-list. Program 5.3 is an example showing that using a local B-list can be more intelligent than using a global one. When $r(Y)$ fails to unify with $r(w)$ with X bound to $[a,\#p]$ and Y bound to $[d,\#q]$, $\#q$ is inserted into the global B-list. Later when $s(X)$ fails and $\#p$ is inserted into B-list (which already contains $\#q$), the interpreter will backtrack into $\#q$ since $\#q > \#p$. This is undesirable as the interpreter should be able to backtrack into $\#p$ directly. If a local B-list is used, and again when $r(Y)$ fails to unify with $r(w)$, $\#q$ is put into the local B-list associated with $r(Y)$. Later when $s(X)$ fails and $\#p$ is inserted into the B-list associated with $s(X)$, there is no more available candidate clause for the subgoal s and the interpreter will backtrack into $\#p$.

We can incorporate local B-list in context resolution as given by the following algorithm:

- When a subgoal P_i is first invoked, $B\text{-list}(P_i) \leftarrow \varnothing$.

- Whenever P_i fails to unify with the head of a candidate clause and a list of contexts which is associated with the function symbol which caused the failure is returned in failure_contexts,

  $B\text{-list}(P_i) \leftarrow B\text{-list}(P_i) \cup$ failure_contexts.

- Whenever P_i fails in a derivation, the algorithm performs the following steps:

  1. $P_{max} \leftarrow \max(B\text{-list}(P_i))$.
  2. $B\text{-list}(P_{max}) \leftarrow B\text{-list}(P_{max}) \cup B\text{-list}(P_i) - \{P_{max}\}$.
  3. Backtracks to P_{max}.

While the local B-list approach has higher intelligence for some Prolog programs, the sorting and merging operations are essential unless a large amount of memory (the total size of the context array) is set aside for each subgoal or a complex scheme is devised which will differentiate all the contexts in a global context array. We sacrifice the intelligence gained for algorithm clarity and implementation simplicity†.

The main weakness in GCA is that it sometimes fails to backtrack directly to a subgoal which causes the failure. During the generation of a B-list, whenever a predicate P fails, they will consider the generators of *all* the variables that occur in the argument of P and backtrack to the most recent one. This backtrack point may not be the one that caused the unification failure. Thus P will fail again. Although they have also considered the handling of 'cut' in their scheme, it cannot handle the case when 'cut' is used for non-monotonic reasoning. Furthermore, they have omitted the discussion of the implementation of many other constructs which are indispensable in almost every real-life Prolog program, for example, 'prove', 'not', 'assert' and 'retract' in theorem provers, expert systems and database programs.

† See Appendix I for the discussions on local B-lists.

The tagging processes in GCA and context resolution are very similar since both schemes keep track of binding of variables at the function symbol level. Thus almost the same amount of work is done in each unification step. During a shallow backtracking, an intelligent interpreter incorporated with context resolution has to insert in the *B*-list all the contexts associated with the function symbol which causes the failure, while nothing has to be done in GCA. However, when a deep backtracking occurs, context resolution requires no further analysis while in the scheme of [Kumar&Lin 88], some analysis has to be performed to reveal the next backtrack point. Thus context resolution, when compared to the GCA, enables a more accurate analysis of the cause of failure, while it requires similar amount of extra work during execution.

5.3. Deduction Analysis (Minimal Unifiable Subsets)

This scheme is presented by Lee Cox in [Cox 84, Cox 87]. It is based on run time unification failure analysis. It is more intelligent than SDDA or GCA in the sense that this scheme eliminates the largest amount of search space in theory. However, the scheme is not practical in itself because of the large computational overhead involved.

During the execution of a Prolog program, a derivation failure occurs when a subgoal cannot unify with any of the candidate clauses. All the derivations performed so far are represented by a set of constraints. Each constraint c_i is an ordered pair (S_i, C_i) where S_i is the subgoal to be executed and C_i is the candidate which is used to solve S_i. A unification failure corresponds to the situation where all the constraints cannot be satisfied simultaneously. A naive interpreter backtracks by removing the most recent deduction performed (one of the constraints) so that unifications can be carried on. However this can easily lead to thrashing since the most recent backtrack point chosen by the interpreter may not be the one that will introduce new bindings in the

subgoal that fails, thus the subgoal will fail again. Cox's scheme tries to analyze the cause of the unification failure and determines the maximal unifiable subsets, subsets of the set of unifiable constraints but not properly contained in any other unifiable subsets. The maximal subset is chosen so that the maximum amount of work done can be saved by removing as few constraints as possible. Note that the maximal unifiable subsets will not provide enough backtrack information for an intelligent interpreter. The backtrack point to be chosen (that is, the corresponding constraint to be removed) depends on the structure of the proof tree and the availability of alternative candidate clauses for that particular constraint.

There is a close relationship between Cox's concept of maximal unifiable subset and context resolution. We will first transform all derivations to constraints by the following scheme. The *class* of a constraint $c_i = (S_i, C_i)$ is defined to be i where S_i is the leftmost subgoal at the goal statement G_i and C_i the head of the candidate clause used to solve S_i. Note that by this scheme, the parent of a subgoal and the subgoal itself will appear in different classes in the set of constraints. As an example consider execution of Program 5.4.

```
p(a);
p(b);

q(T) ← u(T).
q(c);

u(a);

r(V,V);

s(1);
s(2);

t(b,c);

? p(X) & q(Y) & r(Y,Z) & s(W) & t(X,Z);
```

Program 5.4

The execution sequence of the derivation where $t(X,Z)$ is the leftmost subgoal corresponds to the con-

c_0: $(p(X), p(a))$,

c_1: $(q(Y), q(T))$,

c_2: $(u(T), u(a))$,

c_3: $(r(Y,Z), r(V,V))$,

c_4: $(s(W), s(1))$,

c_5: $(t(X,Z), t(b,c))$.

The three maximal unifiable subsets computed by applying Cox's algorithm are $\{c_0, c_1, c_2, c_3, c_4\}$, $\{c_1, c_3, c_4, c_5\}$ and $\{c_2, c_3, c_4, c_5\}$. Notice that the first subset corresponds to shallow backtracking. That is, look for alternative candidate clause that is unifiable with the most recent subgoal $t(X,Z)$. The second subset corresponds to the situation where backtracking is done into p (c_0) and u (c_2), and the third subset

corresponds to backtracking into p (c_0) and q (c_1).

A naive Prolog interpreter will try all the possibilities for Y, and the same failure in $t(X,Z)$ will occur. While Cox's scheme has essentially eliminated any irrelevant backtrack points, his algorithm does not specify how the next backtrack point is chosen. In addition, for the purpose of intelligent backtracking, only one maximal unifiable subset is needed.

Suppose the interpreter encountered a derivation failure and backtracking is required. Let $C = \{c_0, c_1, \ldots, c_n\}$ be the set of all constraints corresponding to the derivation and $D = \{\#j_0, \#j_1, \ldots, \#j_t\}$ be the set of contexts returned during the derivation at G_n where $0 \le t \le n$ and $\#j_i$ $(0 \le i \le t)$ refers to the derivation at G_{j_i} (that is, the constraint c_{j_i}). Note that all the constraints from c_0 to c_{n-1} can be satisfied. It is easy to see that the set $E = C - \{c_{j_t}\}$ is a maximal unifiable subset of C since the function symbol which involved in the unification failure when the constraint c_n is being solved is first introduced at G_{j_0}. The symbol is introduced again during the goals $G_{j_1}, \ldots, G_{j_t}$. When one of these links is broken, the function symbol which causes the unification failure at G_{j_t} is not able to reach G_n and hence will not cause the same failure again. Thus c_n is now satisfied and hence E is a maximal unifiable subset of C.

In the above discussion, we have considered unification failure caused by one function symbol only. It is straightforward to apply similar argument to multiple conflicts in general. When c_n fails, each function symbol f_i which causes unification failure will be associated with a list of contexts D_i where $1 \le i \le m$ and m is the number of function symbols which causes the failure at G_n. The set $E = C - \{x_1, x_2, \ldots, x_m\}$ where $x_i \in D_i$ can be shown to be a maximal unifiable subset of C.

Although Cox's scheme can handle the occurs check, and is suitable for parallel execution and theorem proving in general, even the author admits that the algorithm is not applicable in practice owing to the extensive computation involved during the analyses of graphs. In fact, the problem of finding all maximal unifiable subsets is shown to be NP-hard [Wolfram 86]. However, Cox's algorithm and analysis provide a good foundation for research in intelligent backtracking, that is, they can be used as a theoretical basis for the development of approximate or heuristic algorithms which are more efficient, for example [Codognet,Codognet&Filè 88]. In addition, we have shown that the maximal unifiable subsets alone do not provide enough information for backtracking and that context resolution is just an implementation-oriented realization of Cox's scheme applied to the execution of Prolog programs in the sense that our scheme shows how the relevant maximal unifiable subset and the backtrack point are "computed" without undue overhead.

5.4. Deduction Analysis (Minimal Non-Unifiable Subsets)

This scheme, which is presented in [Bruynooghe&Pereira 84], is complementary to Cox's scheme. Their approach of analyzing a failure is based on finding the minimal non-unifiable subtree.

The idea of this scheme is to associate a deduction tree with each term involved in the unification. For a subgoal, they associate the nodes necessary for the existence of that call, that is, the subgoals where a variable gets a binding. With the head of the candidate clause, they associate the empty deduction tree. The following algorithm specifies the actions taken during unification, where $t-T$ denotes a term t with associated deduction tree T:

1. matching t–T_1 with t–T_2: generate the empty substitution with deduction tree $T_1 \cup T_2$.
2. matching $f(t_1, .., t_n)$–T_1 with $g(t_1, .., t_m)$–T_2: generate 'failure' with $T_1 \cup T_2$ as an inconsistent deduction tree.
3. matching $f(t_1, ..., t_n)$–T_1 with $f(r_1, ..., r_n)$–T_2: match each t_i–T_1 with r_i–T_2.
4. matching X_1–T_1 with t_2–T_2 where t_2 is not a free variable and a substitution X_1/t–T exists: match t–$T \cup T_1$ with t_2–T_2.
5. matching X_1–T_1 with t_2–T_2 with X_2 a free variable: generate the substitution X_1 / t_2–$T_1 \cup T_2$

The deduction tree used in their unification algorithm is very similar to a tag in GCA or a context in context resolution. A unification failure results in a non-unifiable deduction tree. When a deep backtracking is required, the set of all non-unifiable subtrees are analyzed to give the minimal subtree which will give an intelligent backtrack point.

Although they provide a more complete description of the implementation details, this scheme suffers from the same shortcoming as in Cox's scheme, that is, large run time overhead. This makes it only advantageous in some specialized applications. Since the overhead of their full theory is too large for the sequential execution of most Prolog programs, they have incorporated a simplified version of the theory in a Prolog interpreter. Their run time statistics will be presented and analyzed in the next chapter.

There is a close relationship between maximal unifiable subset and minimal non-unifiable subset since one can be readily computed from the other by a simple algorithm presented in [Chen,Lassez&Port 86]. In addition, the authors mention that both approaches complement each other and are not fundamentally different.

5.5. Depth-First Intelligent Backtracking

This scheme is presented in [Codognet,Codognet&Filè 88]. It is based on the construction of *dynamic conflicts graphs* which represent the substitutions computed. This method differs from the method by [Bruynooghe&Pereira 84] mainly in the notions used in describing the backtracking scheme. The authors think that their notions are more precise and they formally prove the correctness about the method. In addition, they implement a simplified version of their scheme in order to reduce the overhead of the original method. The execution statistics are provided in the next chapter for comparison.

Algorithms are provided to construct a dynamic conflicts graph from a set of constraints C. The construction of this graph corresponds to the unification process. In addition to representing the most general unifier of C when C is solvable, the graph contains information about which constraint is responsible for each binding. When a clash (function symbols conflict) is detected, the constraints that caused it can be identified by the DIB algorithm, which is essentially a modified Prolog interpreting algorithm. An infinite term during derivation corresponds to a circular dynamic constraint graph. In this case, the DIB algorithm simply reports failure and stops.

While DIB has a high intelligence, it is not clear as to how their scheme can be incorporated into standard unification and resolution procedures. Thus an implementation of their scheme is non-trivial. In addition, their method is based on graph computation, which can be expensive. The most notable feature in DIB is the notion they choose to present the scheme, which make it possible to formally prove its correctness. In addition, the ability to handle the 'cut' and many system predicates intelligently makes DIB a practical real-life Prolog system.

5.6. Summary

In this chapter, we have looked at some of the well-known intelligent backtracking schemes. Among all the schemes, static data dependency analysis has the lowest overhead. However, it is not able to cope with changing run time conditions thus its intelligence is lowest. Generator-consumer analysis entails much less overhead, but it sometimes fails to backtrack directly to the subgoal which caused the failure. While Cox's and Bruynooghe and Pereira's schemes which are based on unification failure and remove the largest amount of search space in theory, the intensive computational overhead limits their practical use. The intractability results by Wolfram for computing all maximal unifiable subsets (and correspondingly all minimal non-unifiable subsets) seem to further question the applicability of these methods without resorting to heuristics. Depth first intelligent backtracking is a practical implementation of Cox's and Bruynooghe and Pereira's schemes. However, it is not trivial to incorporate this scheme into standard Prolog systems. We have also shown that context resolution is a resolution based approach to determine the maximal unifiable subset and the intelligent backtrack point.

Chapter 6

Performance Results & Evaluations

After context resolution was incorporated into WUP, the modified interpreter was tested on a variety of problems. The same set of problems is used in the evaluation of the backtracking schemes of [Bruynooghe&Pereira 84], [Chang&Despain 85], [Kumar&L 88] and [Codognet,Codognet&Filè 88] so that a comparison can be made. In the first section, we will discuss the benchmark programs used in the evaluation. The amount of extra memory used in the implementation is presented in section two and the execution time statistics are presented in section three. Finally, an analysis of the performance is presented.

6.1. The Benchmark Programs

The benchmark consists of the following Prolog programs, whose sources are provided in Appendix I.

Database Query

> This program poses a complex query to a small database which describes the courses taken by students, the courses taught by professors and the date and place of lectures. Only simple constants are involved in the program.

Naive N-Queen Problem

> This program solves the N non-attacking queens problem by the generate-and-test approach. Lists and functors are used.

Clever N-Queen Problem

> This program solves the same N-queen problem in a clever way. The predicates 'cut' and 'fail' are used extensively.

Tree Insertion Problem

> This program inserts a list of numbers into an ordered binary tree. The program

is almost deterministic because backtracking is always done to the most recent backtrack point.

Clever Map Coloring Problem

This program colors a 13-region map with four different colors so that no two adjacent regions have the same color. The subgoals are arranged in such a way that the same variables are situated in adjacent subgoals, if possible. That is, the consumer of a variable is placed as close to its generator as possible.

Simple Map Coloring Problem

This program solves the same map coloring problem. The subgoals are arranged without paying any attention to the ordering of the variables. In this way, unification failure of a variable may be discovered much later in the execution than in the previous program.

Move Checking

This program checks whether a move generated by a predicate is legal with respect to some given constraints. The subgoals in the goal statement are ordered in two different ways. [Kumar&Lin 88] uses this example to show that there are programs for which intelligent backtracking improves the performance irrespective of the ordering of the subgoals.

6.2. Memory Usage

In the current implementation, the Copy Stack, Run Time Stack and Trail each used 2000 PC_WORDs. A 1000-word context list is used for the associated contexts and a 500-word context array is used for the failure contexts.† Together with the miscellaneous variables used, this give an increase in memory usage of approximately 30%. All the benchmark programs can be executed using this arrangement without memory overflow. Since memory usage is not the primary concern of this implementation, no memory reclamation routine besides that supplied by the system is used.

---

† Since the hashing function divides an address (in number of PC_WORDs) by 4, when we use a N-word Run Time Stack, we need to allocate an N/4-word context array.

6.3. Run Time Statistics

The results are presented in two tables. Table 6.1 is a summary of the comparison of the execution statistics of each program. The parameters to be tested are:

- The number of inference steps performed (*# inferences*) during the execution of the goal.
- The number of unifications attempted (*# unifications*) during the execution.
- The number of deep backtrackings occurred (*# backtracking*) during the execution.
- The run time of a program.

The run time of a program is obtained by measuring the amount of time spent on unification alone. That is, all the time spent on parsing, generation of clause structures and I/O are ignored. This is done by calling the UNIX *times* command which returns the CPU time used while executing instructions in the user space of the program. *times* is called just before entering the unification routine and it is called again just after leaving it. Thus the amount of time spent in the routine can be determined.

The results are obtained by running the programs on an idle Sun-3/75 work station. All the programs have very short running times, except the simple map coloring program. Thus to obtain a more reliable timing measurement, they are queried several times and the total time spent is recorded. Then the same programs are run on an unmodified interpreter and the two sets of times are compared to give the results. In Table 6.1, the second column contains the statistics when the programs are run on an unmodified interpreter. The third column contains the statistics when the programs are run on a modified interpreter with intelligent backtracking enabled. In the fourth column, the percentages changed in performance of the intelligent scheme with respect to the naive scheme are entered. As an example, the first entry is calculated as: $\frac{\#\ inference_{intelligent} - \#\ inference_{naive}}{\#\ inference_{naive}} * 100\%$. Thus a negative percentage means a

speed up and a positive figure means a slow down. The row *time in ms (#)* gives the execution time in milliseconds and the number of times a program is queried to give that figure.

Table 6.2 is a comparison of the speed up (with respect to the execution times) with some other schemes presented in Chapter five. In each column, *CPU* refers to the CPU time speed up, *# cycles* refers to the speed up in terms of the number of PLM machine cycles and *# instrs* refers to the speed up in terms of the number of PLM instructions executed. When an entry is marked with N/A, this means that the program is not tested on a particular scheme. The speed up of Kumar and Lin's scheme in terms of both CPU and machine cycles are reported in the table as the authors mentioned that the CPU time taken by their simulator may not be accurate and that only the number of machine cycles can give an accurate picture.

Table 6.1: Comparison of Execution Statistics

Problem	Naive WUP	Intelligent WUP	% Change
database query			
# inferences	83	42	-49.4%
# unifications	152	66	-56.6%
# backtrackings	44	13	-70.5%
time in ms (#)	10416(100)	2500(100)	-76.0%
6-queen simple			
# inferences	7651	2402	-68.6%
# unifications	9959	2628	-73.6%
# backtrackings	1430	55	-96.2%
time in ms (#)	6283(1)	1350(1)	-78.5%
6-queen clever			
# inferences	3831	3831	0.0%
# unifications	5784	5784	0.0%
# backtrackings	806	806	0.0%
time in ms (#)	2033(1)	2300(1)	+13.1%
7-queen simple			
# inferences	6296	803	-87.2%
# unifications	8425	861	-89.8%
# backtrackings	1340	11	-99.2%
time in ms (#)	4866(1)	383(1)	-92.1%
7-queen clever			
# inferences	1044	1044	0.0%
# unifications	1578	1578	0.0%
# backtrackings	210	210	0.0%
time in ms (#)	516(1)	550(1)	+6.6%
tree insertion			
# inferences	217	217	0.0%
# unifications	308	308	0.0%
# backtrackings	30	30	0.0%
time in ms (#)	1266(10)	1400(10)	+10.6%
map color clever			
# inferences	44	44	0.0%
# unifications	92	87	-5.4%
# backtrackings	12	9	-25.0%
time in ms (#)	2950(100)	3183(100)	+7.9%

Table 6.1: Continue

Problem	Naive WUP	Intelligent WUP	% Change
map color simple			
# inferences	89250	17526	-80.4%
# unifications	270644	36096	-86.7%
# backtrackings	57897	2810	-95.2%
time in ms (#)	80066(1)	10383(1)	-87.0%
move ordering 1			
# inferences	409	154	-62.3%
# unifications	464	168	-63.8%
# backtrackings	55	14	-74.5%
time in ms (#)	2166(10)	700(10)	-67.7%
move ordering 2			
# inferences	295	145	-50.8%
# unifications	331	159	-52.0%
# backtrackings	36	14	-61.1%
time in ms (#)	1466(10)	650(10)	-55.7%

Table 6.2: Comparison With Related Research

Problem	DTA (CPU)	GCA (# cycles)	GCA (CPU)	SDDA (# instrs)	DIB (CPU)	CR (CPU)
database query	-19.4%	-48.9%	-55.4%	-16.0%	-52.4%	-76.0%
6-queen simple	-34.9%	+162.8%	+16.6%	N/A	-61.5%	-78.5%
6-queen clever	+99.3%	+90.6%	+8.6%	N/A	+10.0%	+13.1%
7-queen simple	-82.7%	N/A	N/A	N/A	-89.2%	-92.1%
7-queen clever	+103.9%	N/A	N/A	N/A	+20.0%	+6.6%
tree insertion	+43.8%	+9.0%	0.0%	N/A	+30.0%	+10.6%
mapcolor clever	+63.5%	+7.8%	-3.9%	+0.7%	N/A	+7.9%
mapcolor simple	-99.7%	-99.9%	-99.9%	-99.9%	N/A	-87.0%
move ordering 1	N/A	-54.1%	N/A	N/A	N/A	-67.7%
move ordering 2	N/A	-33.9%	N/A	N/A	N/A	-55.7%

Legends:

DTA	Deduction Tree Analysis	GCA	Generator-Consumer Analysis
SDDA	Static Data Dependency Analysis	CR	Context Resolution
DIB	Depth-First Intelligent Backtracking		

6.4. Analysis

According to Table 6.2, our scheme clearly gives better speed up compared to GCA (# cycles) except for the tree insertion program where context resolution gives a slightly larger overhead, and the simple map coloring problem where context resolution gives a less dramatic speed up. While DIB and context resolution have similar intelligence in theory, context resolution gives better performance results over DIB except for the clever 6-queen program. This is possibly due to the fact that DIB is based on graph computation and thus the overhead can sometimes grow quite fast. For example, consider the tree insertion program where the overhead is +30%. While DTA has similar intelligence as context resolution, its large run time overhead often overshadows the reduction in the number of deductions performed. In terms of the number of inferences, however, the two schemes should be very close. SDDA has the lowest overhead in the clever map coloring problem. However, the fact that it is not able to deal with changing run time conditions can be revealed by considering the database query where SDDA gives the worst performance. This is due to the worst-case analysis performed by SDDA.

Our implementation performs exceptionally well over any other schemes in the first two problems. This is mainly due to the presence of the 'cut', 'fail' and 'not' constructs in these programs which can all be handled intelligently by our system. In addition, the implementation gives an almost uniform overhead (under 14%) for programs which do not benefit much from intelligent backtracking. (That is, programs which are deterministic or where backtrackings are done to the nearest non-deterministic node. For example, the tree insertion and the clever N-queen programs.) Compared to the maximum overhead of +103.9% in DTA, +162.8% in GCA (# cycles) and +30.0% in

DIB, context resolution is a more stable scheme.

6.5. Summary

According to the performance results, DTA involves too much computational overhead and deterministic or almost deterministic programs which exhibit very little backtrackings will give an intolerable amount of overhead. The superior performance on the first two programs for context resolution over GCA, DTA and SDDA indicates the latter schemes' inability to handle the 'cut' and 'not' constructs intelligently. Context resolution almost always performs better than GCA due to the latter scheme's failure to backtrack directly to the relevant subgoal which caused the unification failure. In terms of the number of machine cycles executed, which the authors claim is more accurate, the speed up is even more dramatic. For non-deterministic programs, DIB gives a similar performance results as context resolution. However, the overhead of DIB is still non-trivial for some deterministic or clever programs.

Chapter 7

Conclusion

In this chapter, a summary of the thesis is given, then several important application areas of context resolution are discussed. Finally, we look at one of the extensions that can be done, and propose a list of future research topics.

7.1. Summary of the Thesis

Prolog is a programming language based on the resolution principle of mechanical theorem-proving. One of the major attractions of Prolog is ease of programming. Instead of providing the computer with a sequence of instructions to be executed, a Prolog programmer needs to describe only the logical component of a task. This separation of logic and control components allows a programmer to write concise and readable programs quickly. However, it is generally agreed that Prolog is an inefficient language in terms of run time and memory requirements. One of the major causes of inefficiency is Prolog's exaggerated anticipation of backtracking which makes it less usable than most machine-oriented languages now in popular use, for example, Fortran and C.

In a conventional implementation of Prolog, the interpreter will exhaustively search a proof tree in a depth-first, left-to-right manner. When a failure leaf is encountered, the interpreter will back up to the most recent node with untried alternative clause(s) and start the searching process again on the next branch. Most often, a unification failure is due to the bindings created much earlier in the proof tree and thus the "same" unification failure may be repeated several times before a solution is found. This behavior of Prolog motivates the study of methods for improving the efficiency of

the backtracking mechanism of this language.

Previous works on intelligent backtracking can be divided into two categories, based on the tools used in the analysis. [Cox 84] and [Bruynooghe&Pereira 84] use *deduction trees* to analyze the possibility of unification of a subtree. These analyses and the corresponding algorithms essentially laid the foundations for intelligent backtracking. However, the resulting algorithms are not practical [Cox 84] and can be used only as the basis for the development of more efficient algorithms. A more efficient implementation of Bruynooghe and Pereira's scheme is presented in [Codognet,Codognet&Filè 88]. Their scheme is also based on graph computations and thus requires a possibly intensive modification to the existing Prolog system. Furthermore, the overheads of the scheme on some programs are still quite high.

In the second category, [Chang&Despain 85] and [Kumar&Lin 88] use a *data dependency graph* which indicates the backtrack points by the generator-consumer approach. While the static scheme of [Chang&Despain 85] cannot respond favorably to run time conditions, the scheme of [Kumar&Lin 88] sometimes fails to locate the cause of failure *directly*. Our analytical discussion in Chapter five and the performance results in Chapter six both show that data dependency analysis is less intelligent than unification analysis.

This thesis describes an intelligent backtracking method called *context resolution*, discusses its implementation in Waterloo Unix Prolog, and presents some performance results. The main features of this improved Prolog system are:

high degree of intelligence

Referring to the execution statistics in Chapter six, context resolution compares very favorably with many other schemes when the interpreter is executing some generate-and-test problems which rely heavily on backtracking. From the analysis

in Chapter five, we have shown that context resolution is as intelligent as those schemes which are based on unification failure analyses. In addition, owing to the resolution-based property of context resolution, intelligent backtracking can be incorporated into any standard Prolog systems without any expensive graph computations.

low and stable overhead

According to the execution statistics, even when the interpreter is executing deterministic Prolog programs, or when backtrackings are always done to the most recent choice point with alternative(s), the run time overhead is consistently under 14%. While another scheme may exhibit exceptionally low overhead on some particular programs, the overhead of the same scheme on other programs may be prohibitively high. (For example, the overhead for GCA ranges from +7.8% to +162.8%.) In contrast, context resolution gives more stable performance.

complete implementation

Most of the other schemes have ignored the handlings of the impure constructs of Prolog, which many programmers use to improve the language's usability. This implementation can handle almost all commonly-used impure constructs. The especially outstanding performance for this scheme on the database query and the simple N-queen problem suggests that programs which are written without paying much attention to the control component, possibly with some impure constructs, are particularly suitable to be run on this intelligent interpreter.

transparency and flexibility

This implementation is transparent to a user since no modification on a source program is required before it is executed by the intelligent interpreter. To obtain the most efficient execution time, the interpreter provides a pair of switches, which activates or deactivates the association process and the manipulation process respectively. When a user knows that a program will not benefit much from intelligent backtracking, (s)he can deactivate the mechanism by turning off both switches and no overhead will entail. If only the association process is enabled, a user can activate intelligent backtracking within a specified sequence of subgoals.

We have shown that context resolution correctly captures the minimal amount of information required for intelligent backtracking. (Examples are parent/offspring relationship, deterministic/non-deterministic activations of subgoals and failure-originating

bindings of variables.) This scheme has been incorporated into slightly modified resolution and unification algorithms. Our implementation shows that intelligent backtracking can be implemented on top of a conventional Prolog system without undue overhead. Different implementations are, of course, possible and the efficiency can be further improved by recognizing the properties of a particular system. The performance results show that context resolution achieve a tremendous speed up over a conventional interpreter with naive backtracking for most non-deterministic problems. Even for completely deterministic programs, the overhead is only around 14%. Although the results are highly problem-dependent, there is strong evidence that context resolution will perform well for any arbitrary programs. We believe that context resolution can be implemented on any conventional Prolog systems, which is a major step towards making Prolog a more practical programming language for everyday applications.

7.2. Applications of Context Resolution

The extensive utilization of Prolog in solving a wide range of problems calls for an efficient backtracking scheme. In this respect, context resolution provides an efficient resolution-based procedure which can be incorporated into many systems easily. We will look at some of the application areas of context resolution. In particular, one of the applications is in Prolog program debugging.

7.2.1. General Applications

Prolog has been used successfully in applications such as theorem proving, symbolic integration, plan formation, CAD, compiler constructions and expert systems. Context resolution allows these programs to be executed several times faster with little or no modification to the source programs. While some new logic programming systems, for

example, constraint logic programming [Van Hentenryck 87] (CLP) may give better performance on some problems†, it is often necessary for us to partially rewrite a program written for Prolog so that it can be run on a particular system efficiently and vice versa. Quite often, these modifications are non-trivial. In addition, to incorporate these new schemes in an existing Prolog system, we need to rewrite a considerable portion of the system. Thus context resolution requires minimal work from the programmers or implementors while giving very promising results.

7.2.2. Debugging

One highly useful application of context resolution is in Prolog program debugging. Since a context represents an address in the proof tree where a term instantiation takes place, we are keeping track of the history of the execution in an explicit way. Given a context term t, we can easily identify all the subgoals which are executed to instantiate the terms in t by considering the contexts associated with t. (This idea is similar to that of [Mannila&Ukkonen 86].) Whenever a unification failure occurs and a list of contexts is returned, this list reveals the places where a clause is wrongly declared or where a missing clause should be added. Context resolution is a far more superior scheme than conventional tracing procedure which gives all kinds of irrelevant informations. Note that when context resolution is used in program debugging, we can no longer omit contexts which correspond to deterministic nodes, since now every context corresponds to a node in the proof tree. A failure context which corresponds to a deterministic node may indicate the place where a clause should be added to correct the failure.

† A CLP program can solve the 5-queen puzzle without any backtrackings.

7.3. Extension

The main weakness in the scheme is its inability to intelligently handle some programs when the 'assert' subgoal appears *after* a deep backtracking. We suggest that a pre-processor which scans a program for any 'assert' predicates before execution will cure this problem. This pre-processor can give backtracking information to the intelligent interpreter when a deep backtracking occurs and choose the next backtrack point by taking the 'assert' predicate into account.

7.4. Future Work

Intelligent Debuggers

This is a major research topic which has much to be done. One of the most well-known work in this area is [Shapiro 83]. We have seen why context resolution is particularly suitable for Prolog program debugging. We suggest that context resolution can be incorporated into an interactive debugger to give a powerful high-level debugger which can detect, locate and possibly correct program bugs.

Intelligent Compilers or Prolog Machines

Numerous Prolog compilers or Prolog machines still employ the naive backtracking mechanism. It is an interesting research topic to investigate the possibility of integration of context resolution into them. The 14% slow down for some deterministic programs is a small price compared to the tremendous speed up for most programs which are non-deterministic. Even dedicated Prolog machines are not able to compete with this scheme once a program becomes difficult (like most AI problems). We hope that various concepts addressed in this thesis can be applied to Prolog compilers and Prolog machines as well.

Parallel Execution Schemes

Intelligent backtracking is required even in a parallel execution environment. We believe that context resolution can be useful in administrating backtracking in AND-parallelism.

References

[Bruynooghe 82]
M. Bruynooghe, The Memory Management of Prolog Implementation, in *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K. L. Clark and S.-Å. Tärnlund (eds.), Academic Press, London, pp. 83-98, 1982.

[Bruynooghe 82a]
M. Bruynooghe, A Note on Garbage Collection in Prolog Interpreters, *Proceedings of the First International Logic Programming Conference*, Marseille, France, pp. 52-55, 1982.

[Bruynooghe&Pereira 82]
M. Bruynooghe and L. M. Pereira, Deduction Revision by Intelligent Backtracking, Implementations of Prolog, J. A. Campbell (ed.), Ellis Horwood, pp. 194-215, 1984.

[Chang&Despain 85]
J.-H. Chang and A. M. Despain, Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis, *Proceeding of IEEE Symposium on Logic Programming*, pp. 10-21, 1985.

[Chen,Lassez&Port 86]
T. Y. Chen, J.-L. Lassez and G. S. Port, Maximal Unifiable Subsets and Minimal Non-unifiable Subsets, *New Generation Computing*, vol. 4, no. 2, pp. 133-152, 1986.

[Cheng 84]
M. H. M. Cheng, Design and Implementation of the WUP Environment, Master of Mathematics (Computer Science) Thesis, University of Waterloo, Canada, 1984.

[Codognet,Codognet&Filè 88]
C. Codognet, P. Codognet and G. Filè, Yet Another Intelligent Backtracking Method, *Proceedings of the Fifth International Conference and Symposium*, R. A. Kowalski and K. A. Bowen (eds.), 1988.

[Cox 84]
P. T. Cox, Finding Backtrack Points for Intelligent Backtracking, Implementations of Prolog, J. A. Campbell (ed.), Ellis Horwood, pp. 216-233, 1984.

[Cox 87]
P. T. Cox, On Determining the Causes of Non-unifiability, *Journal of Logic Programming*, vol. 4, no. 1, pp. 33-58, 1987.

[Dobry,Patt&Despain 84]
T. Dobry, Y. Patt and A. Despain, Design Decisions Influencing the Microarchitecture for a Prolog Machine, MICRO 17 proceedings, Oct. 1984.

[Dobry,Despain&Patt 85]
T. Dobry, A. Despain and Y. Patt, Performance Studies of a Prolog Machine Architecture, *Proceedings of IEEE Symposium on Computer Architecture*, pp. 180-190, Aug. 1985.

[Hogger 84]
C. J. Hogger, Introduction to Logic Programming, Academic Press Inc. (London) Ltd., 1984.

[Kowalski 74]
R. A. Kowalski, Predicate Logic as a Programming Language IFIP 74, pp. 569-574, 1974.

[Kowalski&Kuehner 71]
R. A. Kowalski and D. Kuehner, Linear Resolution with Selection Function, Artificial Intelligence 2, pp. 227-260, 1971.

[Kumar&Lin 88]
V. Kumar and Y.-J. Lin, A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog, *Journal of Logic Programming*, vol. 5, no. 2, pp. 165-181, 1988.

[Lin,Kumar&Leung 86]
Y.-J. Lin, V. Kumar and C. Leung, An intelligent backtracking algorithm for Parallel Execution of Logic Programs, *The Third International Conference on Logic Programming*, pp. 55-68, London, 1986.

[Lloyd 84]
J. W. Lloyd, Foundations of Logic Programming, Springer-Verlay, New York, 1984.

[Mannila&Ukkonen 86]
H. Mannila and E. Ukkonen, Timestamped Term Representation for Representing Prolog, *IEEE Symposium on Logic Programming*, 1986.

[Matsumoto 85]
H. Matsumoto, A Static Analysis of Prolog Programs, SIGPLAN Notices, vol. 20, no. 10, 1985.

[Mellish 82]
C. S. Mellish, An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter, in *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K. L. Clark and S.-Å. Tärnlund (eds.), Academic Press, London, pp. 99-106, 1982.

[Pereira 87]
F. Pereira , C-Prolog User's Manual Version 1.3, SRI International, 1987.

[Pereira&Porto 82]
L. M. Pereira and A. Porto, Selective Backtracking, in *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K. L. Clark and S.-Å. Tärnlund (eds.), Academic Press, London, pp. 107-114, 1982.

[Roberts 77]
G. M. Roberts, An Implementation of Prolog, M.Sc. Thesis, University of Waterloo, Canada, 1977.

[Robinson 65]
J. A. Robinson, A Machine-oriented Logic based on the Resolution Principle, *Journal of the ACM*, vol. 12, no. 1, pp. 23-41, 1965.

[Shapiro 83]
E. Y. Shapiro, Algorithmic Program Debugging, MIT Press, Cambridge, 1983.

[van Emden 82]
M. H. van Emden, An algorithm for Interpreting Prolog Programs, *Proceedings of the First International Logic Programming Conference*, The University of Marseille, 1982.

[Van Hentenryck 87]
P. Van Hentenryck, A Consistency Techniques in Logic Programming, Ph.D. Thesis, University of Namur(Belgium), 1987.

[Warren 77]
D. H. D. Warren, Implementing Prolog: Compiling Predicate Logic Programs, D.A.I. Research Report Nos. 39 and 40, University of Edinburgh, Scotland, 1977.

[Wolfram 86]
D. A. Wolfram, Intractable Unifiability Problems and Backtracking, *Third International Conference on Logic Programming*, 1986.

[You&Wang 88]
J.-H. You and Y. Wang, Context Resolution: A Computational Mechanism for Intelligent Backtracking, *Proc. of the 7th Biennial Conference of the CSCSI*, pp. 234-241, Edmonton, Alberta, 1988.

[You&Wong 89]
J.-H. You and B. Wong, Intelligent Backtracking Made Practical, to be submitted.

Appendix I

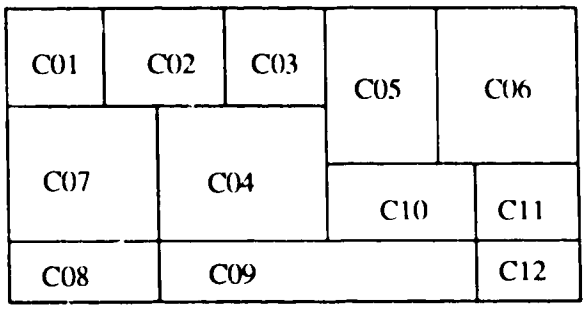
Program Listings

```
% ===========
% database query
% ===========

student(robert,prolog).
student(john,music);
student(john,prolog);
student(john,surf);
student(mary,science);
student(mary,art);
student(mary,physics);

professor(luis,prolog);
professor(luis,surf);
professor(maurice,prolog);
professor(eureka,music);
professor(eureka,art).
professor(eureka,science);
professor(eureka,physics);

course(prolog,monday,room1);
course(prolog,friday,room1);
course(surf,sunday,beach);
course(math,tuesday,room1);
course(math,friday,room2);
course(science,thursday,room1);
course(science,friday,room2);
course(art,tuesday,room1);
course(physics,thursday,room3);
course(physics,saturday,room2);

ask(Student,Course1,Course2,Prof) ←
      student(Student,Course1) &
      course(Course1,_,Room) &
      professor(Prof,Course1) &
      student(Student,Course2) &
      course(Course2,_,Room) &
      professor(Prof,Course2) &
      not eq(Course1,Course2);
```

```
? ask(Student,Course1,Course2,Prof);

% ===============
% naive 6-queen problem
% ===============

queens(L,Config) ←
      perm(L,P) &
      pair(L,P,Config) &
      safe([],Config);

perm([],[]);
perm([X|Y],[U|V]) ←
      delete(U,[X|Y],W) &
      perm(W,V);

delete(X,[X|Y],Y);
delete(U,[X|Y],[X|V]) ← delete(U,Y,V);

pair([],[],[]);
pair([X|Y],[U|V],[p(X,U)|W]) ← pair(Y,V,W);

safe(_,[]);
safe(Left,[Q|R]) ←
      test(Left,Q) &
      safe([Q|Left],R);

test([],_);
test([R|S],Q) ←
      test(S,Q) &
      notOnDiagonal(R,Q);

notOnDiagonal(p(C1,R1),p(C2,R2)) ←
      C is C1 - C2 &
      R is R1 - R2 &
      not eq(C,R) &
      NR is R2 - R1 &
      not eq(C,NR);

? queens([1,2,3,4,5,6],Config);
% use '? queens([1,2,3,4,5,6,7],Config);' for naive 7-queen problem

% ================
% clever 6-queen problem
% ================

queens(Config) ← solution(c(0,[]),Config);

solution(c(6,Config),Config) ← cut;
```

```
% replace the above by 'solution(c(7,Config),Config) ← cut;'
% for clever 7-queen problem
solution(c(M,Config),Conf) ←
      expand(c(M,Config),c(M1,Conf1)) &
      solution(c(M1,Conf1),Conf);

expand(c(M,Q),c(M1,[p(M1,K)|Q])) ←
      M1 is M + 1 &
      column(K) &
      noAttack(p(M1,K),Q);

column(1);
column(2);
column(3);
column(4);
column(5);
column(6);
% add 'column(7);' for clever 7-queen problem

noAttack(_,[]);
noAttack(P,[Q|L]) ←
      noAttack(P,L) &
      ok(P,Q);

ok(p(_,C),p(_,C)) ← cut & fail;
ok(p(R1,K1),p(R2,K2)) ←
      Difr is R2 - R1 &
      abs(Difr,Abs) &
      Difc is K2 - K1 &
      abs(Difc,Abs) &
      cut &
      fail;
ok(_,_);

abs(N,N) ← N > 0 & cut;
abs(N,M) ← M is 0 - N;

? queens(Config);

% ===============
% binary tree insertion
% ==============

tree([],Tree,Tree);
tree([E|Rest],Tree,NewTree) ←
      insert(E,Tree,Temp) &
      tree(Rest,Temp,NewTree);

insert(E,[],t([],E,[])) ← cut;
```

```
insert(E,t(L,N,R),t(NewL,N,R)) ←
      E < N &
      cut &
      insert(E,L,NewL);
insert(E,t(L,N,R),t(L,N,NewR)) ← insert(E,R,NewR);

? tree([46,11,48,46,47,6,5,9,7,5,14,17,14,22,1,32,61,14,56,11,78],[],Tree);
```




```
% ==============
% clever map coloring
% ==============

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);

good(C01,C02,C03,C04,C05,C06,C07,C08,C09,C10,C11,C12,C13) ←
      next(C01,C13) & next(C01,C02) & next(C02,C13) & next(C02,C04) &
      next(C04,C10) & next(C06,C10) & next(C08,C13) & next(C06,C13) &
      next(C02,C03) & next(C03,C04) & next(C03,C13) & next(C03,C05) &
      next(C05,C06) & next(C05,C13) & next(C04,C05) & next(C05,C10) &
      next(C01,C07) & next(C07,C13) & next(C02,C07) & next(C04,C07) &
      next(C07,C08) & next(C04,C09) & next(C09,C10) & next(C08,C09) &
      next(C09,C13) & next(C06,C11) & next(C10,C11) & next(C11,C13) &
      next(C09,C12) & next(C11,C12) & next(C12,C13);

? good(C01,C02,C03,C04,C05,C06,C07,C08,C09,C10,C11,C12,C13);
```

```
% ============
% bad map coloring
% ============

bad(C01,C02,C03,C04,C05,C06,C07,C08,C09,C10,C11,C12,C13) ←
      next(C01,C02) & next(C02,C03) & next(C03,C04) & next(C04,C05) &
      next(C05,C06) & next(C06,C11) & next(C11,C12) & next(C12,C13) &
      next(C09,C13) & next(C09,C10) & next(C04,C10) & next(C04,C07) &
      next(C07,C08) & next(C02,C07) & next(C06,C10) & next(C02,C13) &
      next(C06,C13) & next(C02,C04) & next(C08,C13) & next(C04,C09) &
      next(C03,C05) & next(C08,C09) & next(C01,C13) & next(C03,C13) &
      next(C05,C13) & next(C07,C13) & next(C11,C13) & next(C09,C12) &
      next(C05,C10) & next(C10,C11) & next(C01,C07);

? bad(C01,C02,C03,C04,C05,C06,C07,C08,C09,C10,C11,C12,C13);

% ===========
% move checking
% ===========

legal(X,Y,N) <-
      legal1(X,Y) &
      legal2(X,Y,N);

legal1(X,Y) <-
      Z is Y + 3 &
      X >= Z;

legal2(X,Y,1) <-
      Z is Y + 5 &
      X > Z;
legal2(X,Y,N) <-
      check(X,Y,1,K) &
      N is K + 1;

check(X,Y,N,N) <-
      X is Y + Y &
      cut;
check(X,Y,N,K) <-
      N < 10 &
      X1 is X + 1 &
      Y1 is Y + 1 &
      N1 is N + 1 &
      check(X1,Y1,N1,K);

move( 3,1, 7); move( 3,3,9); move(15,3, 5); move(11,5, 4); move(1,3, 9);
move(20,7, 8); move(30,9,6); move(12,6, 2); move( 5,4,10); move(8,2, 3);
move( 2,8,18); move(12,4,4); move( 5,7,15); move( 8,4,10); move(3,9,19);
```

```
ordering1(A,B,C,X,Y) <-
        move(A,B,C) &
        legal(A,B,X) &
        legal(C,B,Y) &
        X > Y;

ordering2(A,B,C,X,Y) <-
        move(A,B,C) &
        legal(C,B,Y) &
        legal(A,B,X) &
        X > Y;

? ordering1(A,B,C,X,Y);
? ordering2(A,B,C,X,Y);
```

Appendix II

Local B-lists

In this section, we will look at the implementation of the local *B*-lists on context resolution and present some results based on this implementation. Since this work is done after the completion of the thesis, we include a brief summary in this appendix. Further details can be found in [You&Wong 89].

Motivation

As shown in Chapter six, the speed up for the simple map coloring problem is -87.0% in context resolution, compared to -99.9% in many other schemes. These figures correspond to a reduction of the speed up from 1000 times ($\frac{1}{1-99.9\%}$) to 7.7 times ($\frac{1}{1-87.0\%}$), a difference by a factor of approximately 130 times ($\frac{1000}{7.7}$)! One reason for this discrepancy is due to the lookahead mechanism in WUP. In the previous implementation, whenever there is a unification failure in the lookahead clause and a context is returned, the context is put in the global *B*-list and the next clause is checked. This context can make the execution less efficient. When a context *#c* from a lookahead clause is larger than the contexts returned from a derivation failure for a subgoal *P*, *#c* will be chosen during a deep backtracking. However, *P* will fail again since *#c* is more recent than any contexts which are associated with the failed terms in *P* which caused backtracking. By turning off the lookahead, the speed up becomes -91.5% (11.8 times). But there is still a difference by a factor of 85 times. Now we turn our attention to the issue of the local *B*-list, which is the other main reason for this large discrepancy.

Analysis

In the previous implementation, the contexts which are returned during shallow backtrackings are kept in the global *B*-list, and they will cause the same problem as mentioned above. When a local *B*-list is used, the contexts are local to a

subgoal and every context considered during a deep backtracking is responsible for the failure of the subgoal which caused backtracking. We expect that there will be a general reduction in the number of inferences, unifications and backtrackings. However, the run time depends on the overhead of our implementation, which will be described next.

Implementation

Instead of associating a local *B*-list with every subgoal, we introduce one more field in the context array. This field tags the subgoal which introduced the context in the context array. Thus for each context, the corresponding subgoal can be identified. The following algorithm summarizes the operations for the insertion and retrieval of contexts. We assume that the context field in the context array is represented by CA[i].cnt while the tag field is accessed by CA[i].tag.

Insertion:

Whenever a term in the subgoal *P* fails to unify with another term in the head of a clause, the interpreter performs the following steps on both terms:

```
for each context #c in the list of contexts associated with the term do:
    i ← hash(#c);
    CA[i].cnt ← #c;
    CA[i].tag ← address of P;
```

Retrieval:

```
Whenever a subgoal P fails in a derivation:
    i ← highest index in CA where there is a context;
start: if (CA[i].tag = address of P) {
        backtrack_point ← CA[i].cnt;
        CA[i].cnt ← ⊥;
        CA[i].tag ← ⊥;
    }
    else {
        i ← i - 1;
        if (i = 0)
            backtrack_point ← MB;
        else
            goto start;
    }
```

The retrieval operation requires some explanations. First we get the largest index (*i*) in the array. Then we search the array from *i* sequentially downward until we find a tag which matches the failed subgoal or we reach the bottom of the array

(index 0). When we have a match, the context field of that element is our backtrack point (recall that the elements in the array are sorted and the first element encountered is the largest). Otherwise we backtrack to the point indicated by MB.

Results and Conclusion

The results are shown in Tables A.1 and A.2. Besides having achieved a speed up of -99.9% on the simple map coloring problem, the modified implementation gives an overall better performance on all the other programs (when compared to the global *B*-list implementation). This is mainly due to the increased degree of intelligence of the local *B*-list approach and the low overhead for the implementation. From these results, we can again clearly establish the claim that intelligent backtracking based on context resolution can be considered as a standard implementation technique for stack-based sequential Prolog.

Table A.1: Comparison of Execution Statistics

Problem	Naive WUP	Intelligent WUP	% Change
database query			
# inferences	83	27	-67.5%
# unifications	152	38	-75.0%
# backtrackings	44	5	-88.6%
time in ms (#)	10416(100)	1400(100)	-86.6%
6-queen simple			
# inferences	7651	2402	-68.6%
# unifications	9959	2628	-73.6%
# backtrackings	1430	55	-96.2%
time in ms (#)	6350(1)	1283(1)	-79.8%
6-queen clever			
# inferences	3831	3831	0.0%
# unifications	5784	5784	0.0%
# backtrackings	806	806	0.0%
time in ms (#)	2050(1)	2250(1)	+9.8%
7-queen simple			
# inferences	6296	803	-87.2%
# unifications	8425	861	-89.8%
# backtrackings	1340	11	-99.2%
time in ms (#)	4950(1)	400(1)	-91.9%
7-queen clever			
# inferences	1044	1044	0.0%
# unifications	1578	1578	0.0%
# backtrackings	210	210	0.0%
time in ms (#)	516(1)	550(1)	+6.6%
tree insertion			
# inferences	217	217	0.0%
# unifications	308	308	0.0%
# backtrackings	30	30	0.0%
time in ms (#)	1266(10)	1350(10)	+6.6%
map color clever			
# inferences	44	44	0.0%
# unifications	92	87	-5.4%
# backtrackings	12	9	-25.0%
time in ms (#)	2966(100)	3083(100)	+3.9%

Table A.1: Continue

Problem	Naive WUP	Intelligent WUP	% Change
map color simple			
# inferences	89250	133	-99.9%
# unifications	270644	236	-99.9%
# backtrackings	57897	10	-99.9%
time in ms (#)	80066(1)	750(10)	-99.9%
move ordering 1			
# inferences	409	154	-62.3%
# unifications	464	168	-63.8%
# backtrackings	55	14	-74.5%
time in ms (#)	2150(10)	650(10)	-69.8%
move ordering 2			
# inferences	295	145	-50.8%
# unifications	331	159	-52.0%
# backtrackings	36	14	-61.1%
time in ms (#)	1466(10)	633(10)	-58.0%

Table A.2: Comparison With Related Research

Problem	DTA (CPU)	GCA (# cycles)	GCA (CPU)	SDDA (# instrs)	DIB (CPU)	CR (CPU)
database query	-19.4%	-48.9%	-55.4%	-16.0%	-52.4%	-86.6%
6-queen simple	-34.9%	+162.8%	+16.6%	N/A	-61.5%	-79.8%
6-queen clever	+99.3%	+90.6%	+8.6%	N/A	+10.0%	+9.8%
7-queen simple	-82.7%	N/A	N/A	N/A	-89.2%	-91.9%
7-queen clever	+103.9%	N/A	N/A	N/A	+20.0%	+6.6%
tree insertion	+43.8%	+9.0%	0.0%	N/A	+30.0%	+6.6%
mapcolor clever	+63.5%	+7.8%	-3.9%	+0.7%	N/A	+3.9%
mapcolor simple	-99.7%	-99.9%	-99.9%	-99.9%	N/A	-99.9%
move ordering 1	N/A	-54.1%	N/A	N/A	N/A	-69.8%
move ordering 2	N/A	-33.9%	N/A	N/A	N/A	-58.0%

Legends:

DTA	Deduction Tree Analysis	GCA	Generator-Consumer Analysis
SDDA	Static Data Dependency Analysis	CR	Context Res. (local *B*-list)
DIB	Depth-First Intelligent Backtracking		