A Framework for Efficient Execution of Logic Programs.

Sudharsan R. Iyengar
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/4783

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A framework for efficient execution of logic programs

Iyengar, Sudharsan R., Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1989
A Framework for Efficient Execution of Logic Programs

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Sudharsan R. Iyengar

B.E., Electrical Engineering, Regional Engineering College, Durgapur, India, 1984
M.S., Systems Science, Louisiana State University, Baton Rouge, 1987
M.S., Electrical Engineering, Louisiana State University, Baton Rouge, 1987

August, 1989
Acknowledgements

I am indebted to my major professors S. Sitarama Iyengar and Dr. Andrzej Hoppe for their encouragement, support and guidance. More importantly, I am grateful for their enormous amount of time spent in discussing and writing this dissertation. I thank my committee members Prof. Donald H. Kraft, Dr. Doris Carver and Prof. El-Amawy.

I acknowledge the assistance of Dr. Sukhamay Kundu in this project.

This work would not have been fruitful but for the relentless encouragement and motivation provided by my dear uncle, Prof. A. Raman, and his family. My parents and my sisters deserve special credit for the continuous love and affection, poured to me through the distance from India.

Memories will always remain of my friends Lalitha Krishnamurthy, Krishna Prasad, Mohan B. Sharma, Ananta Prasad, Guja (Rajanarayan), and scores of others who all provided me with a nourishing environment throughout my Ph.D. work.

I also acknowledge the Department of Computer Science, in particular Dr. Kraft, for the financial support.
List of Tables

<table>
<thead>
<tr>
<th>Table I</th>
<th>Trace of Naive Prolog Execution</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table II</td>
<td>Comparison of Results in Literature</td>
<td>60</td>
</tr>
<tr>
<td>Table III</td>
<td>Trace of Execution by Intelligent Backtracking</td>
<td>101</td>
</tr>
<tr>
<td>Table IV</td>
<td>Trace of Execution by Intelligent Forward Execution</td>
<td>115</td>
</tr>
<tr>
<td>Table V</td>
<td>Trace of Execution by Forward Jumping and Intelligent Backtracking</td>
<td>133</td>
</tr>
<tr>
<td>Table VI</td>
<td>Trace of Execution by Intelligent Backtracking, Intelligent Forward Execution and Forward Jumping</td>
<td>155</td>
</tr>
<tr>
<td>symbol</td>
<td>meaning</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>∨</td>
<td>or</td>
<td></td>
</tr>
<tr>
<td>¬</td>
<td>not</td>
<td></td>
</tr>
<tr>
<td>←</td>
<td>if</td>
<td></td>
</tr>
<tr>
<td>∀</td>
<td>for all</td>
<td></td>
</tr>
<tr>
<td>∃</td>
<td>there exists</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>such that</td>
</tr>
<tr>
<td>iff</td>
<td>if and only if</td>
<td></td>
</tr>
<tr>
<td>□</td>
<td>success</td>
<td></td>
</tr>
<tr>
<td>■</td>
<td>failure</td>
<td></td>
</tr>
<tr>
<td>Δ</td>
<td>solution of a goal</td>
<td></td>
</tr>
<tr>
<td>∈</td>
<td>belongs to</td>
<td></td>
</tr>
<tr>
<td>∪</td>
<td>set union</td>
<td></td>
</tr>
<tr>
<td>∩</td>
<td>set intersection</td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td>empty set</td>
<td></td>
</tr>
<tr>
<td>θ, σ</td>
<td>state of computation</td>
<td></td>
</tr>
<tr>
<td>Λ</td>
<td>goal tree</td>
<td></td>
</tr>
<tr>
<td>ζ</td>
<td>search method</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

Figure 1. Standard Goal-Search-Tree by Prolog ................................................................. 31
Figure 2. Execution of Prolog on a linear stack ................................................................. 36
Figure 3.1. Recursive Execution Model, of Pereira and Porto ............................................. 41
Figure 3.2. Dependency Tag representation, of Pereira and Porto ...................................... 41
Figure 4.1. Minimal Deduction Subtree that is consistent, of Brunooghee ......................... 43
Figure 4.2. Maximal Deduction Subtree that is consistent, of Cox .................................... 43
Figure 5.1. Type I backtracking based on SDDA, of Chang and Despain ......................... 46
Figure 5.2. Type II backtracking based on SDDA, of Chang and Despain ....................... 47
Figure 5.3. Type III backtracking based on SDDA, of Chang and Despain ....................... 47
Figure 6. Backtracking based on B-list, of Conery and Kibler ......................................... 50
Figure 7. Backtracking based on dynamic B-list, of Vipin Kumar, et al. ......................... 52
Figure 8. Variable Based Backtracking, of Rajasekhar, et al. ........................................... 54
Figure 9. Local-stack used by WAM, of Warren ................................................................. 58
Figure 10. Identification of Failure-Bindings when a predicate is defined by facts ............ 69
Figure 11. Identification of Failure-Bindings when a predicate has facts and rules ............. 71
Figure 12.1. Entries to FBT ................................................................................................. 75
Figure 12.2. Maximal tuples in FBT .................................................................................... 77
Figure 13. Unification with facts and Bit-vector manipulation ........................................... 92
Figure 14. Bit-vector manipulation with application of rules .............................................. 93
Figure 15. Identification of Failure-binding in the presence of facts and rules .................. 94
Figure 16. Incompleteness of identification of Failure-bindings by rules ......................... 97
Figure 17. Incompleteness of identification of Failure-bindings by unification ................... 98
Figure 18. Goal-search tree explored by Intelligent Backtracking based on failure-bindings 100
Figure 19. Goal-search tree explored by Intelligent Forward Execution ......................... 114
Figure 20.1. Possible values of redo(2, 9) for example in Figure 1. ................................. 125
Figure 20.2. Possible values of redo(2, 11) for example in Figure 1. ................................. 126
Figure 21. Goal-search tree explored by Intelligent Backtracking and Forward Jumping. 132
Figure 22. Execution of logic programs on Segmented Stack 146
Figure 23. Execution Model Implementing Intelligent Backtracking, Intelligent Forward Execution and Forward Jumping. 153
Figure 24. Goal-search tree by Intelligent Backtracking, Intelligent Forward Execution and Forward Jumping 154
# Table of Contents

Acknowledgements ................................................................................................................. ii

List of Tables ............................................................................................................................ iii

List of Symbols .......................................................................................................................... iv

List of Figures ............................................................................................................................ v

Abstract ........................................................................................................................................ x

1. Introduction ........................................................................................................................... 1

   1.1. Overview ......................................................................................................................... 1

   1.2. Logic Programming as a Tool for Artificial Intelligence .............................................. 3

   1.3. Prolog as a Logic Programming Language ...................................................................... 6

   1.4. Disadvantages of Exhaustive Search by Prolog ................................................................. 10

   1.5. Motivation for Efficient Execution ................................................................................... 16

   1.6. Contribution of the Dissertation ....................................................................................... 15

   1.7 Outline of the Dissertation ............................................................................................... 17

2. Introduction to Logic Programming ....................................................................................... 19

   2.1. Logic Programs ............................................................................................................... 19

   2.2. Reasoning by Resolution ................................................................................................. 22

   2.3. Unification ....................................................................................................................... 26

   2.4. Procedural Interpretation ................................................................................................. 28

   2.5. Interpreter ....................................................................................................................... 33

   2.6. Implementation Issues ..................................................................................................... 37

3. Literature Review .................................................................................................................. 38

   3.1. Introduction ..................................................................................................................... 38

   3.2. Selective Backtracking Based on Unification Mapping Information ............................. 39

   3.3. Run-time Intelligent Backtracking .................................................................................... 42

   3.4. Semi-Intelligent Backtracking Based on Static Data Dependency Analysis.................... 44

   3.5. Generator-Consumer Approach ....................................................................................... 48
Abstract

The focus of this dissertation is to develop an efficient framework for sequential execution of logic programs. Within this framework the logic programs are executed by pruning the goal-search tree whenever applicable.

Three new concepts for pruning of computation during execution of logic programs are introduced.

Failure-binding

A Failure-binding for a literal is a binding which when applied to the literal fails the goal obtained from the literal. Failure-bindings for a literal are identified by analyzing the goal-tree of a goal which is obtained from the literal. The failure-bindings for a literal are used for intelligent backtracking based on the generator-consumer approach. Intelligent backtracking based on failure-bindings prune the computation of search space which lead to late detection of failure.

Failure-solution

A Failure-solution of a goal is unacceptable to some other subgoal in the forward execution. Failure-solutions of a goal are identified by analyzing the history of computation, during execution. Failure-solutions of the goals are used for intelligent forward execution. Intelligent forward execution prunes the computation of search space which leads to repeated failure resulting from repeated successes of a goal.

Forward Jumping

Forward jumping is a method to avoid reexecution of some subgoals after backtracking (instead of naive forward execution after backtracking). Forward jumping is based on the dynamic subgoal dependencies in a rule. Such jumping prunes the computation of the search spaces which leads to the same sequences of successes of subgoals after backtracking.
To facilitate the implementation of these concepts a new data structure, called segmented-stack, is defined. The space complexity of a segmented stack is linear in the number of nodes in the stack. Depth-first search as well as breadth-first search are very easily implemented on a segmented-stack during execution of logic programs. Execution of logic programs on a segmented-stack allows association of the search space, as well as the solutions, of a goal with the frame of the goal. This enables implementation of intelligent backtracking, intelligent forward execution and forward jumping.

The search based on each of these paradigms is proved to be sound and complete. It is also shown that the implementation of these paradigms preserves the order of results obtained by Prolog. The effects of the non-logical operators, in Prolog, on the paradigms are studied. The search based on these paradigms is compared individually, and collectively, with the standard search by Prolog.

**Keywords:** logic programming, failure-binding, intelligent backtracking, failure-solution, intelligent forward execution, forward jumping, segmented stack.
Chapter 1

Introduction

1.1. Overview

Artificial Intelligence (AI), or computer intelligence, combines the automated computing power of computers to implement effective intelligent techniques. Research in AI has focused on the finding newer paradigms for automating intelligent behavior. Recent attempts to develop comprehensive and adaptable intelligent systems which obtain specific solutions to specific problems have proven beneficial in areas such as medical diagnosis, legal affairs, weather forecasting, testing and safeguarding of nuclear equipments, intelligent database and knowledge based systems, robotics [55, 58], language processing, effective and relevant information retrieval, and image processing.

Research in AI is being directed towards the development of systems which not only obtain solutions for a given problem [5] but also refines their own process dynamically so as to perform more accurately and efficiently. Such techniques use some kind of a learning process, constraint-based efficient search methods, information refinement and enhancement techniques, and self correcting principles.

Over the past two decades, there has been a growing interest in the design and development of AI languages useful in building intelligent systems which are not only efficient and easy to program, but which provide a framework for implementing learning. Fifth Generation projects, by ICOT in Japan [42], concentrate on developing languages and efficient frameworks for programming and executing knowledge based decision support systems. Automated deduction is central to the aim of achieving artificial intelligence.
The AI requirements of a language include provisions for

1) automatic deduction,
2) early and effective constraint realization and propagation,
3) ability to intermix procedures and data,
4) goal directed behavior,
5) knowledge structuring and learning,
6) metalevel programming and control structures, [57].

Three types of languages for programming and developing such intelligent systems are under consideration.

1. Functional languages, like Lisp, are used in developing expert and production systems and for implementing constraint based computational models.

2. Relational or Logic languages, like Prolog [4], are used in developing advanced database and knowledge-base engineering and decision support systems. Logic languages are basically rule based languages.

3. Object oriented languages, like Smalltalk, are used for simulation, testing and performance analysis of real problems.

Of these three types of languages, logic programming languages are most popular, mainly because of their expressive capacity and simplicity.

The main focus of this dissertation is to obtain a general framework that provides efficient execution process of logic programs. In spirit, we attempt to present new computational paradigms for efficient execution of logic programs using new characterizations and a new data structure.
1.2. Logic Programming as a Tool for Artificial Intelligence

Almost all AI problems are solved based on different types of constraints which need to be globally satisfied. This requires extensive search to effect constraint generation, propagation and satisfiability. In logic programs the constraints are values for variables - see section 2.2. Logic programming enables specification of constraint propagation its execution performs the required search on these constraints.

Logic programming began in the early 1970’s as a direct outgrowth of earlier work in automatic theorem proving and predicate logic. Earlier work by Herbrand, Prawitz, Gilmore, Davis, Putnam [4] and others culminated in 1965 with the publication of the landmark paper by Robinson [1 - 6] which introduced the resolution principle - an inference rule well suited to automation on a computer.

Kowalski [7 - 10] showed that an algorithm consists of two disjoint components - the logic and the control - the what and how part of a problem to be solved, respectively. Logic is used to express the what part. Thus, an algorithm is converted to symbolic manipulation.

Logic is an effective symbolic framework for theorem proving. Theorem proving is an automated deduction procedure, a procedure to derive newer information from a given information base. Resolution theorem proving is an effective tool for developing automated inference engine on a symbolic framework.

A symbolic system is developed to express knowledge in the form of propositions, expressions which have only a True or False value, and the relationships between these propositions. The statements, or rules, about these propositions also express how one can validly infer newer propositions from those that are given. More complicated propositions can be expressed in terms of simple propositions by the use of the connectives (e.g. ∧, ∨, ¬, ⊃) between them.
The programmer needs to specify only the logic component of the problem in the form of propositions and rules about the propositions. The control during execution, which is independent of the problem description, provides an answer to the query requested by using standard deduction procedures on the propositions and rules specified by the programmer.

Propositional logic is a sound framework for symbolic computation. The arguments for propositional logic can be extended to predicate logic, where propositions are expressed in terms of predicates, a predicate name followed by n-tuple arguments. Predicate logic is a more general framework, because it allows more generalized expressions in terms of variables and ground terms. Variables can assume different values at different times. The programmer can then specify general information about a predicate using variables. First order predicate logic (FOPL), a restricted form of predicate logic, see section 2.1, provides methods for deducing newer theorems of a theory as logical consequences of the axioms and the theorem which are provided. A logic program consists of a set of procedures, or axioms, and a goal statement.

Again the programmer needs to specify only the logic component of the problem which is information about predicates. The execution, or the control part, provides answer to the goal by using standard deduction procedures.

Predicate logic has two aspects: the syntactic and the semantic aspects. The syntactic aspect is concerned with syntactic constructs of the symbolic system. The semantics is concerned with the meanings attached to the statements in the syntactic constructs.

A clause is a disjunction of literals, which are propositions in the form of predicates. Ground unit clauses are facts, i.e. a single literal with a fixed true or false value, and rules are generalized clauses. A logic program with only ground unit clauses, i.e. facts, is a relational database.
Theorem proving is an automated procedure to derive new information from a given information base. Resolution theorem proving is an effective tool for developing an automated deduction system. All logic programming systems are based on resolution theorem proving.

The problems in artificial intelligence can be converted into symbolic manipulation by proper knowledge representation, manipulation and inferencing problems. Logic programs together with automatic deduction rules are based on the resolution principle and form a powerful framework for developing AI systems. Most logic programming systems are resolution theorem provers.

The credit for the introduction of logic programming goes mainly to Kowalski and Colmerauer [7-10] who introduced in 1972 the fundamental idea that logic can be used as a programming language. A programming language for implementing logic programs is Prolog. The next section gives a brief introduction to Prolog.
1.3. Prolog as a Logic Programming Language

Prolog (PROgramming in LOGic) is a programming language to describe logic programs. The first Prolog interpreter was implemented in Algol by Roussel in Marseille in 1972 [4].

Prolog is based on Robinson's Resolution Principle with specific top-down left-to-right control. The programmer describes his problem in a symbolic framework which he develops. Users can now test a goal on the logic program to obtain newer information. The goal is also a symbolic expression. The goal is mapped onto the program and the theorem proving technique is applied to see if the goal is satisfied by the theorems provided in the program in the form of program clauses - facts and rules.

Programming in Prolog, with its well defined syntax, facilitates generalized specification of information and their inter-relationships. It essentially implements a theorem prover for the programmer. The term "logic programs" has now become analogous to Prolog programs.

Prolog systems employ the computation rule which always selects the leftmost goal in a sequence of goals called the goal-list for execution. It also employs Linear Resolution with a Selector Function, a selection mechanism which uses the order of the program clauses, to select clauses during execution. The execution mechanism is independent of the program specification. Prolog employs deterministic top-down resolution which results in depth-first search to satisfy a goal.

Success is reported if the goal unifies (see section 2.3) with a clause and the clause is true. The goal-list is modified during the execution of a logic program. The search rule is implemented on a linear stack of goals, called the local-stack. An instance of the goal stack represents the branch of the goal-search tree that is currently being investigated. The computation then essentially becomes an interleaved sequence of pushes and pops on this stack [see section 2.5].
The resulting search is a depth-first search of the goal-tree. Depth first rules can be very efficiently implemented on a stack. For a system that searches depth-first, the search rule reduces to an ordering rule which specifies the order in which the program clauses are to be tried. Prolog systems use the order of clauses in a program as the fixed order in which they are to be tried. This is implemented in a very simple and efficient way.

The positive features of Prolog can be enumerated as follows.

- As any other programming language, it has well defined syntax.
- It is closer to a declarative language than are logic programs (see section 2.1). The procedural interpretation of Prolog helps the programmer to follow Prolog programs more clearly.
- The answers to the query are accurate within the domain of the program leading to answers within the closed world (CW) of the program.
- \texttt{Not} is implemented as a negation by failure - that is the negation of a goal is true if the goal cannot be satisfied by the program. This is sufficient within the CW of the program.
- The selection rule is fixed; hence, the behavior of the program is explicit to the programmer.
- The input/output indirection as a result of the unification substitution mechanism (see section 2.3) is a powerful characteristic. This removes the necessity of mode specification in the programs. Thus programs can be used to test for satisfiability of goals or to generate solutions for which the query is true,
- Prolog interpreters are usually interactive in nature and hence encourage man-machine interaction.
- Failure of a goal implies that the goal cannot be satisfied with the given con-
straints applied to the goal. Consequently, the detection of a success or failure of a goal is also sound.

- Backtracking is used to get also another set of constraints for a failed goal. The actions taken by the interpreter to effect backtracking are elementary in nature.

- Alternatives are selected top to bottom and failure is reported if and only if there are no more alternatives for a goal. Thus, all possible paths to satisfying a goal are explored. Consequently, every solution to a goal is reported. Since the search is exhaustive, it is complete within the CW of the program, as long as the search does not get trapped in an infinite loop.

- Many non-logical operators like +, -, is, nl, write, cut, assert and retract, are provided to make Prolog useful for different applications.

- Identical solutions for the predicates may be obtained via different search paths in the goal-search tree. There is no restriction on the type or size of the terms used and the number of solutions for a goal.†

- The control operates on a goal-list and a linear stack of activation records. The implementation of this control is very simple and efficient - translating itself into operations similar to the implementation of a Push Down Automata.

Since Prolog implements depth first search with fixed ordering, the resulting search is naive and exhaustive. Each call to a definition tries the clauses in the definition in exactly the same order every time a program is executed. Even though it employs goal-oriented search, it does not effect selective and relevant search.

† It is precisely due to this generality that the search process gets trapped into an infinite loop.
For an infinite search tree, depth-first search is not fair. Breadth-first search would be fair (avoid infinite search tree), but is less compatible with efficient implementations. The depth-first search method results in incompleteness of Prolog resolution, because it does not guarantee finding the success branch - even if one exists - because the search may get trapped in an infinite branch. Hence, Prolog systems, though sound, are partially incomplete.

Exhaustive search is not always necessary, and the search process can be pruned without losing the soundness and completeness properties of Prolog. The disadvantages of the present implementation of Prolog is presented in the next section.
1.4. Disadvantages of Exhaustive Search by Prolog

Query processing should base its decisions on the entire search, i.e., the search space with all the permutations of the constraints. It is also true that only a select set of permutations of constraints satisfy the query. All sets of permutations are tried by employing backtracking when a failure is encountered during the execution of logic programs. The set of constraints and the resulting search space that is explored is rather large.

The use of traditional backtracking to explore a search space in a top-down, depth-first scheme starts with the initial state with the query goal as the current state. For each forward derivation step, one of the goals yet untried in the current state is used to derive a new current state. Success is reported when the set of untried goals applicable in the current state is empty. Failure is reported when an untried goal in the current state cannot be solved. Forward execution is repeated until a success state, $\Box$, is reached, or a failed state, ■, is reached. In the $\Box$ state, the goal-list is empty and the local-stack contains the path to the successful state. In the ■ state, the local-stack is empty and the goal-state contains the query goal.

After reaching ■, the system returns to a previous state. This process is called backtracking. Forward execution is repeated for this state to derive newer states. If backtracking returns to the state prior to ■, then this is naive backtracking. A failure is cured only when the cause of the failed state is undone. Retracting to the previous state on reaching ■ may not be able to cure the failure. Unfortunately, there is no way to identify the cause of failure. The result of naive backtracking could be unusual trashing, unnecessary backtracking and forward execution.

Due to the depth-first search a subgoal may be tried more than once during different stages of computation.

Consider a state, $f$, in which failure is reported and also assume that backtracking
takes us to state b.

1. Backtracking to b, after the failure at f, and subsequent return to f, without any change to the constraints applied in f, (this could be due to backtracking to b, and subsequent return, without changing the constraints applied in f, or due to applying the same set of constraints to the goal that failed in f).

2. Backtracking to b, after the failure in f, and subsequent return to f without curing the cause of failure of the subgoal in f, (this could be due to backtracking to b and solving a subgoal in b whose new solution does not cure the failure of the subgoal that failed in f).

3. Multiple appearance of the same subgoal in different states during the search due to the appearance of the same subgoal in different rules.

In particular, repeated computation of a subgoal may occur both for the case when a subgoal succeeds (generating the same solutions) and for the case when a subgoal fails (generating the same failure).

The specific deficiencies which result due to the exhaustive depth-first search implemented by Prolog can be enumerated as:

- Naive backtracking (backtracking without any basis) to the immediately previous state results in an exhaustive search. This could result in inconsequential exploration of some of the search space.

- There are dependencies among the subgoals in the rule which imply semantic relationships among the subgoals. Naive backtracking does not use these semantic relationships among the subgoals to identify a better state for backtracking. Again, this could result in inconsequential exploration of some of the search space.
The causes of failure of a subgoal in a state is not analyzed. This analysis could lead to identification of specific causes of failure which could be used for relevant exploration of search-space.

When a goal fails in a state \( f \), the dependency directed backtracking \([5]\) assumes the cause of failure of a goal as the set of all constraints applied to the subgoal that failed in \( f \). The backtracking is based on the dependencies among the subgoals of the rule and the system selects the latest state that generated any constraint for backtracking to ensure completeness of search. The selection of the state for backtracking is not based on the causes of failure of the subgoal in \( f \). The resulting search by dependency-directed backtracking may not cure the failure of the subgoal in \( f \). The cause of failure will eventually be corrected. This amounts to late detection, and correction, of the cause of failure. This results in futile exploration of some of the search space.

The history of computation is not used during different stages of computation. Repeated failures could occur due to the result of repeated identical solutions, constraints, generated by certain other goals. This results in repeatedly exploring in vain search spaces which contain no solutions. This is a direct consequence of not maintaining the information from the history of computation. Such dynamic learning is not employed by Prolog.

Computations continue blindly after backtracking. Some of the computations that are performed after backtracking may be independent of the corrective action taken by backtracking. Consequently, identical search spaces which lead to success, which are independent of the reason of backtracking, may be repeated. These computations need not be repeated. Prolog does not employ any technique to avoid the repetition of such computa-
Search spaces of different subgoals in a rule are not maintained separately. Prolog execution on a linear stack demands that computations be performed in a systematic manner where even independent computations are stacked on top of one another. This allows accessing different records in the chronological order of execution. This rules out independent access to the search space of different goals in a rule. Importantly, the information about the solutions of a goal cannot be associated with the goal.

The area of efficient execution of logic programs have been addressed by many researchers. The next section presents the motivation for intelligent execution of logic programs.
1.5. Motivation for Intelligent Execution

Efficient backtracking methods [1-7], which are essentially variants of dependency directed backtracking [9], have been proposed for eliminating certain inconsequential computations. An intelligent search mechanism employs a search rule which is directed by the information provided to it by the programmer. If no information is provided to determine the search rule (the order of search), then certain information derived during execution is used to effect selective and relevant search space exploration. These intelligent search methods use one or more of the following:

- unification mapping information [11] (see section 3.2)
- dynamic data-dependency [12, 13, 14, 15, 16] (see section 3.2)
- static data-dependency [17, 18] (see section 3.4)
- variable binding informations [27] (see section 3.7)
- generator-consumer lists of variables [19 - 26] (see sections 3.5 and 3.6)
- selective page retainment in the main memory [28, 35, 40]
- elimination of tail-recursion [29 - 32] (see section 3.8)

The major deficiencies of the exhaustive search employed Prolog, presented in section 1.3, have not been addressed by any of the existing solutions, for sequential Prolog, in the literature. The information content of the program is static in that there is no learning involved from the computations that are performed during execution. An intelligent search rule should utilize the semantics prescribed by the rules in the logic program together with a learning process and effective search techniques. The focus of this dissertation is to develop and present paradigms to eliminate all inconsequential search space exploration during execution.
1.6. Contributions of the Dissertation

We present newly developed paradigms applicable to logic programs. A framework base for efficient execution of logic programs is developed based on these paradigms and a new data structure for implementing these paradigms. The major contributions being made by this dissertation to the field of logic programs are as follows.

i) An exact analysis of the failure of a goal in logic programs is done to identify definite causes of failure of a goal. This gives global informations about predicates which is associated with the predicates for future use, thus effecting a learning process.

ii) A systematic evaluation method for identifying the causes of failure of the current goal, in Prolog, is presented. This analysis does not demand excessive dynamic time and space. An algorithm to identify the causes of failure is also presented.

iii) An intelligent backtracking scheme based on the identified exact causes of failure of a goal is proposed. The resulting search resulting from intelligent backtracking, based on exact causes of failure, is proved to be sound and complete.

iv) An intelligent forward execution algorithm to identify and eliminate repeated computations which result in failure during forward execution is presented. The intelligent forward execution is based on the notion of a failure-solution of a goal. The failure solutions of a goal are identified from the history of the execution. The resulting search based on intelligent forward execution based on failure solutions of a goal is proved to be complete and sound.

v) Forward jumping algorithm to eliminate repetition of computation during forward execution after backtracking is presented. Forward jumping is
based on subgoal dependencies in a rule. The forward jumping algorithm, which further prunes redundant computations so as to result in successes, is proved to be complete and sound.

vi) A new data structure, a segmented stack, for implementing the above algorithms is defined. Execution of logic programs on a segmented stack facilitates associating search spaces and solutions with a goal.

vii) Execution of logic programs based on the paradigms enumerated above is shown to preserve the order of results as obtained by Prolog.

viii) The search space explored by the use of the paradigms enumerated above is compared to the search space explored by standard Prolog.

ix) A framework which employs all the proposed paradigms is detailed.
1.7. Outline of the Dissertation

Chapter 2 presents a formal introduction to logic programs describing in detail the resolution principle and the implementation of Prolog as a programming language. The important implementation issues of a Prolog interpreter are analyzed in detail.

Chapter 3 reviews the literature pertaining to efficient execution of logic programs and presents the significant contribution to intelligent sequential execution of logic programs. Some of the contributions of parallel execution models which are applicable to sequential execution are exemplified.

Chapter 4 introduces the concept of a failure-binding of a goal after presenting a discourse on the causes of failure of a goal. A modified unification algorithm which enables the identification of failure-bindings is also presented. The selection of the backtrack literal, backtracking and subsequent forward execution is illustrated. The resulting search space explored is related to the search space explored by Prolog. It is proved that the identification of failure-binding is sound. The causes of incompleteness of the identification of failure-binding is scrutinized. A method to use the identified failure-bindings as global information about a predicate is described. This information is associated with the predicates and are used during subsequent search. The intelligent backtracking method based on failure-bindings is proved to be correct. The forward and backward execution algorithms based on failure-bindings are presented, and proved to preserve the soundness and completeness property of Prolog.

Chapter 5 introduces the notion of a failure-solution of a goal. A method to use the dynamically identified failure solutions from the history of computation for intelligent forward execution is proposed. The intelligent forward execution algorithm is presented. The correctness of this algorithm and the soundness and completeness properties of the resulting search are proved.

The concept of forward-jumping during sequential execution of logic programs
is introduced in chapter 6. An algorithm to obtain a redo-list (see section 6.3), which contains the subgoals which are independent of the corrective action taken by the backtracking process, is presented. Different instances of forward-jumping are illustrated. Forward jumping for recursive backtracking is also detailed. The resulting search is proved to be complete and sound. The search space explored by forward jumping is compared to the standard search space explored by Prolog.

Chapter 7 introduces the new data structure, a segmented stack, for implementing the paradigms and algorithms developed in chapters 3, 4 and 5. The segmented stack, together with its operations, is defined. The use of a segmented stack as a local stack for executing logic programs is illustrated.

Chapter 8 considers the effects of non-logical operators on the paradigms introduced in chapters 3, 4, and 5.

Chapter 9 presents a brief summary of the contributions of the dissertation. The flow-diagram of the framework which implements the concepts developed in this dissertation is given. A note for future research concludes the chapter.
2. Introduction to Logic Programming

2.1. Logic Programs

Logic programming differs fundamentally from other conventional programming in that it requires the programmer to describe the logical structure of a problem. The primitive symbol in logic programs is a proposition (see section 1.2). A proposition can have only one of the two values - True or False. Other primitive symbols are the connectives, viz $\land$, $\lor$, $\neg$, and $\leftarrow$. A sentence is made of propositions with connectives over such propositions. The knowledge that can be represented by a proposition is the relationship among the propositions. All queries are answered using logical inferences.

A logic program describing a problem consists of sentences, which are propositions, and which are predicates, expressing knowledge relevant to the problem. The formulation of this knowledge makes use of two basic concepts: the existence of discrete objects and the existence of relations between them. Objects are terms made up of constants and variables. The principal constituents of a logic program are objects, variables, propositions and connectives. Both objects and relations are given names - this creates symbolic models. Sentences express the relations or logical properties of the names.

A predicate is a predicate name with an n-tuple of arguments. Each of the arguments of a proposition is a term. Logic relations are named by predicate symbols and knowledge is expressed as sentences, formulas, constructed from predicates. A formula is defined as follows:

1) any predicate is a formula

2) if $F_1$ and $F_2$ are formulas, then so are $(F_1)$, $F_1 \land F_2$, $F_1 \lor F_2$, $F_1 \leftarrow F_2$, and
The use of variables in the arguments of predicates allows the writing of generalized sentences. The role of the variables in a sentence is governed by their mode of quantification (existential or universal).

The set of all sentences constructed with rules 1 - 4 given below constitutes the language of first order logic.

1) A term is a constant or a variable or an n-tuple of terms prefixed by a functor.

2) A predicate is a n-tuple of terms prefixed by a predicate symbol.

3) A formula is either a predicate or several predicates with connectives, with a quantifier for the variables appearing in the formula.

4) A sentence is a formula in which every occurrence of a variable is within the scope of the quantifier for that variable.

Consider the following logic program:

1. father(john, bob).
2. father(george, john).
3. father(george, donald).
4. mother(mary, bob).
5. mother(mary, jenniffer).
6. mother(chrstina, donald).
7. ∀ (parent(X, Y) ← mother(X, Y) ∨ father(X, Y))
8. ∀ (sibling(X, Y) ← parent(Z, X) ∧ parent(Z, Y))

The first six propositions in the logic program are facts. For example, the first fact, father(john, bob), expresses the father relationship between the objects john and bob. Similarly, the fourth fact, mother(mary, bob), expresses the mother relationship
between the objects mary and bob.

The father and mother relationship are both 2-tuple predicates. The proposition father(john, bob) can represent the information that john is the father of bob, i.e., the object which is the first argument is the father of the object that is the second argument.

The 7th and 8th propositions in the logic program given above are formulas, or rules. The knowledge represented by the formula \( \forall \text{parent}(X, Y) \leftarrow \text{mother}(X, Y) \lor \text{father}(X, Y) \) is that,

for all \( X \) and \( Y \), \( X \) is a parent of \( Y \) if \( X \) is the mother of \( Y \) or if \( X \) is the father of \( Y \).

This is expressed by the propositions which appear on the right hand side of the implication in the rule. Knowledge represented in the statements of logic programs are used during inferencing newer facts, or theorems.
2.2. Reasoning by Resolution

It is necessary to have a proof procedure that carries out the variety of processes involved in reasoning with statements in predicate logic. Resolution is such a procedure. The resolution proof procedure gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form. The convenient standard form is the clausal form.

A literal is an instance of a predicate. A clause is a disjunction of literals. The convenient clausal form is obtained from a general logic program by applying the following steps.

1. Convert all the formulae to *conjunctive normal form* - for example \(((a \land b) \lor c)\) is converted to \(((a \lor c) \land (b \lor c))\).

2. Converting all implications, \(\leftarrow\), from formulas using the fact that \((a \leftarrow b)\) is equivalent to \((\neg a \lor b)\).

3. The scope of the negation is reduced to the next literal using the fact that \((\neg(a \land b))\) is equivalent to \((\neg a \lor \neg b)\).

4. Normalize all quantifiers by moving all quantifiers to the left of the formula. This formula is said to be in *prenex normal form*.

5. Eliminating all *existential* quantifiers (\(\exists\)) by using assertions, i.e., facts. The resulting formula is said to be in *skolem normal form*.

6. The universal quantifiers are dropped as they are now implicit.

A clause which contains at most one negation is called a Horn clause. A Prolog program is a set of Horn clauses.

Resolution produces proofs by refutation. In other words, to prove a statement resolution attempts to show that the negation of the statement produces a contradiction, unsatisfiability, with the known statements. The resolution procedure is a simple iterative process. At each step two clauses, called the parent clauses, are compared.
(resolved) yielding a new clause, called the resolvent, that has been inferred from them.

Given two clauses \((a \lor b \lor \neg c)\) and \((\neg a \lor b \lor \neg c)\) the resolvent of it is \((b \lor \neg c)\), since either of a or \(-a\) is true and \((b \lor \neg c)\) is common to both the clauses. If \((b \lor \neg c)\) is true, giving an interpretation, then both parent clauses are true with this interpretation.

The resolvent represents ways that the two parent clauses interact with each other. If the resolvent is satisfiable then both parent clauses can be satisfiable simultaneously. An interpretation of a formula is the truth set, i.e., the set of values for the variables in the formula, which makes the formula true. An interpretation of the resolvent is an interpretation of the parent clauses. If resolution produces an empty clause, then the parent clauses cannot be satisfied simultaneously.

When resolution of clauses yields an empty clause, \(\Box^+\), then there exists a contradiction among the clauses. There may not be any contradiction among the clauses when the proof procedure is complete, in which case the clauses are said to be satisfiable. That is, when an empty clause is not derivable by the resolution proof procedure, then the clauses are said to be satisfiable. The resolution proof procedure is exhaustive.

An interpretation which satisfies a set of clauses is said to be its model. Resolution tests for satisfiability for all clauses with all combinations of interpretations available to find a model.

An interpretation over some domain \(D\) is said to satisfy a sentence iff (if and only if) it makes that statement to evaluate as true. It satisfies a set of sentences, \(S\), iff

\[\uparrow\, \text{This is similar to a state in which failure occurs in Prolog. Hence, the use of identical symbols.}\]
it satisfies each sentence in S. S logically implies some sentence s iff for all possible
domains every interpretation which satisfies S also satisfies s.

Sentences are of three types - denials, assertions or implications - of the form
(¬(A₁, A₂, ..., Aₙ)), A, or (A ← B₁, B₂, ..., Bₙ), respectively. A is the consequent and (B₁,
..., Bₙ) are antecedents of the implication.

If propositions are predicates, then we must consider all combination of values
for variables, i.e., bindings, to complete the proof procedure. The fundamental notion
underlying the reasoning process which explores the logical consequences of the
knowledge is the logical implication.

A substitution is an assignment of a set of terms to variables. A unifying substi­
tution when applied to A and B, makes A = B. A most-general-unifier (mgu) is a most
general substitution which makes A = B.

All computational problems can be formulated using only denials, assertions or
implications. All solvable problems can be solved by using general depth-first resolu­
tion.

Resolution possesses the soundness and completeness properties That is, every
theorem derivable by S is a valid formula and every valid formula is derivable by S,
respectively. Sentences consisting only of denials, assertions and implications form a
subclass of the clausal form and are called Horn clauses.

The resolution method is a good tool for testing the consistency of a set of Horn
clauses, i.e., Prolog programs, provided an adequate selection strategy is used. Algo­
rithmic properties of a function can be represented in a suitable set of clauses. The
values of that function can then be obtained by the resolution method.

Negation is implemented as "negation by failure", i.e. the negation of an expres­
sion is true if the expression is not true within the framework of the program (closed
world assumption). This means that when the negation of a goal is true, then the goal
is false, within the framework of the program.

The closed world assumption gives partial incompleteness. What this means is that the resolution proof procedure is accurate within the framework of the program - the assertions and implications supplied in the program.

The proof by refutation procedure uses the proof by negation principle, i.e., if a set of sentences is unsatisfiable, then each of the sentences is true. When a goal, G, is tested with a logic program, P, then the clauses $P \cup G$ is tested for satisfiability. If the clauses in $P \cup G$ are satisfiable then, G is false, because each clause in P is known to be true. If $P \cup G$ is unsatisfiable, then G is also true.

If the left hand side of an implication is a single literal, then we can convert this implication into conjunctive normal form using the rule that $(B \leftarrow A)$ is equivalent to $(\neg B \lor A)$. The proof procedure becomes very simple in this case, because a goal which is identical to B is unsatisfiable if A is unsatisfiable. In $B \leftarrow A$, B is called the head and A is called the body of the implication or rule. If the goal G is identical to the head of the clauses $C_1, C_2, ..., C_n$ and the body of each of the clauses are false, then G is unsatisfiable. A Prolog program applies this principle during query processing.

Unification and substitution form the basic activities for resolution theorem proving in first order predicate logic. The first activity during the proof procedure is to identify if the goal G is identical to the head, H, of a clause, C. This is achieved by unification of G with H.
2.3. Unification

The unification of two expressions can be described as follows

i) If either expression is a variable occurring in the other, then the expressions do not unify. (Occur check is done for assignments of the type $X := f(X)$ unification, which is an unreasonable event.)

ii) If either expression is a variable not occurring in the other, then unification succeeds and the latter to the former is the unification substitution. (A variable '$X$' unifies with another variable '$Y$'. The substitution is $X/Y$. A variable '$X$' unifies with a constant 'a', then $X$ takes a value 'a'; the substitution $X ← 'a'$ or $X/a$)

iii) If either expression is a constant and the other is either a different constant or a structured, term then the expressions do not unify. (A constant 'a' fails to unify with a constant 'b' or a structured term ['X a'].)

iv) If the expressions are structured terms having different principal functors, then the expressions do not unify. If the expressions are structured terms having identical functors, then the expressions unify, if each of the arguments unify. If any of the arguments fail to unify then the expressions fail to unify. All the substitutions which are applied during the unification of the arguments are recorded by the unification process to give the unification substitutions between the expressions.

A predicate name is a functor name. The arguments of the predicate are the arguments of the function. Variables and constants form the n-tuple arguments of a predicate functor.

The unification procedure in Prolog is essentially a pattern matching and substitution algorithm. The pattern matching proceeds matching the goal pattern G and the pattern H, which is the head of a clause C, from left to right. If the functors of G and
H are not identical, then G fails to unify with H. If the functors of G and H are identical then each argument of G is unified with the corresponding argument of H. Unification fails even if one of the arguments of G fails to unify with the corresponding argument of H. The left to right pattern matching and substitution quits at the first failure to unify.

Example

<table>
<thead>
<tr>
<th>G</th>
<th>H</th>
<th>Unify</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X, Y, a).</td>
<td>p(P, Q, R).</td>
<td>Yes</td>
<td>X &lt;- P; Y &lt;- Q; R &lt;- a</td>
</tr>
<tr>
<td>q(X, Y).</td>
<td>q([a, b], c).</td>
<td>Yes</td>
<td>X &lt;- [a, b]; Y &lt;- c</td>
</tr>
<tr>
<td>r(X, Y, Y).</td>
<td>r(a, a, a).</td>
<td>Yes</td>
<td>X &lt;- a; Y &lt;- a</td>
</tr>
<tr>
<td>r(X, Y, Y).</td>
<td>r(a, b, c).</td>
<td>Fail</td>
<td>Ø</td>
</tr>
<tr>
<td>s(a, b, c).</td>
<td>s(a, c, d).</td>
<td>Fail</td>
<td>Ø</td>
</tr>
</tbody>
</table>

If G unifies with H, then G is substituted by the body of C, with the unification substitutions applied to the subgoals in the body of C. G is true if the body of C is true. An interpretation of the body of C is an interpretation for G. Note that the body of C can contain more than one subgoal. Each of these goals is processed in the same way G was processed.
2.4. Procedural Interpretation

There are no factors to determine a precise course of execution of logic programs. The call selection rule is more casually called the computation rule. The standard computation rule always selects, in each step, the first call in the goal-list. The selected subgoal is resolved with the clauses. Since the clauses are implications, on unification with the rule-head, the subgoal is replaced by the rule-body. Then, to solve the subgoal the rule-body is solved. The standard search rule chooses clause for unification in the order of appearances in the program text.

A logic program can be executed as any declarative program. This is done by considering the goal as a procedure, the procedure name is the predicate name and the parameters are the arguments of the predicate. The procedures are defined as the clauses. The head of a clause is the header of the procedure and the subgoals in the body of the clause form the body of the procedure. The subgoals in a clause are executed when a procedure is executed. Note that there can be more than one clause for any predicate. Each of these clauses uniquely defines one procedure.

A Prolog program is formed by a set of instructions that can also be interpreted as productions of a definite clause grammar. The Prolog language uses resolution as an elementary execution mechanism. The choice of sequence of instructions to be executed in the course of program execution is determined from the text of the program. Each production is a two level process. When the goal matches with a rule-head, the goal is replaced by the rule-body and then the rule-body is executed instead of the goal.

The procedural interpretation [48] views the execution goals as a set of procedure calls. Each one is processed by calling an appropriate procedure. A goal-list $\{A_1, ..., A_n\}$ is a collection of procedures, $A_1, A_2, ..., A_n$. The procedure is the name of the predicate and its arguments are parameters. For any given program, the total com-
putation space is the set of all computations derivable from it using standard procedures calling operations.

The unification procedure draws upon a very important aspect of logic programs. The unification procedure finds a substitution which makes two literals identical. Using the procedural interpretation to execute a goal and using unification procedure to execute a call to the appropriate procedure leads to a property called I/O indirection. This means that the calling statement need not have all its arguments instantiated. The calling statement contributes some instantiation to the called procedure and the unification with the head of the called procedure and its execution contributes some instantiation to the calling statement.

It is understood that when this call becomes replaced by the invoked procedure body, the textual order of the latter's calls is not altered. An application of a rule confines computation to a selected subspace defined by the subspace of the computation of the subgoals in the rule. The total computation space is the computation tree whose root is the initial goal and whose paths from that node represent various computation. A program is said to be unsolvable when its subspace contains no successful computation.

The computation of a goal can be depicted as a goal-search tree. Every node in the tree represents the execution of a subgoal in the path to solving the goal. The substitutions applied to a node are indicated on the path from the root to the node and the instantiations generated by the successful execution of the goal corresponding to a node is indicated on the branches from the node. The number of branches from a node equals the number of alternative clauses that are applicable to a goal corresponding to a node. Failure of a goal is shown as "F" next to the node. Consider the rule

\[ \text{and the predicate relation given below. Figure 1 shows part of the standard goal-} \]

† This rule is take from Kundu, [34].
search-tree by Prolog, for this rule, until the first success is encountered. Table I gives the corresponding trace of execution, using the notations developed in section 4.1.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1(b1, c1)</td>
<td>p1(b1, c1). p2(a1). p3(c1). p4(a2, b1, d1). p5(c2, a2, d1, e1). p6(fl). p7(a2, f1). p8(b2). p9(a2, d1, f2). p10(b2, d1). p11(a2, c2).</td>
</tr>
<tr>
<td>p1(b1, c1)</td>
<td>p1(b1, c1). p2(a2). p3(c2). p4(a2, b2, d1). p5(c2, a2, d1, e1).</td>
</tr>
<tr>
<td>p1(b2, c2)</td>
<td>p1(b2, c2).</td>
</tr>
<tr>
<td>p6(f1)</td>
<td>p6(f1). p7(a2, f1). p8(b2). p9(a2, d1, f2). p10(b2, d1). p11(a2, c2).</td>
</tr>
<tr>
<td>p6(f2)</td>
<td>p6(f2). p7(a2, f2).</td>
</tr>
</tbody>
</table>
Figure 1. Standard Goal-Search-Tree by Prolog
<table>
<thead>
<tr>
<th>goal</th>
<th>success/fail</th>
<th>binding</th>
<th>backtrack literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1(Y, Z), ())</td>
<td>success</td>
<td>Y/b1; Z/c1</td>
<td></td>
</tr>
<tr>
<td>(p2(X), ())</td>
<td>success</td>
<td>X/a1</td>
<td></td>
</tr>
<tr>
<td>(p3(Z), (Z/c1))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), (X/a1, Y/b1))</td>
<td>fail</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p3(Z), (Z/c1))</td>
<td>fail</td>
<td>-</td>
<td>p2</td>
</tr>
<tr>
<td>(p2(X), ())</td>
<td>success</td>
<td>X/a2</td>
<td></td>
</tr>
<tr>
<td>(p3(Z), (Z/c1))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), (X/a2, Y/b1))</td>
<td>success</td>
<td>U/d1</td>
<td></td>
</tr>
<tr>
<td>(p5(Z, X, U, W), (Z/c1, X/a2, U/d1))</td>
<td>fail</td>
<td>-</td>
<td>p4</td>
</tr>
<tr>
<td>(p4(X, Y, U), (X/a2, Y/b1))</td>
<td>fail</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p3(Z), (Z/c1))</td>
<td>fail</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), (X/a2, Y/b1))</td>
<td>success</td>
<td>U/d1</td>
<td></td>
</tr>
<tr>
<td>(p5(Z, X, U, W), (Z/c1, X/a2, U/d1))</td>
<td>fail</td>
<td>-</td>
<td>p4</td>
</tr>
<tr>
<td>(p4(X, Y, U), (X/a2, Y/b1))</td>
<td>fail</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p3(Z), (Z/c1))</td>
<td>fail</td>
<td>-</td>
<td>p1</td>
</tr>
<tr>
<td>(p2(X), ())</td>
<td>success</td>
<td>X/a2</td>
<td></td>
</tr>
<tr>
<td>(p3(Z), (Z/c2))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), (X/a2, Y/b2))</td>
<td>fail</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p5(Z, X, U, W), (Z/c2, X/a2, U/d1))</td>
<td>success</td>
<td>W/c1</td>
<td></td>
</tr>
<tr>
<td>(p6(V), ())</td>
<td>success</td>
<td>V/f1</td>
<td></td>
</tr>
<tr>
<td>(p7(X, V), (X/a2, V/f1))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p8(Y), (Y/b2))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p9(X, U, V), (X/a2, U/d1, V/f1))</td>
<td>fail</td>
<td>-</td>
<td>p8</td>
</tr>
<tr>
<td>(p8(Y), (Y/b2))</td>
<td>fail</td>
<td>-</td>
<td>p7</td>
</tr>
<tr>
<td>(p7(X, V), (X/a2, V/f1))</td>
<td>fail</td>
<td>-</td>
<td>p6</td>
</tr>
<tr>
<td>(p6(V), ())</td>
<td>success</td>
<td>V/f2</td>
<td></td>
</tr>
<tr>
<td>(p7(X, V), (X/a2, V/f2))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p8(Y), (Y/b2))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p9(X, U, V), (X/a2, U/d1, V/f2))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p10(Y, U), (Y/b2, U/d1))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(p11(X, Z), (X/a2, Z/c2))</td>
<td>success</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table I: Trace of Standard Prolog Execution
2.5. Interpreter

The interpreter that executes a logic program has to remember what data has been assigned so far to the variables. It needs to remember how the locus of control evolved to its current state. This is needed to deal with the return operation after every successful procedure exit and the need to deal with backtracking, since that operation has to recall previously discovered opportunities still open for exploration.

Program execution accesses two important data areas. The first one contains some suitable compacted and codified facsimile of the input logic program and is static. The second area is highly dynamic and is used by the interpreter to record the history of its own actions. This is generally in the input heap and the execution stack. The stack represents both the execution control state and the data state.

The interpreter changes its attention from one control to another, and it traces out a path called the locus of control. Consider the following rules:

\[ P: A \text{ if } P_1, P_2, ..., P_n. \]
\[ Q: B \text{ if } Q_1, Q_2, ..., Q_n. \]

Assume that some of the subgoals in body of \( P \), \( P_1, ..., P_{n-1} \), have been solved. The subgoal \( P_i \) is tried next. If \( P_i \) matches with \( B \) then the procedure \( Q \) is executed. After \( Q_1, ..., Q_n \) is solved, control goes back to procedure \( P \) to execute \( P_{i+1} \). If any \( Q_i \) is unsolvable, then control goes back to \( P \) to try another candidate clause which are applicable for \( P_i \). (This is shallow backtracking). If there are no untried candidate clauses applicable for \( P_i \), then control goes back to \( P_{i-1} \). (This is deep backtracking.) The complete computation space is explored before computation stops.

Exploration of computation is achieved by creating and storing a frame for each procedure call on a linear stack called the local-stack. The procedure entered and not exited is called active. The frame is maintained as long as the procedure is active.
The frame can contain variable cells to know the variable values. But, the output assignments of a frame have to be undone on backtracking. If the frames contain cells for variables, then discarding the frame is enough to achieve backtracking. Having cells for variables in a frame leads to excessive use of space, since duplication is rampant. One can use another area called the trail which contains the values for the active variables. A cell with trail pointer is created in every frame. On deleting a frame, all assignments from the trail-pointer to the top of the trail are undone for effective backtracking.

A two stack representation has a local-stack and a global-stack. The local-stack contains the frames of the procedure calls. The global stack contains variables that are global to the called procedure. A cell for the global frame pointer is every frame.

In order to search the complete computation space, a goal-list is maintained. The goal-list contains all the goal that have to be solved before the primary goal is announced as a success or failure. Every procedure call modifies the goal-list and the local-stack as follows:

1. The first literal in the goal-list is chosen for execution.
2. A frame is created for this goal and is added to the top of the activation stack.
3. A procedure to solve this goal is selected.
4. If no procedure is available to solve this goal, then the goal is announced as a failure and backtracking is invoked.
5. If the procedure selected to solve the goal is an assertion, i.e., a fact, then the unification substitution that makes the procedure selectable is applied to the goal-list.
6. If the procedure selected to solve the goal is a rule, then the goal-list is the
resultant of appending the goal-list with the rule-body.

7. The above steps are repeated until the goal-list is empty - when a result to the primary goal is available.

If the stack is empty, then announce failure; else announce success and pronounce the substitution applicable to the primary goal as a result of the success.

A push occurs when the goal at the top of the goal-list is selected for execution. If the goal at the top of the stack unifies with a program clause, then the resolvent is added to the head of the goal-list. A pop occurs when there are no program clauses whose head matches the goal at the top of the stack. The goal at the top of the stack is added to the head of the goal-list. If the goal-list contains the body of a program clause, then this body is removed from the goal-list. The next choice of matching clause for the new top of stack is investigated. Computation continues in this manner until the goal-list is empty. This is the state $\square$, and success is reported. If the local-stack is empty then, the state $\blacksquare$ is reached and failure is reported. Figure 2 illustrates the execution of a Prolog program on a linear stack for two instances (marked "#") Figure 1.

The execution of a logic program is carried out by the interpreter. There are numerous issues pertaining to the implementation of interpreters for logic programs. These are described briefly in the next section.
Figure 2. Execution of Prolog on a linear stack.
2.6. Implementation Issues

Many implementation issues pertaining to implementation of an interpreter for Prolog have been addressed by researchers. Some of the key issues that are addressed are the data structures that are to be used and efficient management of dynamic data during execution.

Efficient memory management [28, 33] and backtracking [11 - 23] schemes can be used to avoid unnecessary computations and prudent searches. Last call optimization is a technique which deletes a frame from the activation stack, once the last call for a procedure has been activated.

One of the most recent result is based on the locality of reference in the execution of Prolog programs (the locality being the top of the stack). All but the page which contains the top of the stack is kept in the main memory. Hashing, indexing and special techniques for coding the programs are developed for efficient selection of clause for executing a goal [28, 33].

Warren [29, 31, 32] has developed a stack-based method of executing compiled Prolog clauses. The method employed by the Warren Abstract Machine (WAM) eliminates tail-recursion. WAM maintains certain information in registers to effect speedy selection of the backtrack literal on the stack, and resetting of the local-stack, trail and heap.

The significant results pertaining to efficient sequential execution of logic programs are described in the next chapter. These results are based on selective backtracking for re-exploration of the search space.
Chapter 3

3. Literature Review

3.1. Introduction

Research in the area of efficient execution of logic programs is exemplified by the Japanese Fifth Generation Computer Project, started in the 1980's. Towards this objective, researchers have been focusing on efficient implementation of logic programs, which would be useful in knowledge processing systems. Prolog has proved to be a successful programming language. In this chapter we describe in detail the results obtained by other researchers in their attempts to develop a model to efficiently execute logic programs.

The basic computational mechanism of Prolog is pattern matching on terms of logic. The matching expands the computation into a proof tree, until terms are shown to be true or false. In sequential execution, the proof tree is expanded in a depth-first manner. The goals of a clause are executed left to right and the goals of the most recent clause entry are executed first. The unification creates a binding of variables to terms and other variables, which affects further unification attempts. If, at any time the system fails to find a match for a goal, it backtracks, rejecting the most recently activated clauses and undoing any bindings made by the matches with their clause heads.

An obvious and simple technique in efficient execution of logic programs is identifying and backtracking to a better backtrack literal when a failure occurs during execution.
3.2. Selective Backtracking Based on Unification Mapping Information

The unification substitution mappings employed during goal search brings in dependencies among the different goals. Pereira and Porto [11] developed a model to use this information that is generated at runtime to select a better backtrack literal when a failure occurs during goal search.

In their model a goal execution is represented as a box with four ports, DO, DONE, REDO and UNDONE. The DO port of a goal is entered when the goal is first activated. The DONE port of a goal is exited on a successful execution of the goal. On backtracking to a goal, the control re-enters the goal execution box through the REDO port. Control exits through the UNDONE port of a goal on unsuccessful execution of a goal. Every goal execution box decomposes itself into similar boxes for the subgoals in a rule on unification of the goal to the head of the rule. Figure 3.1 shows the execution model developed by Pereira and Porto.

The main idea of selective backtracking is to select, at each failed goal, a single goal to backtrack to, not necessarily the previous goal as in standard backtracking in Prolog. Consequently, entry into the REDO port of G is allowed, only if G has been selected as the backtrack goal for the last failed goal; otherwise, control flows directly to the UNDONE port of G.

At the UNDONE port of a goal, in a rule, selection of the backtrack goal takes place. Deselection of the backtrack goal is done when backtracking to the goal occurs.

The backtrack goal is selected from among:

Ancestors of G

The goals whose alternative clauses avoid a reactivation of G (essentially the parent of G)
Modifying goals of G

The goals whose match if undone will undo failure (the goals on which G is dependent)

Legacy set of G

The set of candidate backtrack goals inherited from the failed goals that selected the backtrack goal

A dependency tag list representation is used to maintain all the dependencies generated at runtime, in a dependency-graph. After every unification this dependency-graph is modified. This graph maintains all the information required to do optimal backtracking.

The modifying goals of a failing goal, G, is constructed whenever G fails and the backtracking goal is selected. The overhead incurred in modifying and maintaining the dependency-graph, with every unification, is rather large and complex. Analyzing unification failures to construct the modifying goals is also expensive. Figure 3.2 shows the dependency tag representation and the modifying goals for two instances (marked "#") from Figure 1.
Figure 3.1. Recursive Execution Model

```
b1 ---- Y ---- 1 ---- Z ---- c1       b2 ---- Y ---- 1 ---- Z ---- c2

2 ---- X ---- a1                      2 ---- X ---- a2

4 ---- U ---- d1                      5 ---- W ---- e1

6 ---- V ---- f1
```

Figure 3.2 Dependency Graph representation

Figure 3. Backtracking based on Unification information. (Pereira et al.)
3.3. Run-time Intelligent Backtracking

The selection of the appropriate backtrack goal without maintaining the dependency-graph can be done using runtime information [12 - 16]. As execution of goal search proceeds, a deduction tree is built. A naive interpreter always considers the whole proof tree as a failing tree when a failure occurs. A smaller deduction subtree can be identified as the cause of failure. This could lead to pruning of redundant search space. Cox and Pietrzykowski [12, 13] and Bruynooghe and Matwin [14 - 16] describe methods to find a minimal deduction subtree in which unification is impossible, i.e., which holds no solution, and maximal deduction subtree in which unification is possible, i.e. could contain solutions.

Whenever a goal fails, the whole deduction tree becomes inconsistent. A minimal failing deduction tree can be constructed by analyzing the variable bindings, [16]. The clause choice, i.e., the program clause currently applied, to one of the leaf nodes of one of the minimal failing trees is changed and computation proceeds as usual. Alternatively, a maximal subtree that is consistent can be constructed by analyzing the variable bindings, and then a clause choice of one of the leaf nodes not in that subtree is changed and computation is continued as usual, [12, 13].

Bruynooghe [15] also presents a backtracking scheme in which dependencies of a goal P are constructed when P is unified with various clause heads. Subsequently, a scheme to generate and maintain the modifying goals of P is developed in the same paper.

A lot of information about runtime dependency and bindings has to be maintained to implement intelligent backtracking. Figures 4.1 and 4.2 show the minimal and maximal deduction subtrees described by Bruynooghe and Cox, et al., respectively.
Unification substitutions

\begin{align*}
X/a_2 & \text{ at } 2 \\
Y/b_1 & \text{ at } 1 \\
Z/c_1 & \text{ at } 1 \\
U/d_1 & \text{ at } 4
\end{align*}

1 and 3 do not share substitutions

**Figure 4.1.** Minimal Deduction Subtree that is consistent, of Brungoohee.

**Figure 4.2.** Maximal Deduction Subtree that is inconsistent, Cox.

On failure of \text{p}5 modify either \text{p}1\ or \text{p}3.

On failure of \text{p}5 modify either \text{p}1\ or \text{p}3.
3.4. Semi-Intelligent Backtracking Based on Static Data Dependency Analysis (SDDA)

In general, there are two kinds of backtracking: shallow backtracking and deep backtracking. Shallow backtracking is backtracking to alternative candidate clauses, when the head of the current candidate clause cannot unify with the calling literal. Deep backtracking occurs when there are no untried candidate clauses for the current procedure call, so backtracking occurs to any previously invoked procedure.

Dependencies among the subgoals in a rule are used for deep backtracking. A static-data-dependency-graph which represents the static dependencies of the subgoals in a rule, can be generated using the textual information about the rule. The SDDA of a rule is a directed acyclic graph and is independent of the runtime unification substitution information. Literals in the SDDA which are equidistant from the root are said to be in one layer.

Chang and Despain [17, 18] propose a method to select the backtrack literal from the SDDA, when a failure occurs. They construct an SDDA for each clause at compile time. They assume that if two literals do not share any unbound variables, then they are independent, because the dependence among literals which do not share a variable (static information) is generated at runtime. They define three types of backtracking.

Type I

The parent of a literal, in the SDDA of a rule, is chosen as a backtrack literal when a literal fails during forward execution, i.e., without producing any solutions.

Type II

The next left brother sibling in the same layer of the SDDA of a rule is selected as the backtrack literal when a literal fails after producing solutions.
in its forward execution. What this means is that the literal succeeded, but its solutions were not accepted by some successor of the literal in the SDDA of the rule.

**Type III**

Backtracking within a clause defaults to naive backtracking, as in Prolog, if the clauses succeeded.

The SDDA is used during type I and type II backtracking. Intelligent backtracking based on the SDDA is local to a procedure (rule). This reduces the number of redundant steps and its effect is inter-procedural.

Literals in the same layer of an SDDA of a clause (rule) are assumed to be independent during the forward execution. The literals in a layer may not be independent if any literal in the next layer has been executed.

The backtracking based on SDDA does not always give the best selection of a backtrack literal, but avoids excessive runtime overhead. Hence, this method of backtracking is called semi-intelligent backtracking. Also, many SDDAs can be constructed from a given rule. There is no systematic way of identifying the best SDDA.

Figure 5.1, 5.2, and 5.3 show the use of SDDA for Type I, Type II and Type III backtracking, Chang and Despain [17 -18], for two instances (marked ") from Figure 1.
5.1. Type I backtracking based on SDDA

On failure at 6 backtrack to 0.

Static Data Dependency Graph (SDDA)
Backtracking after Forward Execution. On failure at 6 after failure at 9 backtrack to 4.

5.2. Type II backtracking based on SDDA

Backtracking after succeeding by a rule.
On failure at 9 backtrack to 8.

5.3. Type III backtracking based on SDDA

Figure 5.
Semi-Intelligent Backtracking based on SDDA. (Chang et al.)
3.5. Generator-Consumer Approach

The generator-consumer relationship among the literals in a rule is based on the dependencies that are caused by sharing variables by the literals. This relationship gives information similar to the information obtained by the data-flow analysis of the literals in a rule. Conery and Kibler [19 - 23] use the generator-consumer information in a rule to select a backtrack literal. This scheme incurs small overhead and yet can eliminate redundant backtracking in many problems.

The generator-consumer relationship can be described as follows. One of the literals sharing a variable X is considered the generator of X, the other literals which use the variable X are considered as consumers of X. The generator-consumer relation can be depicted as a data dependency graph D, which is a directed graph, in which a literal L is the parent of another literal M, if for some variable X, M is the generator of X and L is a consumer of X.

Whenever a goal P, fails the backtrack point is chosen from the set of goals that have generated the values of the variables X, ... , X, that occur in the arguments of P,. To enable the construction of this set of goals, some extra information is maintained with each variable - namely, the generator of the binding of the variable. The backtrack goal selection using the generators of the variables of the failing goal is not precise, but runtime overhead is small.

Whenever unification of a goal P with some clause head results in assignment of a value to a variable, X, a tag P is attached to the variable X, indicating that the value to the variable X was generated at the goal P. This assignment is undone when backtracking takes to P, or to the left of P. This tag is unique at any moment due to the single assignment property of logic programs - there can be only one assignment to a variable at a time.

When a goal P fails, then the modifying goals of P are the generators of the
variables used by P and the parent of P - the goal that unified with the head of the clause containing P. The B-list with a literal P is the order list of the modifying goals of P. Backtracking takes the first element of B-list. This method leads to backtracking to any point in the proof tree rather than to the clause containing the failed goal. This is because the generator/consumer relationship that transcends clause boundaries is also maintained. Figure 6 illustrates the use of B-list, by Conery and Kibler, for backtracking for two example from Figure 1.

The scheme developed by Conery and Kibler misses solutions under situations similar to the type II [17 - 18]. This was cured, independently, by Kumar [24 - 25] and Woo and Choe [26], independently.
<table>
<thead>
<tr>
<th>goal</th>
<th>B-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[0]</td>
</tr>
<tr>
<td>2</td>
<td>[0]</td>
</tr>
<tr>
<td>3</td>
<td>[2]</td>
</tr>
<tr>
<td>4</td>
<td>[2, 1]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 2, 1]</td>
</tr>
<tr>
<td>6</td>
<td>[0]</td>
</tr>
<tr>
<td>7</td>
<td>[6, 2]</td>
</tr>
<tr>
<td>8</td>
<td>[1]</td>
</tr>
<tr>
<td>9</td>
<td>[6, 4, 2]</td>
</tr>
<tr>
<td>10</td>
<td>[4, 1]</td>
</tr>
<tr>
<td>11</td>
<td>[2, 1]</td>
</tr>
</tbody>
</table>

**Figure 6.** Backtracking based on B-list, of Conery and Kibler.
3.6. Modified B-list Approach

The approach by Kumar [24 - 25] and Woo and Choe [26] is as follows.

When a literal Q is selected as the backtrack goal, when a goal P fails, Q being the most recent goal in modifying(P), Q is also passed the \{(modifying(P) - \{Q\)}\). When Q fails the latest goal from (modifying(Q) \cup \{(modifying(P) - \{Q\)}) is selected as the backtrack goal. The major difference here is the use of dynamic B-lists. The backtracking method based on dynamic B-lists is shown to be complete. Figure 7 illustrates the modified B-list, approach by Vipin Kumar, for two instances (marked "#") of Figure 1.
<table>
<thead>
<tr>
<th>goal</th>
<th>B-list</th>
<th>goal</th>
<th>B-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[0]</td>
<td>1</td>
<td>[0]</td>
</tr>
<tr>
<td>2</td>
<td>[0]</td>
<td>2</td>
<td>[0]</td>
</tr>
<tr>
<td>4</td>
<td>[2, 1]</td>
<td>4</td>
<td>[2, 1]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 2, 1]</td>
<td>5</td>
<td>[4, 2, 1]</td>
</tr>
<tr>
<td>6</td>
<td>[0]</td>
<td>6</td>
<td>[4, 2]</td>
</tr>
<tr>
<td>7</td>
<td>[6, 2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>[6, 4, 2]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B-list after backtracking from 9.

Backtrack literal is the first element in the B-list with a literal.

Figure 7. Backtracking based on dynamic B-list, of Vipin Kumar, et al.
3.7. Variable Based Intelligent Backtracking

An approach based on variables, rather than the predicates, is presented by Rajasekhar and Narasimha Murthy [27]. The variables used by the goal is used for identifying the backtrack literal. These results developed are comparable to those of Kumar in terms of complexity, but could lead to the elimination of a lot of redundant backtracking. A tag with every variable is maintained. This tag indicates where the variable got its binding, its present instantiation. On failure of a goal, the tags of the variables used by the goal is examined in order to select the backtrack literal. The modifying goals are maintained similarly as with Kumar's method. This scheme is not very easily implemented on a linear stack.

Figure 8 illustrates the use of variable tags for backtracking for two instances (marked "#") of Figure 1.
Backtrack based on the variables used by the failing literal.

Figure 8. Variable Based Backtracking, of Rajasekhar, et al.
3.8. Warren Abstract Machine

Prolog requires a stack, called the local-stack, to hold frames for each active procedure. The local-stack contains all the bookkeeping information together with the value cells for local variables. A frame is reclaimed only on backtracking.

The Warren Abstract Machine (WAM) [29 - 32] is the most efficient method for executing Prolog programs by elimination of tail recursion. In WAM, the local-stack contains only the records of the goals in the current path which have pending untried alternatives. A frame is reclaimed when its last alternative is used up. Maintaining the most recent choice-points in a register and employing structure sharing rather than structure copying leads to further efficiency [28].

The code for the clauses reside on a heap and execution unfolds on local, global and trail stacks. The local-stack contains the variables of the clause, along with the environment pointer to manage code and data. The global-stack holds complex structures and the trail keeps pointers indicating the unification substitutions created by the goals. The global-stack holds complex structures, such as lists, and the trail keeps pointers to variables which are not local to the current clause and are assigned during unification, in order to undo their bindings during backtracking. As each goal is called for unification, an activation record is created on the local stack for its variables. New structures created by the goal are placed on the global-stack.

Activations on the local-stack can be deallocated after a goal returns and has no alternate clauses to try (called success popping) or when backtracking occurs. Global locations are kept until backtracking occurs. Stack allocation and deallocation of the stacks eases storage reclamation substantially, reducing
runtime storage needs.

On the local-stack, there can be several goals with alternatives yet to be tried. These goals are called choice-points. When unification fails, the most recent goal, which is called the choice-point is selected for backtracking. The last-choice register maintains this choice-point and helps in this selection. Each activation maintains the last-choice pointer, which is loaded to the register when this goal is executed. Each activation also maintains a fail point pointing to its next alternate, yet to be tried. The trail is unwound after backtracking by looking at the choice-point register, in order to undo some of the unification substitution.

Thus, when a goal fails, the processor need only look at the choice-point register to find the correct environment to restore. The trail is unwound back to the beginning of that choice-point goal by popping each address and tagging the location as undefined. Shallow backtracking needs only transfer of control to the fail point, but deep backtracking needs to clear the local- and global- stack and reload stack pointers from the choice-point activation and then transfer control to the fail point of the choice-point goal.

Figure 9 illustrates the information on the WAM local-stack for two instances (marked "#") of Figure 1.
The departure from the conventional Prolog execution in WAM is that instead of passing the information about the caller to the procedure, it is now given a pointer to the actual goal to be executed next on backtracking, called the choice-point. The choice-point points to the most recent ancestor with further alternatives left to execute.

The execution is efficient in space and time by elimination of unnecessary activations on the stack and by prior knowledge of alternatives in the form of choice-points.
Figure 9. Local-stack used by WAM, of Warren.
3.9. Discussion

The techniques described in sections 3.1 - 3.6 have significant inherent drawbacks in effective implementation for savings in sequential implementation of Prolog. These drawbacks arise due to excessive run-time time and space requirements for

runtime maintenance of unification substitution information, and
maintenance of dynamic data dependency graph.

Severe difficulties in implementations of these results also arise due to the implementation of Prolog on a linear stack.

The runtime cost is measured in terms of the amount of work done during execution and the amount of dynamic space that is required. A method is sound if every solution obtained by the method is valid (correct). A method is complete if every solution to a problem (valid or invalid) is obtained by the method. The following table compares the various results for their time and space complexity together with the savings in computation.
Table II: Comparison of Results in Literature

Parallel execution models and parallel logic programming languages are being developed for faster execution of logic programs [36 - 54]. Table II compares the performance of the existing results in literature. These methods do not address the central issues for intelligent search, described in section 2.4. This dissertation addresses these issues, leading to intelligent search; it proposes and presents efficient implementable algorithms.
4. Intelligent Backtracking Based on Failure Bindings

4.1. Introduction

When a goal fails, the conventional assumption is that the cause of failure of the goal is the set of all bindings applied in obtaining the goal. In this chapter the notion of a failure-binding for a literal is defined. This definition is extended to predicates. A failure-binding for a literal is a precise cause of failure of the goal which is obtained from the literal. This precise cause is a member of the set containing the bindings applied to obtain the goal. There can be zero or more failure-bindings for a literal identified when a goal fails. This depends on the information content of the logic program.

A method which uses a bit-vector to identify the failure-bindings of a literal is presented. Intelligent backtracking based on failure-bindings of the currently failed literal is presented. The savings in redundant backtracking by intelligent backtracking based on the failure-bindings of a literal is illustrated.

The search based on failure-bindings, $\zeta$, is shown to satisfy the completeness and soundness properties of Prolog. $\zeta$ also obtains the results of a goal in the same order as obtained by Prolog. Forward and Backward Execution Algorithms for $\zeta$ are given.

Consider the example in Figure 1. When the failure of $p5(Z, X, U, W)$ with

see [59].

A literal is an instance of a predicate. The arguments of the predicate are substituted by terms made up of constants and variables to give a literal.

A more general definition of failure binding can be found in [34].
the bindings Z/c1, X/a2, U/d1 is detected, then the backtrack literal selected by
dependency directed backtracking is the latest of the generators of Z, X and U.
This is the generator of U, i.e. p4. Hence, dependency directed backtracking
selects p4 as the backtrack literal. On analysis of the predicate relation of p5, it
can be seen that p5 can never succeed with Z/c1. p5 will never succeed, unless
the binding Z/c1 is changed.

We show that such precise individual causes of failure of a goal can be
identified and used effectively for pruning goal search tree. The premise to such
identification is that every failure is due to failure during unification, which is a
pattern matching process, and a failure occurs only after exploring the whole
goal search tree of a goal.

The gain in efficiency derived by employing intelligent backtracking based
on failure-bindings is, however, not without a cost. Additional work is required
in identifying and maintaining the causes of failure during the search. The
amount of work done in identifying and maintaining the causes of failure is
overshadowed by the savings derived by their use, as is described in the later sec­
tions.
4.2. Failure-Binding

When a goal $G$ fails, the causes of failure of $G$ is not known. In fact, the cause of failure is attributed to the set of bindings applied to obtain $G$. Intelligent backtracking methods described in [5 - 13] perform relevant search. These methods still suffer in performance due to insufficient information about the cause of failure of a goal.

In a logic program a variable can be in either of the three states: Free, Ground or Partially-instantiated. In a Ground state the variable is instantiated to a ground value. For example, $X/a$ means $X$ is in the Ground state with a ground value 'a'. In a Free state the variable is completely uninstantiated. In the Partially-instantiated state, part of the variable is in the Ground state and part of it is in the Free state. If $X/[Y, Z]$ with $Y/b$ and $Z$ in the Free state indicates, then $X$ is said to be in the Partially-instantiated state.
A Prolog literal is an instance of a predicate with many variables that form its arguments. There are no restrictions on the state of these variables that form the arguments of the goal (i.e., there are no mode restrictions). A consequence of a mode free program is the I/O indirection of Logic Programs. During execution, a set of bindings is applied to a prolog literal to obtain a goal. The success of a goal may generate zero or more newer bindings to the variables which were in the Free state at the time of the call of the goal. A goal fails if there is no solution to the predicate with the bindings applied to obtain the goal. The goal can never succeed with this set of bindings. A different set of bindings is necessary to satisfy the predicate.

**Definition 4.1.** A prolog goal $g$ for a predicate $p$ is a pair $g = (p', \alpha)$ where

1. $p'$ is an instance of $p$.
2. $\alpha$ is a set of instantiations applied to all variables in $p'$.

Alternately we represent the goal $g$ with the argument tuple of $p'$ with the substitutions from $\alpha$ applied to all the variables in $p'$.

---

† An instance of $p$ is a literal of $p$, with its arguments substituted by terms.

§ A Prolog goal $g = (p', \alpha)$ is also an instance of the predicate $p$.

Notation suggested by Kundu, [33].
Definition 4.2: Let \( g \) be a prolog goal, \( g = (p', \alpha) \). The Variable-set of \( g \), denoted \( V_g \), is the set of all the Free variables in \( g \). The Constant-set of \( g \), denoted \( C_g \) is the set of all the constants in \( g \).\(^1\)

**Example**

<table>
<thead>
<tr>
<th>predicate</th>
<th>goal</th>
<th>argument tuple representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(\text{arg1, arg2, arg3, arg4}) )</td>
<td>( (p(X, Y, d, Z), (X/a)) )</td>
<td>( p(a, Y, d, Z) )</td>
</tr>
<tr>
<td>( q(\text{arg1, arg2, arg3}) )</td>
<td>( (q(P, Q, R), (R/[a, b])) )</td>
<td>( q(P, Q, [a, b]) )</td>
</tr>
<tr>
<td>( r(\text{arg1, arg2}) )</td>
<td>( (r(a, b), {}) )</td>
<td>( r(a, b) )</td>
</tr>
<tr>
<td>( s(\text{arg1, arg2, arg3}) )</td>
<td>( (s(P, Q, d), {}) )</td>
<td>( s(P, Q, d) )</td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>instance of predicate</th>
<th>Variable-set of instance</th>
<th>Constant-set of instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(a, Y, d, Z) )</td>
<td>( {Y, Z} )</td>
<td>( {a, d} )</td>
</tr>
<tr>
<td>( q(P, Q, R) )</td>
<td>( {P, Q, R} )</td>
<td>( {} )</td>
</tr>
<tr>
<td>( r(a, b) )</td>
<td>( {} )</td>
<td>( {a, b} )</td>
</tr>
<tr>
<td>( s(P, Q, d) )</td>
<td>( {P, Q} )</td>
<td>( {d} )</td>
</tr>
</tbody>
</table>

The successful execution of a goal \( g \) generates a set of new instantiations for the variables in \( V_g \). Due to the single assignment rule, a successful execution of a goal can assign values only to variables whose instantiations are not defined in \( \alpha \). We

\(^1\) \( g \) uniquely defines \( V_g \) and \( C_g \).
denote this set of newer instantiations generated when \( g \) succeeds by \( \Delta_0 \), or simply \( \Delta \). \( \Delta \) can possibly be empty. When \( g \) fails, \( \Delta \) is necessarily empty.

A goal \( g = (p', \alpha) \) fails if every alternative applicable to the predicate \( p \), of which \( p' \) is an instance, fails to solve \( p'' \). \( p'' \) fails if

1) \( g \) fails to unify with any fact of \( p \),
2) \( g \) fails to unify with the rule-heads of some rules for \( p \), and
3) every rule for \( p \), \( p :\cdot p_1, p_2, \ldots, p_m \), \( 1 \leq j \leq k \), which is applicable to \( g \) (there are \( k \) rules applicable to \( p'' \)) fails to solve \( g \).

The classical assumption is that the cause of failure of \( p' \), which gave \( g \), is \( \alpha \), since \( p' \) will always fail with this particular combination of bindings in \( \alpha \). The following definition is used for identifying precise cause of failure for \( p' \).

**Definition 4.3:** If the goal \( p'' = (p', \{X/v\}) \) fails, then \( X/v \) is a failure-binding for \( p' \). Similarly, if \( p'' = (p', \{\}) \) with the constant-set of \( p' C_p = \{c\} \) is false, then \( c \) is a failure binding for \( p' \).

It can be observed from the definition of a failure-binding of \( p' \) that if \( X/v \) is a failure-binding for \( p' \), then any goal \( g' \) obtained by applying \( X/v \) to \( p' \) and any other values for other variables in \( V_p \), is also false. In general there can be more than one failure-binding for \( p' \).

The failure-bindings for \( p' \) are identified by combining the causes of failure for \( p' \) from the failure of a general goal \( g = (p', \alpha) \) by each alternative of \( p \).

When a goal \( g = (p', \alpha) \) fails, the causes of failure for \( p' \) with respect to each of the alternatives are identified. The cause of failure for \( p' \) by an alternative is obtained from the facts:

---

* Personal communication, Kundu [33].
Fact 1:
A binding $X/v \in \alpha$ is a cause of failure for $p'$ due to a fact of $p$ if $v$ fails to unify for $X$ in the fact.

Fact 2:
A constant $c \in C_p'$ is a cause of failure for $p'$ due to a fact of $p$ if $c$ fails to unify in the fact.

Fact 3:
A binding $X/v \in \alpha$ is a cause of failure for $p'$ due to an alternative of $p$ which is a rule if

a) $v$ fails to unify for $X$ in the rule-head of the rule or

b) part of $v$ is a failure-binding of some subgoal in the rule-body of the rule. The unified version of the rule, $r''$, obtained after $g$ unifies with the rule head of the rule $r$

$$r : p : p_1, p_2, p_3, ..., p_m.$$

is

$$r'' : p'', p'' : p_1'', p_2'', ..., p_m''.$$

$X/v \in \alpha$ is a cause of failure for $p'$ by the rule $r$ if there exists some $X'/v'$, $v'$ is a part of $v$, obtained via unification of $g$ to the rule-head of the rule, and $X'/v'$ is a failure-binding of some subgoal $p_j$ in the rule.

Fact 4:
A binding $(c \in C_p')$ is a cause of failure for $p'$ due to an alternative of $p$ which is a rule if

a) $c$ fails to unify in the rule-head of the rule or

b) part of $c$ is a failure-binding of some subgoal in the rule-body of the rule. The unified version of the rule, $r''$, obtained after $g$ unifies with the rule head of the rule $r$

$$r : p : p_1, p_2, p_3, ..., p_m.$$
is

\[ r'' p'', p'' : - p1'', p2'', \ldots, pm''. \]

c is a cause of failure for \( p' \) by the rule \( r \) if there exists some \( c' \), which is a part of \( c \), obtained via unification of \( g \) to the rule-head of the rule \( c' \) is a failure-binding of some subgoal \( p_1'' \) in the rule.

**Fact 5:**

If there are no rules or facts of \( p \), then every member of \( \alpha \) and every member of \( C_\varphi \) is a cause of failure for \( p' \).

A failure binding for \( p' \) can be obtained from the complete goal-search-tree of \( g' = (p', \alpha) \), which is known when the failure of \( g \) is detected.

1. If the goal-tree of \( g \) is of height zero, i.e., there are no facts or rules of \( p \), then every \( X/v \in \alpha \) and every \( c \in C_\varphi \) is a failure-binding for \( p' \) (Fact 5).
2. If the goal-search-tree of \( g \) is of height one, i.e., there are only facts of \( p \) but no rules, then a binding \( X/v \in \alpha \) (or a constant \( c \in C_\varphi \)) is a failure-binding for \( p' \) if \( X/v \ (c) \) is a cause of failure for \( p' \) with all the facts of \( p \), by Fact 1 (Fact 2).
3. If the goal-tree of \( g \) has height \( \geq 2 \), i.e., there are facts and rules of \( p \), a binding \( X/v \in \alpha \ (c \in C_\varphi) \) is a failure-binding for \( p' \) iff
   
   (a) \( X/v \ (c) \) is a cause of failure for \( p' \), according to Fact 1 (Fact 2) with respect to each of the facts of \( p \) and
   
   (b) \( X/v \ (c) \) is a cause of failure for \( p' \), according to Fact 3 with respect to each rule of \( p \), by Fact 3 (Fact 4).

**Example**

† We only consider goal-trees which are finite, i.e., there is no infinite branch.

* This definition is my rewording of notes from Kandu [33].
For the goal \( p'' = (p(X, b, Z), (X/a1, Z/c2)) \) with no facts and rules for \( p \) the failure-bindings for \( p(X, b, Z) \) are \( b, X/a1 \) and \( Z/c2 \).

**Example**

Consider that the predicate \( q \) is defined only by the facts given below.

<table>
<thead>
<tr>
<th>Facts of ( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q(a2, b2, c1, d3). )</td>
</tr>
<tr>
<td>( q(a3, b2, c2, d1). )</td>
</tr>
<tr>
<td>( q(a2, b3, c3, d2). )</td>
</tr>
</tbody>
</table>

The failure-bindings for different instances of \( q \) according to (2) above is shown in Figure 10.

<table>
<thead>
<tr>
<th>goal</th>
<th>failure-bindings</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q(P, Q, R, d1), (P/a2, Q/b1) )</td>
<td>( Q/b1 )</td>
<td>( b1 ) fails to unify for ( Q )</td>
</tr>
<tr>
<td>( q(a1, S, T, d1), (T/c2) )</td>
<td>( a1 )</td>
<td>( a1 ) fails to unify</td>
</tr>
<tr>
<td>( q(a1, Y, Z, W), (Y/b1, Z/c3) )</td>
<td>( a1; Y/b1 )</td>
<td>( a1 ) fails to unify ( b1 ) fails to unify for ( Y )</td>
</tr>
<tr>
<td>( q(X, Y, Z, d1), (X/a2, Z/c3) )</td>
<td>none</td>
<td>no common cause of failure</td>
</tr>
</tbody>
</table>

Figure 10. Identification of Failure-Bindings when a predicate is defined by facts.

Consider the goal \( q(P, Q, R, d1), (Q/b1) \) in Figure 10. The value \( b1 \) fails to unify, for \( Q \), in each of the facts for \( q \). Hence, \( Q/b1 \) is a failure-binding for \( q(P, Q, d1) \). Consider another goal \( q(a1, S, T, d1), (T/c2) \). The value \( a1 \) fails to unify in each of the facts for \( p \). Hence, \( a1 \) is a failure-binding in \( q(a1, S, T, d1) \). Two failure-

\[ \dagger \text{ From discussions with Kundu, [33].} \]
bindings for \( q(a_1, Y, Z, W) \) can be identified when the goal \( (q(a_1, Y, Z, W), \{Y/b_1, Z/c_3\}) \) fails, because \( a_1 \) does not unify in each of the facts for \( p \) and \( b_1 \) does not unify for \( Y \) in each of the facts for \( p \). No failure-bindings for \( q(X, Y, Z, d_1) \) can be identified when the goal \( (q(X, Y, Z, d_1), \{X/a_2, Z/c_3\}) \) fails, because none of \( a_2, c_3, \) or \( d_1 \) fails to unify in each of the facts for \( p \).
Example

Figure 11. shows the identification of failure-binding for \( r(X, Y) \) using the facts and rules of \( r \).†

Consider the following facts and rules for the predicate \( r \) and the goal \( r'' = (r(X, Y), \{X/a1, Y/b1\}) \).

\[
\begin{align*}
1. & \quad r(a1, b2) . \\
2. & \quad r11(a1) . \\
3. & \quad r12(b2) . \\
4. & \quad r21(a1, c1) . \\
5. & \quad r22(c1, b2) . \\
6. & \quad r(X, Y) :- r11(X), r12(Y) . \\
7. & \quad r(X, Y) :- r21(X, Z), r22(Z, Y) .
\end{align*}
\]

<table>
<thead>
<tr>
<th>alternative</th>
<th>cause of failure of ( r(X, Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Y/b1 )</td>
</tr>
<tr>
<td>6</td>
<td>( Y/b1 )</td>
</tr>
<tr>
<td>7</td>
<td>( Y/b1 )</td>
</tr>
</tbody>
</table>

Figure 11. Failure-binding in the presence of facts and rules.

The facts of \( r \), listed above, suggest that \( Y/b1 \) is a failure-binding for \( r(X, Y) \). The first rule of \( r \) fails to solve \( r(a1, b1) \), because \( p12(Y) \) fails and \( Y/b1 \) is a failure binding of \( r12(Y) \). This suggests that \( Y/b1 \) is a failure-binding for \( r(X, Y) \). The second rule proceeds with the computation of \( r21(X, Z) \), which succeeds with a solution \( Z/c1 \). The goal \( r22(Z, Y) \) fails and \( Y/b1 \) is a failure-binding for \( r22(Z, Y) \). The

† This example was given by Kundu, [33].
suggestion that Y/b1 is a failure binding in r(X, Y) still holds. There are no more alternatives for r. Thus, the suggestion that Y/b1 is a failure-binding for r(X, Y) is substantiated by all the facts and rules of r. Note that Y/b1 is still the failure-binding for r(X, Y), even if the fact p21(a1, c1) is removed." In general, removal of any fact or rule cannot eliminate or invalidate any failure-bindings in that if X/v is a failure-binding in g then on removing any fact or rule it will still be true that X/v is a failure-binding in p". Alternatively, the addition of a fact or a rule can easily eliminate or invalidate a failure-binding for g.

It is interesting to note that a failure-binding which is identified by analyzing the goal-tree of a goal, g = (p', α), is applicable to any other goal, g' = (p', α'). What this means is that every goal which is obtained from p', by applying a failure-binding for p', will fail. If the arguments of an instance are either variables or constants, but no generalized terms, then a failure-binding for p' is applicable for any instance of the predicate p, i.e. any goal g" = (p", α"") will fail if either C_p or α", or both, contain one or more failure-bindings for any instance p' of p. The substitution of these failure-bindings in p" should correspond to those in p'.

**Theorem 4.1.** If X/v ∈ α (c ∈ C_p) is a cause of failure for p', obtained by analyzing the goal-tree of g = (p', α) which fails, due to every alternative, then X/v (c) is a failure-binding for p'.

**Proof:** By definition, if there are no alternatives for p, then every X/v ∈ α and every c ∈ C_p is a failure-binding for p'. If X/v (c) is a cause of failure for p' by each alternative, then either v failed to unify for X (c failed to unify) in each of

† Developed in conjunction with Kundu, [33].

‡ A more general definition of failure-binding, and its use, is considered by Kundu [33 - 34].
the facts of \( p \), and in some rule-heads of \( p \), or part of \( v \) (c) is a failure-binding of some subgoal in every rule that is applicable to \( g \). Effectively, this says that \( X/v \) (c) individually would cause failure of \( g \) with each alternative of \( p \). This will then be true if \( \alpha \) contained only \( X/v \) (\( C_r \) contained only c). Hence, every alternative of \( p \) will fail a goal \( p'' = (p', \{X/v\}) \) \( (p'' = (p', \{})) \), \( C_r = \{c\} \), leading to failure of \( p'' \). Thus \( X/v \) (c) is a failure-binding for \( p' \).

**Theorem 4.2.:** If \( X/v \in \alpha \) (c \( \in C_r \)) is a failure-binding for \( p' \), then \( X'/v \in \alpha' \), or \( v \in C_{r'} \), (c \( \in C_{r'} \), or \( X'/c \in \alpha' \)) is a failure-binding for \( p'' \) if \( X' \), or \( v \), in \( p'' \) corresponds to \( X \) in \( p' \) (c, or \( X' \), in \( p'' \) corresponds to c) in \( p' \).

**Proof:** Since \( X/v \) is a failure-binding for \( p' \) we can say that \( v \) (c) fails to unify for \( X \) in the facts of \( p \) and in some rule-heads of the rules of \( p \) and either \( v \) (c) or parts of \( v \) (c) are failure-bindings of some subgoal in each of the the rules of \( p \). The same facts and rules are applicable for \( p'' \). Since \( X' \), or \( v \), in \( p'' \) corresponds to \( X \) in \( p' \) (c, or \( X' \), in \( p'' \) corresponds to c) in \( p' \) the same cause of failures for \( p' \) are applicable to \( p'' \). Hence, \( X/v \) (c) is a failure-binding for \( p'' \).

The definition of a failure-binding can be generalized for the case of a compound goal containing several ANDed goals (such as a rule body). In this case, we say that a binding \( X/v \) (constant c) is a failure-binding for the compound goal if it is a failure-binding for at least one of the literals in the compound goal.

Consider the facts and rules of \( r \) in Figure 11, without the fact \( r_{21}(a_1, c_1) \). There are two failure-bindings for the compound goal "\( p_{21}(X, Z), p_{22}(Z, Y) \)", namely, \( X/a_1 \) (from \( p_{21}(X, Z) \)) and \( Y/b_1 \) (from \( p_{22}(Z, Y) \)).

---

§  Personal communications, Kundu [33 - 34].
The definition of a failure-binding for a literal can also be generalized from the case of a single binding of $X/v_1$ to the case of a case of a set of bindings $(X/v_1, X/v_2, ..., X/v_n)$, a set of constants $(c_1, c_2, ..., c_n)$, or a set of bindings and constants, which together cause a failure of the literal. We call such a composite set of bindings that makes the literal false a composite failure-binding for the literal.† The conventional assumption that when a goal $g = (p', \alpha)$ fails, the cause of failure is $\alpha \cup \mathcal{C}_p$. This gives us a composite failure-binding.

Example

In Figure 10, we have $(X/a_3, Z/c_3)$ as a composite failure-binding for $q' = q(X, Y, Z, W)$, because the goal $(q(X, Y, Z, d_1), \{X/a_3, Z/c_3\})$ fails.‡

The advantage of using the composite failure-bindings is that these may be encountered more often than the single variable failure-bindings. Put in another way, the composite failure-binding is a weaker notion than the single variable failure-binding. The generalization from singular failure-bindings to composite failure-bindings is straightforward. The problem, however, is that the composite failure-bindings can be numerous. Their identification and maintenance is much more complex and expensive than identification of singleton failure-bindings.

† Personal communications, Kundu [33-34].

‡ Personal communications, Kundu [33-34].
4.3. Failure Binding Table

There is a failure binding table $FBT(p)$ for each predicate $p$. The tuples of $FBT(p)$ represent a failure-binding applicable to $p$. Every tuple of $FBT(p)$ represents one failure-binding applicable to $p$. The tuples of $FBT(p)$ are obtained as follows:

1. Let $X/v$ (c) be a failure-binding for $p'$.
2. We obtain $p''$ from $p'$ by substituting every constant in $C_{p'}$ by a dummy variable # (obtain $p''$ from $p'$ by substituting every constant in $C_{p'}$, other than $c$, by a dummy variable #). The dummy variable # always unifies.
3. The argument tuple of $(p'', (X/v))$ (the argument tuple of $(p'', (\{\}))$) is added to $FBT(p)$.

Example The entries to the $FBT(q)$ from Figure 10 is shown below.

$$
\begin{array}{|c|c|c|c|}
\hline
P & b1 & R & # \\
\hline
A1 & S & T & # \\
A1 & Y & Z & W \\
# & b1 & Z & W \\
\hline
\end{array}
$$

Entries to FBT from Figure 10.

If a predicate $p$ has only facts then $FBT(p)$ can be computed by knowing the domain of the program. The entries in $FBT(p)$ could then be numerous.

Notation and concept of FBT developed in conjunction with Kundu, [33]. The use of subsumption to keep the size of FBT small is due to Kundu, [33, 34]. Kundu, [34], uses '*' instead of '#'.

†
If there are rules of p, then no part of FBT(p) can be computed at compile time because no negative facts are allowed in Prolog. FBT(p) is initialized to be the empty table. All entries to FBT(p) are generated during execution. As the execution proceeds, more and more failure-bindings for the various predicates are identified and they in turn are used to optimize the subsequent search process. The FBT(p) is void when the program is modified (externally or using the Prolog assert clause).

If \( s = s1(s_1, s_2, \ldots, s_n) \) and \( t = t1(t_1, t_2, \ldots, t_m) \) are two argument tuples, then \( s \) is said to subsume (i.e., be more general than) \( t \) if \( s \) unifies with \( t \) with the substitutions going from \( s \) to \( t \), i.e., substitutions are made in \( s \) to make \( s \) identical to \( t \).

**Example**

The tuple \( s = (a1, b1, Z) \) subsumes the tuple \( t = (a1, b1, c1) \), because by substituting \( Z/c1 \), \( s \) can be made identical to \( t \). However, \( s \) does not subsume the tuple \( q = (X, b1, c1) \).

The tuples in each FBT(p) are maintained as maximal (most general) in the sense that tuple \( t \) is added to FBT(p) only if it is not subsumed by an existing tuple in FBT(p). Furthermore, when a tuple \( t \) is added to FBT(p), all the existing tuples \( s \in \) FBT(p) which are subsumed by \( t \) are deleted.

---

§ Personal communications, Kundu [33].

† This subsumption definition corresponds to conventional definition of subsumption in database theory and other related fields of computer science.
Example

The maximal tuples in $\text{FBT}(q)$ from Figure 10 are shown in Figure 12.

\begin{tabular}{|c|c|c|c|}
\hline
a1 & S & S & # \\
# & bl & Z & W \\
\hline
\end{tabular}

Figure 12.2.: Maximal tuples in FBT
4.4. Forward Execution Based on Failure-Bindings

Forward execution is modified when there are failure-bindings for a literal which is used to obtain the current goal. The forward execution algorithm based on failure-bindings is given in Algorithm A1.

The failure-bindings of a literal are used for intelligent backtracking based on the generator-consumer approach. The generators and consumers of a variable in a rule dynamically change. The next section describes the maintenance of the generators and consumers in a rule in detail.
Phase I:

Let $g = (p', \alpha)$ be a new current goal.

If (goal-list = empty)
then
   Report Success.
   Apply Algorithm B1 (see section 4.6)
elseif (g is subsumed by FBT(p)) or (no more alternatives of p)
then
   Report failure /* g fails */
   Apply Algorithm B1 (see section 4.6)
else
Phase II:

Let $i$ be the next alternative for $p$.

if (no more alternative of $p$)
then
   Report failure /* g fails */
   Apply Algorithm B1
if (alternative $i$ is a fact and if $g$ unifies with $i$)
then
   Apply Algorithm A1 at Phase I
elseif (alternative $i$ is a rule)
then
   if ($g$ fails to unify with the rule-head of $i$)
   then
Apply Algorithm A1 at Phase II

else

let i" be the body of the unified version of rule i.

if (the a subgoal q" in i" is subsumed by a tuple in FBT(q))

then

Apply Algorithm A1 at Phase II

else

Apply Algorithm A1 at Phase I

Algorithm A1: Forward Execution Algorithm based on Failure-bindings.
4.5. Generators and Consumers

Consider a rule $r$. The head of $r$ is denoted $\text{head}(r)$ and its body is denoted $\text{body}(r)$. The generator-consumer list (in short, the gc-list) of a variable $X$ in $r$ is defined as the ordered list of subgoals in $\text{body}(r)$ which contain the variable $X$ in their arguments.\footnote{This section on generators and consumers is largely taken from Kundu, [33 - 34], with adaptation to atomic constants and variables.} The gc-list of a variable $X$ in rule $r$ is denoted $\text{gc}(r, X)$. It is assumed that no rule in the program has alternate paths.\footnote{For example the rule-body "$\vdash g_1, g_2, (g_3; g_4), g_5." has two alternate paths, namely "$g_1, g_2, g_3, g_5."$ and "$g_1, g_2, g_4, g_5."$.} If $c$ is a constant in $\text{body}(r)$, then we let $\text{gc}(r, c) = \text{head}(r)$. This is done to indicate the fact that the presence of $c$ is due to the rule itself.

An instance of a predicate may appear more than once in $r$. Hence, a convenient way to represent $\text{gc}(r, X)$ is by a list of the indices of the subgoals in $\text{body}(r)$, where the subgoals are successively assigned the indices 1, 2, ..., in left-right order.\footnote{The index $i \geq 0$ refers to a subgoal or the head of rule $r$ which has the index $i$.} The indices imply the order of literals in the rule. The index 0 goes to $\text{head}(r)$ and for a constant $c$, the assignment $\text{gc}(r, c) = [0]$ is done to indicate that $c$ in $r$ is explicit. The gc-lists for a rule $r$ are easily determined by a single left to right scan of the rule-body. An application $r'$ of a rule $r$ is obtained from the set of unification substitutions applied to $\text{head}(r)$, when a goal, $g$, unifies with $\text{head}(r)$. The gc-lists of the variables in the unified form $r'$ of $r$ can be constructed from the gc-lists for $r$ as follows.

1. Let $\{X_i \leftarrow e_i\}$ denote the unification mappings from $g$ to $\text{head}(r)$. $X_i$
occurs in head(r) and e is an expression made up of constants and variables, Yj's, in g.

2. For each variable construct
\[ e(X_i) = \{ X_k \mid X_k \text{ occurs in head(r) and body(r) and } \exists Y_j \text{ which appears in } e_i \text{ and in } e_h \}. \]

3. Construct the gc-list of the variables in r from e(X') as follows:
\[ gc(r', X_i) = (\cup gc-list(r, X_k) \mid X_k \in e(X_i)) \]

4. The gc-lists of r are ordered appropriately with duplicate occurrences of indices, if any, removed.

Note that e(X_i) = \emptyset if X_i is substituted by a constant in r'. Consequently, gc(r', X_i) = \{0\}. If c is a constant in body(r'), then we let gc(r', c) = \{0\}. A simpler notation, gc(X), for gc(r, X), is used when the rule application is understood.

Example

Consider the rule r, from Figure 1.:


The gc-lists of r are
\[ gc(r, X) = [2, 4, 7, 9, 11] \quad gc(r, U) = [4, 9, 10] \]
\[ gc(r, Y) = [1, 4, 8, 10] \quad gc(r, V) = [6, 7, 9] \]
\[ gc(r, Z) = [1, 3, 5, 11] \quad gc(r, W) = [5] \]

Now, consider an application r' of r obtained by the unification of the goal g = (p0(X', X'), \{\}) to the head(r). The unification mapping from g to r is given by
\[ \{X \leftarrow X', Y \leftarrow X'\}. \]

and
\[ e(X) = \{X, Y\} \text{ and } e(Y) = \{X, Y\} \]

Therefore, the gc-lists of \( r' \) are

\[ \text{gc}(r', X) = \text{gc}(r, X) \cup \text{gc}(r, Y) = \{1, 2, 4, 7, 8, 9, 10, 11\} \]
\[ \text{gc}(r', Y) = \text{gc}(r, X) \cup \text{gc}(r, Y) = \{1, 2, 4, 7, 8, 9, 10, 11\} \]

For the (dummy) variables \( Z, U, V, \) and \( W \) which appear only in the body of rule \( r \) (and hence only in the body of \( r' \)), their gc-lists remain unaltered, e.g., \( \text{gc}(r, Z) = \text{gc}(r', Z) \).

Consider \( r'' \), which is an application of \( r \) for the goal \( g'' = (p0(X'', Y''), \{Y/y0\}) \). The unification mappings are

\[ \{X \leftarrow X, Y \leftarrow y0\} \]
\[ e(X) = \{X\} \text{ and } e(Y) = \emptyset \]

Therefore, the gc-lists of \( r'' \) are

\[ \text{gc}(r'', X) = \text{gc}(r, X) \]
\[ \text{gc}(r'', Y) = [0] \]

The other gc-lists remain the same as before.

The gc-lists of \( r''' \) which is an application of \( r \) for the goal

\[ (p(X''', Y'''), \{X''''/[x0, y0]\}) \]

are

\[ \text{gc}(r''', X) = [0] \]
\[ \text{gc}(r''', Y) = \text{gc}(r, Y). \]

The gc-lists give an abstract representation of the rule \( r \) in that these lists are not affected by certain changes in permutations and composition of the arguments of a literal in \( r \). For example, if the literal \( p4(X, Y, U) \) in the above example were replaced by \( p4(Y, X, U) \), or by any other literal which uses the variables \( X, Y \) and \( U \), the gc-lists would remain unchanged.
The importance of the gc-lists comes from the fact that they contain all the relevant information about the structure of the rule r for determining intelligent backtracking operations as explained in subsequent sections (and forward jumping, as explained in chapter 7).

A state of computation, θ, in a Prolog execution corresponds to a node in the goal-search tree, or more precisely, the path from the root to that node. The path describes the sequence of successive rule-applications and their associated unification substitutions. Thus, the state of computation gives the current goal, the active set of variables and their instantiations. If the same rule r is used more than once along a path, then they are distinguished by their order of occurrence in the path.

The notion of a generator for a variable X in a particular application r' of rule r and in a computation state θ, the path from the root to a node in the goal-search-tree, is defined as follows:

I Let the subgoals j ≥ 1 in r' be the subgoal that is being investigated in the state θ. The subgoals 1 ≤ i < j in r' have been satisfied in states prior to θ.

II If, for some i, 1 ≤ i < j and i ∈ gc(r', X), X is assigned a value in course of satisfying the subgoal i in r', then i is called the generator of X for r' in the state θ and we write gen(r', θ, X) = i.§

The subgoals m ∈ gc(r', X) and m > gen(r', θ, X) are called the consumers of X

§ The assignments of the form X ← c, where c is a constant value (atomic or structured constant) indicates that X is assigned a value. The discussion for the assignments of the form X ← e can be easily extended to the case when e is a general expression. The expression e could contain new variables which were not in r'. Hence, newer gc-lists appear and computation proceeds as described.
denoted by \( \text{con}(r', \theta, X) \). If \( c \) is a constant in \( \text{body}(r') \), then \( \text{gen}(r', \theta, c) = 0 \) and \( \text{con}(r', \theta, c) = \emptyset \) for all applicable states \( \theta \) in \( r' \).

In general, only some of the variables of \( r' \) will have their generators defined in a state \( \theta \). A simple method for updating the values of \( \text{gen}(r', X) \) as the search for the successive subgoals of \( r' \) is continued is as follows.

1. Initially, for each variable \( X \) and constant in \( \text{body}(r') \) \( \text{gen}(r', X) = 0 \).
2. When a subgoal \( i \) of \( r' \) succeeds, we let \( \text{gen}(r', X) = i \) for each variable \( X \), which is assigned a constant value by \( i \).
3. If \( i \) is the \( \text{gen}(r', X) \), then this is not modified unless backtracking goes to a subgoal \( j \leq i \).

Simpler notations like \( \text{gen}(X) \) or \( \text{gen}(r', X) \) for \( \text{gen}(r', \theta, X) \) and \( \text{con}(X) \) or \( \text{con}(r', X) \) for \( \text{con}(r', \theta, X) \) can be used when an application of a rule and the state of computation is understood.

**Example:**

Consider an application \( r' \) of the rule \( r \), shown in Figure 1, for an application of a goal \( g = (p0(X', Y'), \{Y'/y0\}) \).

\[
\begin{align*}
  r' : p0(X, y0) :&- p1(y0, Z), p2(X), p3(Z), p4(X, y0, U), p5(Z, W), p6(V), \\
\end{align*}
\]

The gc-lists of \( r' \) are

\[
\begin{align*}
  \text{gc}(r', X) &= \text{gc}(r, X) \\
  \text{gc}(r', Y) &= [0].
\end{align*}
\]

The gc-lists of other variables appearing only in \( \text{body}(r') \) remain the same as the gc-lists of the rule \( r \), as in the earlier example. The generators of all the variables and constants in \( r' \) are set to 0.
Consider the computation state $\theta$ where the first eight subgoals in body($r'$) have succeeded and the computation of the subgoal $p_9(X, U, V)$ is not yet complete. More specifically, assume that the state $\theta$ was reached as follows:

1. The subgoal $p_1(y_0, Z)$ succeeded, without generating a value for $Z$.
2. $p_2(X)$ succeeded, with an output assignment $X/x_0$, modifying $\text{gen}(X) = 2$.
3. $p_3(Z)$ has succeeded, generating $Z/z_1$ and modifying $\text{gen}(Z) = 3$.
4. $p_4(x_0, y_0, U)$ succeeded, generating $U/u_0$ and modifying $\text{gen}(U) = 4$.
5. $p_5(z_1, W)$ succeeded, without assigning a value to $W$.
6. $p_6(V)$ succeeded, generating $V/v_0$ and modifying $\text{gen}(V) = 6$.
7. $p_7(x_0, v_0)$ succeeded, without any output assignments.
8. $p_8(y_0)$ succeeded, without any output assignments.

The search for the subgoal $p_9(x_0, u_0, v_0)$ is assumed to be incomplete. Here, $j = 9$.

9. The associated generators and consumers for $r'$ in the state $\theta$ are given by

$$
\begin{align*}
\text{gen}(X) & = 2 & \text{con}(X) & = [4, 7, 9, 11] \\
\text{gen}(y_0) & = 0 & \text{con}(y_0) & = \emptyset \\
\text{gen}(Z) & = 3 & \text{con}(Z) & = [5, 11] \\
\text{gen}(U) & = 4 & \text{con}(U) & = [9, 10] \\
\text{gen}(V) & = 6 & \text{con}(V) & = [7, 10] \\
\text{gen}(W) & = \text{undefined} & \text{con}(W) & = \text{undefined} .
\end{align*}
$$
4.6. Backward Execution Based on Failure-bindings

When a subgoal \( f \) reports failure in state \( \theta \) during execution then backtracking is applied. A previous state, \( \theta' \), in which a goal \( b \) is executed, is selected for backtracking. Any subgoal \( \leq (f-1) \) may be selected for backtracking. Forward execution is started in the state \( b \) and is executed.

The intelligent backtracking schemes described in chapter 3 are based on identifying the most recently generated binding that could cure the failure of a goal \( g \), when \( g \) fails without any solution. However, a better backtrack point can be selected if the failure-bindings of literal \( f \) that failed in the state \( \theta \), are known. It is shown that the earliest of the generators of the failure-bindings can be selected as the backtrack literal.\(^\dagger\) This is based on the observation that every failure-binding of \( f \) has to be rectified before \( f \) can succeed.

\(^\dagger\) The selection of backtrack literal was developed in conjunction with Dr. Kundu. [33, 34]
Example

Consider a goal $g = ((q(X, Y, Z, d_1), (X/a_1, Y/b_1, Z/c_3))$ that fails with the failure-bindings $X/a_1$ and $Y/b_1$ for $q(X, Y, Z, d_1)$. Also assume that $\text{gen}(X) = 3$, $\text{gen}(Y) = 1$ and $\text{gen}(Z) = 4$. Dependency directed backtracking will select the latest of the generators whose solutions were applied to $g$. Hence the backtrack literal selected would be 4.

However, since both $X/a_1$ and $Y/b_1$ have to be rectified to solve $g$ we have to redo literals 3 and 1. Hence, the backtrack literal selection that is based on failure-bindings would select 1 as the backtrack literal. This literal is the closer to the root of the goal-tree than those selected by other methods.

---

Backward Execution Algorithm

If the argument tuple of a goal $g = (p', \alpha)$ is subsumed by any tuple in $\text{FBT}(p)$, then $g$ will fail because a failure-binding is applied to $p'$ to obtain $g$. The tuples that subsume $g$ give failure-bindings of $p'$. This is because every tuple in $\text{FBT}$ corresponds to one failure-binding of the predicate. Algorithm B1 gives the Backward Execution algorithm based on failure-bindings.

Example

Consider the goal $g = (q(X, Y, Z, d_1), (X/a_1, Y/a_1))$. This gives the argument tuple of $g$ as $q(a_1, b_1, Z, d_1)$. Let the tuples $s_1 = q(a_1, S, T, #), s_2 = q(X, b_1, Z, W)$ in $\text{FBT}(q)$ subsume $p(a_1, b_1, Z, d_1)$. The tuple $s_1$ represents $X/a_1$ as a failure-binding of $p$ and the tuple $s_2$ represents $Y/a_2$ as a failure-binding of $p$. Hence, $X/a_1$ and $Y/a_2$ are identified as the failure-bindings for $q(X, Y, Z, d_1)$. 
Let $g = (p', \alpha)$ be the current goal that failed.

If $(g$ is subsumed by $FBT(p))$
then

identify the failure bindings of $p'$ from $FBT(p)$.

else /* $g$ is not subsumed by $FBT(p)$ */

Identify failure-bindings for $p'$ from the alternatives of $p$
Update $FBT(p)$ for every failure-binding in $p''$
if (there are no failure-bindings in $p''$)
then

Backtrack literal is selected according to Kumar

else /* there are failure-bindings of $p''$ */

if ($c \in C_p$ is a failure-binding for $p'$)
then

backtrack literal = 0

else

backtrack literal = min{$\text{gen}(X) \mid X/\alpha$ is failure-binding for $p'$}

Backtrack

Update bit-vector if backtrack literal = 0 (See section 5.7)

Continue Forward Execution.

Algorithm B1: Backward Execution Algorithm Based on Failure-Bindings.
4.7. Bit-vector Implementation to Identify Failure Bindings

The following discussion for identifying the failure-bindings is limited to Prolog program with only two modes for the arguments of the predicates, the Ground state and the Free state. In essence, it is limited to programs without structured terms, though structured constants are allowed. Structured terms lead to nested structures, and the unification process and identification of cause of failure becomes dependent on this process.

Failure-binding can be identified in a fairly straightforward manner as Prolog backtracks from a failure of a subgoal to its parent (= the head of the rule which gave rise to the subgoal), by simply translating the failure-binding (if any) of the subgoals in the rule as causes of failure for the goal using the unification substitution mapping in the reverse direction, and by combining the latter with the current tentative failure-bindings at the parent.

The failure-bindings of all the predicates can be identified by analyzing the program. This is not always necessary. The objective is to prune computation during execution. In other words, the objective is to explore the search space efficiently while being relevant to the query. Hence, it would be ideal to identify the failure-bindings, if any, during the computation, without incurring excessive runtime overhead. An important observation to be kept in mind is that a goal fails only after an attempt to explore the entire search space of the predicate of which the goal is an instance.

For every goal executed, the failure-bindings can be identified by identifying the causes of failure by each of the alternatives from Facts 1, 2, 3, 4 and 5. (see section 4.2)

Consider a goal \( g = (p', \alpha) \). The failure-bindings for \( p' \) are identified as follows.
1. A bit-vector is maintained with \( g \). The length of this bit-vector is \(| C_p \cup \alpha |\), i.e., there is a bit for every constant in \( p' \) and every binding in \( \alpha \).

2. Every bit in this bit-vector is initialized to '1'. Thus each member of \( C \cup \alpha \) is a potential failure-bindings for \( p' \).

3. The causes of failure for \( p' \) by every alternative of \( p \) are identified. The suggestion of failure-bindings for \( p' \) is maintained incrementally, i.e., the common causes of failure for \( p' \) by all the alternatives that have been tried so far are suggested as potential failure-bindings for \( p' \), when the alternative \( m \) is tried.

   For each alternative, the bit corresponding to a binding \( X/v (c) \) in the bit-vector with \( g \) is ANDed with '1' if \( X/v (c) \) is a cause of failure for \( p' \) due to the alternative. Otherwise, the bit is ANDed with a '0'.

4. The common causes of failure for \( p' \) after exhaustion of all the alternatives of \( p \) are identified as the failure-bindings for \( p' \).

   The final bit-vector with \( g \), after exhausting all the alternatives of \( p \), is used to identify the failure-bindings for \( p' \), if any. There is a one-to-one correspondence between every bit in the bit-vector of \( g \) and the elements of \( C_p \cup \alpha \). Every bit which is '1' in the final bit-vector with \( p' \) gives the corresponding element in \( C \cup \alpha \) as a failure-binding in \( p' \). This is obtained from the bits which are '1' in the bit-vector after all the alternatives are exhausted.

   The unification algorithm, which is essentially a pattern matching algorithm in Prolog, quits after the first mismatch. This pattern matching algorithm is modified to identify all the mismatches over the whole argument tuple of the goal \( g \). Each mismatch then gives one cause of failure of \( g \).

   It is to be noted that the extension of the unification algorithm to scan the whole argument tuple of the goal does not hurt the complexity of the unification process, but
can potentially perform more than the standard unification algorithm. In the bargain, more causes of failures of \( p' \) are identified.

Example

Consider the bit-vector modification with unification with facts.

\[
g = t(a_1, b_1, c_1).
\]

<table>
<thead>
<tr>
<th>facts</th>
<th>causes of failure</th>
<th>bit-vector before unification</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t(a_2, b_1, c_2) )</td>
<td>( a_1; c_1 )</td>
<td>( [1, 1, 1] )</td>
</tr>
<tr>
<td>( t(a_2, b_2, c_4) )</td>
<td>( a_1; b_1; c_1 )</td>
<td>( [1, 1, 0] )</td>
</tr>
<tr>
<td>( t(a_2, b_3, c_1) )</td>
<td>( a_1; b_1 )</td>
<td>( [1, 0, 1] )</td>
</tr>
</tbody>
</table>

Figure 13. Unification with facts and Bit-vector manipulation

Figure 13 illustrates the identification of the causes of failure of \( g \) obtained by unification and the refinement of the bit-vector with \( g \).

During backtracking from a subgoal \( j \) in a rule, \( r \), if the selected backtrack literal is 0, then the failure-bindings of \( j \) together with the unification mapping applied to the subgoal \( 0, \) head(\( r \)), in the reverse direction is used to identify the causes of failure of the goal by the rule.
Example:

Consider the goal \( g = (p(X, Y, Z), \{X/[a1, a2], Y/[b1, b2], Z/[c1, c2]\}) \) and the rules

\[
p(X, Y, Z) :- p11(X), p12(Y), p13(Z).
p(M, N, O) :- p31(M, N), p32(N, O), p33(O, M).
\]

The causes of failure of \( g \) by the rules and the corresponding modification to the bit-vector with \( g \) are shown in Figure 14.

<table>
<thead>
<tr>
<th>bit-vector</th>
<th>failure-binding</th>
<th>subgoals in rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 1, 1]</td>
<td>( X/[a1, a2] ) for ( p11(X) )</td>
<td></td>
</tr>
<tr>
<td>[1, 0, 0]</td>
<td>( S/a1 ) for ( p24(S, T, U) )</td>
<td></td>
</tr>
<tr>
<td>[1, 0, 0]</td>
<td>( M/[a1, a2] ) for ( p33(O, M) )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 14. Bit-vector manipulation with application of rules

Example

Consider the goal \( (\text{query}(X, Y), \{X/a1, Y/b1\}) \) and the following program. The identification of the failure-bindings of \( \text{query}(X, Y) \) by the facts and rules of the program is shown in Figure 15.

\[
q1(a1).
q2(b2).
q2(b3).
q3(a1, c1).
q3(a2, c1).
q4(c1, b2).
\text{query}(a1, b2).
\text{query}(X, Y) :- q1(X), q2(Y).
\text{query}(X, Y) :- q3(X, Z), q4(Z, Y).
\]
In Figure 15 computation starts with a search for a solution of to the goal query(a1, b1) with the initialized bit-vector [1, 1]. The first fact indicates that Y/b1 is a cause of failure for query(a1, b1), modifying the bit-vector [0, 1]. On application of the next rule q1(a1) succeeds and q2(b1) fails. The failure of q2(b1) identifies Y/b1 as a failure-binding for q2(Y). This eliminates all computation within the first rule applicable to the query, i.e., q1(X) is not tried any more, and backtracking goes to the initial goal, the query(a1, b1), using the gc-list of Y. The bit-vector at the query(a1, b1) is modified to [0, 1]. The next applicable rule translates itself into q3(a1, Z) and q4(Z, b1). q3(a1, Z) succeeds with a binding Z/c1. q4(c1, b1) fails, identifying Y/b1, obtained during unification to the rule-head, as a failure-binding for q4(Z, Y). This suggests Y/b1 as a cause of failure for query(a1, b1) by this rule. The bit-vector of query(a1, b1) is modified [0, 1]. There are no more alternatives for query(a1, b1), so the final bit-vector [0, 1], which indicates Y/b1 as a failure-binding for query(X, Y).

---

<table>
<thead>
<tr>
<th>goal</th>
<th>applicable fact or rule</th>
<th>initial bit vector</th>
<th>resultant bit vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>query(a1, b1).</td>
<td>query(a1, b2).</td>
<td>[1, 1]</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>q1(a1)</td>
<td>query(X, Y) :- q1(X), q2(Y).</td>
<td>[0, 1]</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>q2(b1).</td>
<td>q1(a1).</td>
<td>[1]</td>
<td>[0]</td>
</tr>
<tr>
<td></td>
<td>all facts and rules indicate failure due to Y/b1.</td>
<td>[1]</td>
<td>[1]</td>
</tr>
<tr>
<td></td>
<td>(Intelligent backtracking takes to goal being tried.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>query(a1, b1).</td>
<td>query(X, Y) :- q3(X, Z), q4(Z, Y).</td>
<td>[0, 1]</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>q3(a1, Z).</td>
<td>q3(a1, c1).</td>
<td>[1, 0]</td>
<td>[0, 0]</td>
</tr>
<tr>
<td>q4(c1, b1).</td>
<td>all alternatives fail for q4 indicating Y/b1 as the cause of failure.</td>
<td>[1, 1]</td>
<td>[0, 1]</td>
</tr>
<tr>
<td></td>
<td>(Intelligent backtracking takes us the goal being tried)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>query(a1, b1).</td>
<td>no more alternatives</td>
<td>[0, 1]</td>
<td>[0, 1]</td>
</tr>
</tbody>
</table>

Figure 15. Identification of Failure-binding in the presence of facts and rules.
4.8. Incompleteness of the Identification of Failure-Bindings

An interesting point to be noted here is that if we focus our attention only on the part of the goal-tree for the goal g which is visited by Prolog, then we may miss identifying some failure bindings. The identification of failure-bindings by Prolog, using the bit-vector implementation, is not complete, i.e., every failure-binding of a goal is not identified when the goal fails. The reasons for the incompleteness are two fold. They are

1. Left to right control of execution of subgoals in a rule. This is due to incomplete gathering of information about the causes of failure for the goal by the applicable rules for the goal. The failure of a subgoal may invalidate the application of a rule. Subsequent subgoals are not tried. The subsequent subgoals may indicate additional causes of failure for the goal.

2. Even with an extended unification algorithm, some failure-bindings can be missed by the bit-vector implementation algorithm. This happens if the unification fails and does not identify any cause of failure. This is illustrated in Figure 16.†

The following is true using the Prolog control structure. Every alternative can indicate many bindings as causes of failure by the alternative. As a rule body is executed left-to-right, all the failure-bindings of all the failing subgoals in the rule may not identified. The first subgoal that fails and leads to backtracking to the parent, i.e., head of the rule, leads to preventing further execution of the body of the rule. This results in incomplete information about the causes of failure by the rule, because the failure-bindings of the subgoals that are not executed may identify causes of failure of

† The failure of unification without identifying a cause of failure was singularly pointed out by Kundu [33]
the goal by the alternative. Since, no assumption can be made about the causes of failure of the goal by the alternative, only the identified causes of failure are used to identify failure-bindings.

What this means is that if we compute the failure-bindings only on the basis of the searches performed by Prolog, then certain failure-bindings may be missed, though no erroneous failure-bindings will be reported. This will partially reduce the effectiveness of the new backtracking method, in that it will be reduced more often than necessary to ordinary intelligent backtracking.

Bits in the bit-vector are ANDed with '1' only if the corresponding binding, or constant, is a cause of failure. Otherwise they are ANDed with '0'. The incomplete information by an application of a rule could result in the improper ANDing of '0' to some of the potential causes of failure. This could result in not identifying possible failure-bindings, as shown in Figure 16.

Example

In Figure 11 we cannot identify Y/b1 as the failure-binding for r22(Z, Y) if the fact r21(a1, c1) is removed. The first fact for r identifies Y/b1 as the cause of failure for r(X, Y). The first rule for r identifies Y/b1 as a cause of failure of r". The left-to-right processing of subgoals in a rule-body of the second rule by Prolog causes r21(X, Z) to fail, reporting X/a1 to be the failure-binding for r21(X, Z), which in turn translates to X/a1 a cause of failure for r(X, Y). This is not supported by the first fact or the first rule for r. The goal r22(Z, Y) is never tried, and hence Y/b1 which is a failure binding for r22(Z, Y) is never identified, to identify Y/b1 as a a cause of failure for r(X, Y). The second rule does not support Y/b1 as a cause of failure of r". One then erroneously concludes that r(X, Y, Z) does not have a failure-binding.
Example

Consider the goal (query(X, Y), (X/a2, Y/b1)) and the program in Figure 15. Figure 16 shows the incompleteness of the identification of failure-bindings.

<table>
<thead>
<tr>
<th>goal</th>
<th>applicable fact or rule</th>
<th>initial bit vector</th>
<th>resultant bit vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>query(a2, b1)</td>
<td>query(a2, b1)</td>
<td>[1, 1]</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>q1(a2)</td>
<td>query(X, Y) :- q1(X), q2(Y). X/a2 is failure-binding for q1(X) backtrack to query(a2, b1)</td>
<td>[1, 1]</td>
<td>[1, 0]</td>
</tr>
<tr>
<td>q3(a2, Z)</td>
<td>query(X, Y) :- q3(X, Z), q4(Z, Y). q3(a1, c1). Y/b1 is failure-binding for q4(Z, Y) backtrack to query(a2, b1)</td>
<td>[1, 0]</td>
<td>[0, 0]</td>
</tr>
<tr>
<td>q4(c1, b1)</td>
<td>no more alternatives</td>
<td>[1, 1]</td>
<td>[0, 1]</td>
</tr>
</tbody>
</table>

Figure 16. Incompleteness of Identification of Failure-bindings by rules.

Figure 16 shows the bit-vector method to identify the failure-bindings of (query(X, Y), (X/a2, Y/b1)). Computation of the goal query(a1, b1) proceeds as follows:

1. The first fact modifies the bit-vector to be [1, 1].

2. The first rule is applied, and q1(a2) is tried first. q1(a2) fails and indicates X/a2 as a failure-binding for q1(X). The first rule application fails, the goal q2(b1) is not executed. The bit corresponding to Y/b1 in the bit-vector with the query(a2, b1) is ANDed with '0', because Y/b1 is not identified as a cause of failure by the first rule. The resulting bit-vector with the goal is [1, 0] (the bit corresponding to X/a2 is ANDed with '1').

3. The next applicable rule translates itself into q3(a2, Z) and q4(Z, b1). q3(X, Z) succeeds generating the binding Z/c1. q4(c1, b1) fails and identifies Y/b1 as a failure binding for q4(Z, Y). This modifies the bit-vector with the query(a2, b1) to [0, 0]. ANDing the bit corresponding to Y/b1 with '1' and the others with '0's.
4. There are no more alternatives applicable to the query. The query(a2, b1) fails and no failure-binding for query(X, Y) is identified.

According to Theorem 4.2 and Figure 15, Y/b1 is a failure-binding for query(X, Y). The information that Y/b1 is a cause of failure of query(a2, b1) is not substantiated by the first rule during the execution, because the subgoal q2(Y) was not executed when the subgoal q1(X) fails. Hence, the incompleteness prevails in the identification of failure-bindings.

Example

The incompleteness of identification of failure-binding is also due to unification failure without identifying any cause of failure. Consider the goal (p(X, Y, Y), (X/a, Y/b)). Figure 17 shows the unification with the facts for p.

<table>
<thead>
<tr>
<th>facts</th>
<th>initial bit-vector</th>
<th>final bit-vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(a1, b1, b2).</td>
<td>[1, 1]</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>p(a, b1, b2).</td>
<td>[1, 1]</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>p(a1, b, b).</td>
<td>[0, 1]</td>
<td>[0, 0]</td>
</tr>
</tbody>
</table>

Figure 17. Incompleteness of identification of failure-binding by unification.

Consider that the goal (p(X, Y, Y), (X/a, Y/b)) as defined only by the facts in Figure 17. The unification process and the subsequent bit-vector does not identify any failure-binding for p(X, Y, Y). It is easy to observe that the goal (p(X, Y, Y), (X, Y), ( ), (X/a)) fails and, as per the Definition 1, X/a is a failure-binding of p(X, Y, Y).

The unification algorithm identifies mismatches as causes of failure. In the above example, the failure in unification in the different facts are due to different bindings. The failure-binding in the goal (p(X, Y, Y), (X/a)) is due to the interaction of X and Y in different facts. The unification algorithm is oblivious to such situations.
Example

The following example illustrates recursive intelligent backtracking based on failure-bindings across clauses.

query(X, Y) :- q1(X, Y), q2(Y, Z), q3(X, Z).
q1(X, Y) :- ..... q2(Y, Z) :- ..... q3(X, Y) :- q31(X, Z), q32(Z, Y), q33(X, Y).
q31(X, Y) :- ..... q32(X, Y) :- ..... q33(X, Y) :- q331(X, Z), q332(Y, Z), q333(X, Y).
q331(X, Y) :- ..... q332(X, Y) :- ..... q333(X, Y) :- q3331(X, Y), q3332(X, Y).
Suppose, the goal (query(X, Y), \{X, Y\}, \{\}, \{X/a1\}) is tried. Also, assume that
the literals, obtained after unification and substitution. q1, q2, q3, q31, q32, q33, q331,
q332, q333, q3331, q3332 have succeeded and the literal q333 fails with X/a1 as a
failure-binding. Also, assume that the literals q333, q33 and q3 have exhausted their
alternatives. Then, gen(X) in the rules "q33(..) :- .....", "q33(..) :- ...", "q3(..) :- ....."
and "query(..) :- ....." would be 0. Hence, backtracking is recursively applied from
q333 to q33 to q3 to query, since q333, q33 and q3 have exhausted their alternatives.
Applying the techniques used in WAM, it is easy to implement intelligent backtracking
based on failure-bindings and can backtrack across clauses.

Figure 18. shows the complete goal search tree explored by intelligent backtracking
based on failure-bindings, for the example in Figure 1. Table III gives the trace of
execution by intelligent backtracking based on failure-bindings for the example in
Figure 1.
Figure 18.
Goal-tree by Intelligent Backtracking based on failure-bindings
<table>
<thead>
<tr>
<th>goal</th>
<th>success/fail</th>
<th>binding</th>
<th>failure-binding</th>
<th>backtrack literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1(Y, Z), {})</td>
<td>success</td>
<td>Y/b1; Z/c1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p3(Z), {Z/c1})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a1, Y/b1})</td>
<td>fail</td>
<td>X/a1</td>
<td></td>
<td>p2</td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p3(Z), {Z/c1})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a2, Y/b1})</td>
<td>success</td>
<td>U/d1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p5(Z, X, U, W), {Z/c1, X/a2, U/d1})</td>
<td>fail</td>
<td>Z/c1</td>
<td></td>
<td>p1</td>
</tr>
<tr>
<td>(p1(Y, Z), {})</td>
<td>success</td>
<td>Y/b1; Z/c1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p3(Z), {Z/c1})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a1, Y/b1})</td>
<td>fail</td>
<td>X/a1</td>
<td></td>
<td>p2</td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p3(Z), {Z/c1})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a2, Y/b1})</td>
<td>success</td>
<td>U/d1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p5(Z, X, U, W), {Z/c1, X/a2, U/d1})</td>
<td>fail</td>
<td>Z/c1</td>
<td></td>
<td>p1</td>
</tr>
<tr>
<td>(p1(Y, Z), {})</td>
<td>success</td>
<td>Y/b2; Z/c2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p3(Z), {Z/c2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a1, Y/b2})</td>
<td>fail</td>
<td>X/a1</td>
<td></td>
<td>p2</td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p3(Z), {Z/c2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a2, Y/b2})</td>
<td>success</td>
<td>U/d1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p5(Z, X, U, W), {Z/c2, X/a2, U/d1})</td>
<td>success</td>
<td>W/c1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p6(V), {})</td>
<td>success</td>
<td>V/f1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p7(X, V), {X/a2, V/f1})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p8(Y), {Y/b2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p9(X, U, V), {X/a2, U/d1, V/f1})</td>
<td>fail</td>
<td>V/f1</td>
<td></td>
<td>p6</td>
</tr>
<tr>
<td>(p6(V), {})</td>
<td>success</td>
<td>V/f2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p7(X, V), {X/a2, V/f2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p8(Y), {Y/b2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p9(X, U, V), {X/a2, U/d1, V/f2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p10(Y, U), {Y/b2, U/d1})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(p11(X, Z), {X/a2, Z/c2})</td>
<td>success</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table III: Trace of Execution by Intelligent Backtracking based on failure-bindings

† Identified from FBT(p4) search.
Theorem 4.3.: The computation of a failure-binding by the bit-vector implementation method is sound.

Proof: On failure of a goal by an alternative, a bit in its bit-vector is ANDed with a '1', if the binding, or constant, corresponding to the bit is identified as a cause of failure by the alternative. The alternative will not satisfy any goal with this cause of failure in the argument of the goal. Thus, a bit in the bit-vector is '1' if and only if the binding corresponding to the bit has been identified as a cause of failure by every alternative that has been tried so far. Consequently, if a bit is '1' after exhausting each of the alternatives then the binding corresponding to the bit has been a cause of failure by all alternatives, and, hence, is a failure-binding. A binding or a constant is identified as a failure-binding, iff the binding or the constant is identified as a cause of failure by each alternative.

If a binding, or constant, is not identified as a cause of failure by an alternative, then the bit corresponding to the binding (constant) is ANDed with a '0'. Consequently, if a bit is '0' after exhausting all the alternatives, then the binding corresponding to the bit is not a cause of failure for each of the alternatives, and, hence, is not identified as a failure-binding for the goal.

Due to the fact that '1' AND '0' = '0' once a bit is ANDed with a '0', it remains '0'. Thus, the binding corresponding to this bit is not identified as a failure-binding. Hence, the identification of failure-binding by the bit-vector implementation is sound.
4.9. Soundness and Completeness of $\xi$

In this section we prove that the search based on failure-bindings, $\xi$, is sound, complete and that it preserves the order of the solutions obtained by Prolog.

A method is sound if every answer that is derived by a method is a valid (correct) one. A method is complete if the method derives every solution. No solution, valid or invalid, should go undervived.

The basic execution mechanism is the same as in Prolog, Intelligent backtracking is a special case of backtracking based on failure-bindings of the current literal.

**Theorem 4.4.** The algorithm for selection of backtrack literal is optimal.

**Proof:** If there are no failure-bindings for the failed literal, then common dependency-directed backtracking is applied, according to [17 - 27]. If there are failure-bindings for the failed literal then the earliest of the generators of the failure-bindings is selected as the backtrack literal. Every failure-binding is a definite cause of failure for the literal, which needs to be rectified. If a literal which is not the earliest of the generators of the failure-bindings for the failed literal is selected as the backtrack literal then the computation would arrive, after backtracking and forward execution, at the failed literal with at least one of its unrectified failure-binding. Hence, this literal will fail again. This will occur until all the failure-bindings of the failed literal are corrected, called late detection of failure. Hence, backtracking to any other literal which is not the earliest of the generators of failure-bindings for a failed literal would not be of any consequence to the failed literal. Thus, the selection of the earliest of the generators of the failure-bindings of the failed as the backtrack literal is most efficient and hence optimal.
Theorem 4.5.: \( \zeta \) is sound.

Proof: A solution to a goal is reported if the goal unifies with a fact, in which case the goal is true, or if all the subgoals in a rule applicable to the goal are true, implying that the goal is true. No other solution of a goal is reported. Hence \( \zeta \) is sound.

Theorem 4.6.: \( \zeta \) is complete.

Proof: The search defaults to ordinary dependency-directed backtracking when there are no failure-bindings of a goal. Intelligent backtracking is applied when there are failure-bindings for the current literal that failed. Backtracking to an intermediate literal, \( b \), is equivalent to jumping to \( b \) in the goal-search-tree which generated a failure-binding. Resuming computation at \( b \) continues search in the next right subtree at \( b \), discarding the subtree of \( b \) which uses a failure binding of the currently failed literal. This subtree does not contain any solution to the initial query. Hence, discarding further search in this subtree does not miss any solution.

Since the identification of failure-bindings is sound, and since the selection of a backtrack literal based on failure-bindings is correct, no subtree which could possibly contain a solution is pruned. Thus no solution that could be derived by Prolog is missed. Hence, \( \zeta \) is complete, within the closed world assumption.  

Theorem 4.7.: The order of solution derived by \( \zeta \) is the same as those obtained by Prolog.

Proof: The proof is trivial for naive backtracking, as in standard Prolog, and dependency-directed backtracking. The control mechanism is identical to that by Prolog and \( \zeta \) is sound and complete. Hence, \( \zeta \) preserves the order of the results.
Chapter 5

5. Intelligent Forward Execution Based on Failure-Solutions

5.1. Introduction

The notion of a failure-solution of a goal is introduced in this chapter. A simple method to identify the failure-solutions of a goal is presented. A systematic method to associate the failure-solutions of a goal in a failure solution table (FST) is illustrated. Intelligent forward execution is based on the failure-solutions of a goal. The savings in computation by intelligent forward execution is compared to the standard search by Prolog. The completeness and soundness properties of the search method based on failure-solutions, \( \zeta' \), is proved. It is also proved that \( \zeta' \) preserves the order of solutions obtained by Prolog.

Exhaustive and thorough search of the goal-tree by Prolog has a shortcoming in that computations of the same successes and same failures can be repeated. One of the reason why standard, dependency-directed or intelligent backtracking may not be able to cure a failure is that the backtrack literal can have many repeated identical solutions which could result in repeated failures.

Consider the execution of the body of a rule "\( \vdash \ldots p(X, Y), \ldots, q(X, Y) \ldots \)". The bindings generated by \( p(X, Y) \) must satisfy \( q(X, Y) \). Let us assume that a solution, \( \Delta \), of \( p(X, Y) \) does not satisfy \( q(X, Y) \). This also means that \( \Delta \) fails to satisfy the goal that unified with the rule-head of the rule in which \( p(X, Y) \) and \( q(X, Y) \) appear, because the rule-application will not succeed with the solution \( \Delta \) of \( p(X, Y) \). If \( \Delta' \) is another solution of \( p(X, Y) \) and \( \Delta' = \Delta \), or if \( \Delta' \) is weaker than \( \Delta \) (\( \Delta \) subsumes \( \Delta' \)),\(^1\)

\(^1\) The weakness property based on subsumption definition was developed in conjunction with Kundu, [33]. Kundu also suggested maintaining a table of successes for each predicate \( p \) (similar to FBT(\( p \))). This can be used to avoid search success-tree of a subgoal if it had been executed be-
then the rule application will fail again. However, the computation encountered in an intelligent forward execution from $p(X, Y)$ onwards with $\Delta'$ can be safely avoided.
5.2. Failure-Solutions

Semantically, a goal $g = (p', \alpha)$ is true (that is, satisfiable, or logically follows from the program) if and only if there is a clause instance $I$ of of a clause $C$, in the program, such that the head of $I$ is identical to $p'$ and all the goals in the body of $I$ are true. By the procedural interpretation of a logic program, a goal $g$ is true if and only if

a) there is a fact $F$ that unifies with $p'$ or

b) there is a rule $H :: B_1, B_2, ..., B_n$ such that $p'$ unifies with $H$ and $B_1, B_2, ..., B_n$ are true.

The subgoals in a rule are executed left-to-right. The bindings that are applicable at a node in the goal-tree of $g$ are the solutions of all the subgoals which are above it in the goal-tree of $g$.

Prolog execution has two phases.

1) During forward execution, goals are unified with facts or rule-heads. The goal is true when it unifies with a fact. When the goal unifies with a rule head, the goal is replaced by the unified version of the rule-body. The goal is true when all the subgoals in this rule-body are true. The subgoals are executed left-to-right. Alternatives that are applicable to a goal are selected top-to-bottom from the textual description of the program.

2) When a goal fails, a backtrack literal is selected by the backward execution algorithm and execution at this literal starts again for additional solutions.

Let $g$ be the primary goal. Along the path to the solution of $g$ many subgoals are tried. Let $n = (q', \beta)$ be a subgoal in the current path to a solution to $g$. Let $n$ be executed in the state $\sigma$. The successful computation of $n$ gives a set of newer bindings, $\Delta$, as its solution. The state of computation is modified from $\sigma$ to $\sigma'$. $\Delta$ can possibly be empty; and $n$ may fail in the state $\sigma$, in which case $\Delta$ is necessarily empty. Note that the computation of $n$ in the state $\sigma$, can have other solutions, i.e., $\Delta''$'s with $\Delta' = \Delta$ or
\[ \Delta' \neq \Delta. \]

The computation of \( n \) in the state \( \sigma \) is denoted by \( \sigma_n \). The goal-tree of \( g \) is denoted \( A^g \). The subtree of \( A^g \) which is below the computation of the subgoal \( \sigma_n \) is denoted \( A^g_{\sigma_n} \).

**Definition 1:** The solution \( \sigma_n \) is a failure-solution of \( n \) for \( g \) if there is no solution for \( g \) in the subtree \( A^g_{\sigma_n} \).

**Example**

Refer to In Figure 1 there is no solution for \( p0(X, Y) \) in the subtree under \( p1(Y, Z) \) with the solution \{\( Y/b1; Z/c1 \)\}. Hence, \{\( Y/b1, Z/c1 \)\} is a failure solution of \( p1(Y, Z) \).

Forward execution will fail to find any solution of \( g \) if a new solution, \( \Delta' \), of subgoal \( n \) is identical or weaker than a previously detected failure-solution, \( \Delta \), of \( n \) for \( g \). It is then proper to skip the redundant search represented by the search space \( A^g_{\sigma_n} \).

A different solution of \( n \) is required for solving \( g \).

**Example**

In Figure 1, on backtracking to \( p1(Y, Z) \) a new solution \{\( Y/b1, Z/c1 \)\} is obtained. This solution is identical to an earlier solution which failed to produce a solution for \( p0(X, Y) \). Obviously, this solution will fail during forward execution, as can be observed from the goal-search tree that has been explored earlier.
5.3. Failure Solution Table

Definition 2: A Failure Solution Table (FST) of \( q'' = (q', \alpha) \) which is executed in the state \( \sigma \) and which is denoted FST\( (q'', \sigma) \), is a table containing the failure-solutions of \( q'' \) in the state \( \sigma \). Every tuple in the FST\( (q'', \sigma) \) is the set of argument tuples of the goal \( (q'', \alpha \cup \Delta) \), where \( \Delta \) is a failure-solution of \( q'' \) in the state \( \sigma \).

The operations that are required to maintain the FST\( (q'', \sigma) \) are:

1. `create(FST(q'', \sigma))` - creates an empty table
2. `find(FST(q'', \sigma), \Delta)` - finds if \( \Delta \) is identical or subsumed by a tuple in FST\( (q'', \sigma) \)
3. `add(FST(q'', \sigma), \Delta)` - adds the argument tuple \( (q', \alpha \cup \Delta) \) to FST\( (q'', \sigma) \)
4. `destroy(FST(q'', \sigma))` - deletes FST\( (q'', \sigma) \).

An FST is associated with every node in the current path in the goal-tree of \( g \), the primary goal. As newer subgoals are invoked, their FST’s are created. FST’s are initialized empty at the first call to a subgoal. When a subgoal exhausts its solutions or backtracking takes to a node above the subgoal in the goal-search-tree, the FST of the subgoal is destroyed. When a subgoal is selected as the backtrack literal by the backtracking algorithm, then the subgoal may add a tuple to its FST, if the latest solution of the subgoal is a failure-solution. When a solution \( \Delta \) to a subgoal \( q'' \) is obtained, the FST\( (q'', \sigma) \) is searched, to find, if the tuple \( (q', \alpha \cup \Delta) \) is subsumed by the existing tuples in FST\( (q'', \sigma) \). The entries in FST\( (q'', \sigma) \) are not applicable for \( q'' \) executed in another state \( \sigma' \neq \sigma \). Therefore, on backtracking to the left of \( q'' \) the FST\( (q'', \sigma) \) is made void (destroyed).
5.4. Intelligent Forward Execution Based on Failure-Solution

FST(q", σ) is used in intelligent forward execution from q". When q" succeeds with a solution Δ, then find(FST(q", σ), Δ) is done to find if Δ is a previously identified failure-solution. If Δ is identical or weaker than any tuple in FST(q", σ), then forward execution from q" is aborted and a new solution for q" is requested. Otherwise, forward execution continues as usual.

If the argument tuple of the goal (q', α ∪ Δ), is subsumed by any existing tuple in FST(q", σ) then forward execution from q" is aborted and q" is further executed to find more solutions of q". A flag is maintained with every goal that is active. This flag is initialized to False when the goal is first invoked. Algorithm A2 gives the forward execution algorithm based on failure-solutions.
Let $q'' = (q', \alpha)$ be the current goal, executed in state $\sigma$.

**Phase 1**

\[
FST(q'', \sigma) = \emptyset; \\
\text{flag}(q'', \sigma) = false;
\]

**Phase 2:**

\[
solved = \text{exec}(q''); /* \text{exec} \text{ is a function to execute a goal */}
\]

if (solved = true)

then

\[
\text{found} = \text{find}(FST(q'', \sigma), \Delta); /* \Delta \text{ is the new solution of } q'' */
\]

if (found = true)

then

Apply Algorithm A2 at Phase 2 to $q''$.

Else

\[
\text{next-goal} = \text{fetches(goal-list)} /* \text{fetches a goal from goal-list */}
\]

if (next-goal = $\emptyset$) /* goal-list is empty */

then

report success

\[
\text{flag}(q'', \sigma) = true;
\]

Apply Algorithm A2 at Phase 2 to $q''$.

else

Apply Algorithm B2 at Phase 1 to next-goal.

else

Apply Algorithm B2 to $q''$.

---

**Algorithm A2:** Forward Execution Algorithm based on Failure-Solutions.
5.5. Backward Execution Based on Failure-Solution

On backtracking to \( q'' \) from \( r'' \), the flag with \( r'' \) is returned to \( q'' \). The value of the flag returned identifies the failure-solution of the \( q'' \). If no solution for the primary goal \( g \) is found in the interim of forward execution from \( q'' \) and backtracking to \( q'' \), then an addition corresponding to \( \Delta \) is done to \( \text{FST}(q'', \sigma) \). The absence or presence of a solution of \( g \) with the present solution of \( q'' \), \( \Delta \), is determined by the flag returned by the backtracking algorithm. If the flag returned to \( q'' \) is True, then the \( \text{flag}(q'', \sigma) \) is set to true. If the flag returned to \( q'' \) is False then \( \Delta \) is a failure solution of \( q'' \) and the tuple corresponding to \( q'' \) is added to \( \text{FST}(q'', \sigma) \). On backtracking from \( q'' \), to \( b'' \), the flag \( \text{flag}(q'', \sigma) \) is returned, to \( b'' \). Algorithm B2 gives the backward execution algorithm based on Failure Solutions.

Let \( q'' \) be the current goal that failed.
Let \( b'' = (b', \sigma') \) be the selected backtrack literal.

```plaintext
if (flag(q'', \sigma) = False)
then

\[ \text{FST}(b'', \sigma') = \text{add}(\text{FST}(b'', \sigma'), \Delta') /\* \Delta' \text{ is the previous soln. of } b'' */ \]

else

\[ \text{flag}(b'', \sigma') = \text{True}; \]
Backtrack to \( b'' \);
Apply Algorithm A2 at Phase 2 to \( b'' \).
```

Algorithm B2: Backward Execution Algorithm based on Failure-Solutions.

Data structures optimal for searching and insertion are ideal for maintaining \( \text{FST}'s \). Data structures such as AVL-trees or B-trees are quite efficient for \text{add} and \text{find} operations.
The goal-tree for the execution of the example of Figure 1 by intelligent forward execution is shown in Figure 19. The trace of execution based on use of FST is given in Table IV.
Figure 19. Goal-search tree explored by Intelligent Forward Execution
<table>
<thead>
<tr>
<th>goal</th>
<th>success/fail</th>
<th>binding</th>
<th>e FST ??</th>
<th>backtrack literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1(Y, Z), [])</td>
<td>success</td>
<td>Y/b1; Z/c1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p2(X), [])</td>
<td>success</td>
<td>X/a1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), [Z/c1])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p4(X, Y, U, [X/a1, Y/b1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p3(Z), [Z/c1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p2</td>
</tr>
<tr>
<td>(p2(X), [])</td>
<td>success</td>
<td>X/a2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), [Z/c1])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p4(X, Y, U, [X/a2, Y/b1])</td>
<td>success</td>
<td>U/d1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p5(Z, X, U, W, [Z/c1, X/a2, U/d1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p4</td>
</tr>
<tr>
<td>(p4(X, Y, U, [X/a2, Y/b1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p3(Z), [Z/c1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p1</td>
</tr>
<tr>
<td>(p1(Y, Z), [])</td>
<td>success</td>
<td>Y/b1; Z/c1</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>(p1(Y, Z), [])</td>
<td>success</td>
<td>Y/b2; Z/c2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p2(X), [])</td>
<td>success</td>
<td>X/a1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), [Z/c2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p4(X, Y, U, [X/a1, Y/b2])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p3(Z), [Z/c2])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p2</td>
</tr>
<tr>
<td>(p2(X), [])</td>
<td>success</td>
<td>X/a2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), [Z/c2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p4(X, Y, U, [X/a2, Y/b2])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p3</td>
</tr>
<tr>
<td>(p5(Z, X, U, W, [Z/c2, X/a2, U/d1])</td>
<td>success</td>
<td>W/e1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p6(V), [])</td>
<td>success</td>
<td>V/f1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p7(X, V), [X/a2, V/f1])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p8(Y), [Y/b2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p9(X, U, V), [X/a2, U/d1, V/f1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p8</td>
</tr>
<tr>
<td>(p8(Y), [Y/b2])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p7</td>
</tr>
<tr>
<td>(p7(X, V), [X/a2, V/f1])</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>p6</td>
</tr>
<tr>
<td>(p6(V), [])</td>
<td>success</td>
<td>V/f2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p7(X, V), [X/a2, V/f2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p8(Y), [Y/b2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p9(X, U, V), [X/a2, U/d1, V/f2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p10(Y, U), [Y/b2, U/d1])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p11(X, Z), [X/a2, Z/c2])</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table IV: Trace of Execution by Intelligent Forward Execution
5.6. Soundness and Completeness of $\varsigma'$

It is proved that the search based on failure-solutions, $\varsigma'$, is sound. It is also shown that $\varsigma'$ preserves the order of solutions obtained by Prolog.

**Theorem 5.1.** $\varsigma'$ is sound.

**Proof:** A solution to a goal $g$ is reported if $g$ unifies with a fact or if all the subgoals in a rule applicable to $g$ is true. Hence, $\varsigma'$ is sound.

**Theorem 5.2.** $\varsigma'$ is complete.

**Proof:** If a solution $\Delta$ of a subgoal, $n$, executed in state $\sigma$, is subsumed by $a$ in $\text{FST}(n, \sigma)$, then forward execution from $n$ is avoided. This indicates that $\Delta$ is identical to or weaker than a previously determined failure-solution of $n$. It can then safely be said that the subtree $\Lambda^\Delta_\sigma$ does not have a solution for $g$. Hence, pruning the search of subtree $\Lambda^\Delta_\sigma$ does not miss any solution. The search of a subtree of the goal-tree $g$ is pruned, by aborting forward execution, if and only if the subtree does not have a solution for $g$. This ensures the completeness of search, by $\varsigma'$.

**Theorem 5.3.** $\varsigma'$ preserves the order of results as obtained by Prolog.

**Proof:** No solution is missed and every solution obtained by Prolog is obtained. The control is the same as in Prolog, i.e., left-to-right within a rule and top-to-bottom selection of alternatives. Hence, $\varsigma'$ preserves the order of the result.

**Theorem 5.4.** No two states in the goal-tree by $\varsigma'$ of a goal $g$ which fails are identical.

**Proof:** A state in a goal-tree corresponds to the path in the goal-search tree. Thus, every node in a goal-search tree corresponds to a state. Consider a goal $g$ executed in the state $\sigma$. Let $\Delta$ be a solution of $g$. Also assume that $\Delta$ is a failure-solution of $g$. If another solution of $g$, $\Delta' = \Delta$, is obtained, then forward
execution from $g$ is aborted. This is done for all failure-solutions of $g$. This means that computation from $g$ onwards does not encounter two identical states which lead to failure. If $g$ does not have any solution then this is applicable to the whole goal-tree of $g$. As a corollary to Theorem 5.4, it can be stated that "there are no two identical paths in the goal-tree by $\zeta'$ of a goal $g$ which fail".
5.7. Difficulty in Implementation of $\zeta'$

It should be noted that association of $\text{FST}(q'', \sigma)$ with $q''$ is not possible on a linear stack of goals - because of the stacking of records in the chronological order of execution. A goal can be solved by a rule whose subgoals may be solved by different rule applications and so on. This leads to the record $q''$ being buried deep in the local-stack. Due to the simplicity of implementation of stack, the address of this record is not known. Hence, the search space and solution of a goal cannot be associated with it. A new data structure which is introduced in chapter 7 is needed to effect intelligent forward execution.
6. Intelligent Forward Jumping

6.1. Introduction

The notion of forward jumping for sequential execution of Prolog is addressed and a scheme to avoid computation of subgoals independent of the corrective actions taken by backtracking is presented. These independent subgoals fall in the range of the failed subgoal, f, and the backtrack subgoal, b, in a rule. The goals are identified from the subgoal dependencies in a rule together with the generators and consumers of the variables in the rule. The resulting search based on forward jumping, $\zeta''$, is related to the goal-tree of Prolog, and the savings in computation is illustrated. It is proved that $\zeta''$ satisfies the completeness and soundness properties of Prolog and that it preserves the order of solutions obtained by Prolog.

The notion of intelligent forward jumping is based on the observation that when backtracking occurs, say, from the subgoal position f to the subgoal position $1 \leq b < f$ within the body of a rule, then only some of the subgoals in the range $b \leq i \leq f$ (in short, $[b..f]$) need to be re-evaluated. The inequalities $\leftarrow, =, \geq$ between the literals indicate the order of appearance of the literals in a rule. The set of subgoals that have to be re-evaluated to ensure completeness of search is denoted by redo(b, f). The reason that we are able to avoid re-evaluation of the other subgoals in the range $[b..f]$ is that the re-execution of these subgoals would proceed exactly in the same way as before, through the same series of successes, failures, and backtrackings and generating the same intermediate variable bindings, if any, on the way [33, 34].

A separate representation scheme and method for forward jumping is being independently developed by Dr. Kundu. [33 - 34]
The forward jumping scheme avoids computation of successful subgoals which are independent of the corrective action taken to cure the failure of the currently failed subgoal. The list of subgoals that can potentially affect the cause of failure of the most recently failed subgoal is identified from the gc-lists of the variables in the rule. After backtracking, only those subgoals that are in the redo-list are recomputed, thus effecting forward jumping over subgoals that are not in this list.

Consider the rule

\[ r^* : p(X, Y) :- r(X), s(Y), t(X, Z), u(Z, M), v(Y, T). \]

Execution of \( p(X, Y) \) may proceed in the following manner *

1. \( r(X) \) succeeds, generating the binding \( X/a \)
2. \( s(Y) \) succeeds, generating the binding \( Y/b1 \)
3. \( t(a, Z) \) succeeds, generating the binding \( Z/c \)
4. \( u(c, M) \) succeeds, generating the binding \( M/d \)
5. \( v(b1, T) \) fails.

Backtracking would take to \( s(Y) \), for a new binding for \( Y \). In the process the search space of \( t(a, Z) \) and \( u(c, M) \) is deleted. After \( s(Y) \) succeeds, with a new binding for \( Y \), the goals \( t(a, Y) \) and \( u(c, M) \) start from their first alternative. Let the recomputation continue as follow:

6. \( s(Y) \) succeeds, generating the binding \( Y/b1 \)
7. \( t(a, Z) \) succeeds, generating the binding \( Z/c \)
8. \( u(c, M) \) succeeds, generating the binding \( M/d \)

Now, \( v(b2, T) \) is tried for solutions. Note that when \( s(Y) \) succeeds with a new binding \( Y/b2 \), \( t(a, Z) \) succeeds with the binding \( Z/c \), and \( u(c, M) \), succeeds with the binding \( M/d \). These recomputations of successful subgoals encountered in the previous computations need not be repeated. The execution of the goals \( t(X, Z) \) and \( u(Z, \)

* These observations were developed jointly with Kundu, [33].
M) are independent of the new solution of s(Y). Forward jumping is a scheme to avoid such unnecessary successful recomputation.
6.2. Dependencies among subgoals

The gc-list of the variables which appear on the RHS of a rule is the list of the literals which appear on the RHS of a rule and whose declaration has the variable X, as its argument or part of its argument. Note that since the declaration of the literal differs throughout the program, the gc-lists are pertinent only to the rule; it is valid as long as the rule is not modified. (Refer Chapter 4)

Consider a rule

"query(...) :- 1, 2, 3, ..., n."

where the literals are replaced by their corresponding indices. Consider two literals i and j in the rule.

**Definition 6.1:** A literal b is said to be exclusive of the literal f, and vice versa, if b and f do not share any variables. In other words, there is no gc-list which contains both b and f.

**Definition 6.2:** A literal b is said to be related to the literal f, and vice versa, if b is not exclusive of f. Note that every literal is related to itself.

**Definition 6.3:** A literal f is dependent on a literal b, f ≥ b, if it is related to i or if there exists a literal k, b ≤ k ≤ f, which is dependent on b. Every literal is dependent on itself.

**Definition 6.4:** In a range of literals [b..f], a literal p is parallel to a literal b, b ≤ m ≤ f, if p is not dependent on b and if there exists a literal k, p < k ≤ f, which is dependent on b.

**Definition 6.5:** In a range of literals [b..f] if a literal o > i is neither dependent on the literal b, nor is parallel to the literal b in the range [b..f], then the literal o is said to be orthogonal to the literal b in the range [b..f].
When two goals are exclusive of one another, they are semantically independent of one another. When two goals are related to one another, they are semantically dependent on one another. If goal \( b \) appears before the goal \( f \) and if the goals \( b \) and \( f \) are related to one another, then the solution of \( b \) affects the solution of \( f \). If literal \( f \) is dependent on \( b \), then the solution of \( f \) is directly or indirectly affected by the solution of \( b \). If a literal \( p \) is parallel to literal \( b \) in the range \([b..f]\), then the state of computation in which \( f \) is executed is affected by \( p \) as well as by \( b \). If a literal \( o \) is orthogonal to \( b \) in the range \([b..f]\), then it implies that the computation of \( b \) does not affect the computation of \( o \) and the computation of \( o \) does not affect the state in which \( f \) is executed.

6.3. Redo-list

Consider the situation when a rule \( r \) is executed and a subgoal \( f \) fails. Also, assume that the backtrack literal selected is \( b \), which is executed in the state \( \sigma \).

**Definition 6.6:** The set redo\((r, b, f, \sigma)\) is the set of goals which are affected by the recomputation of \( b \) and whose solutions affect the state in which \( f \) is executed.

The use of the redo-list is inter-procedural (i.e., applicable within a rule). When the rule application is understood we represent redo\((r, b, f, \sigma)\) by redo\((b, f)\). The computation of the redo-list for a range \([b..f]\) is done to find the subgoals in this range which need to be re-executed for completeness of the search.

**Example**

Refer to In Figure 1 consider the case when \( p4(a1, b1, U) \) fails. The selected backtrack literal is \( p2(X) \). Thus, \( f = 4 \) and \( b = 2 \). The re-execution of \( p2(X) \) may generate a new binding for \( X \). It is clear that no matter what the new binding of \( X \) may be, there is no need to re-evaluate \( p3(Z) \).
We denote by redo(b, f, σ) the list of the subgoals i, b ≤ i ≤ f, whose reexecution after backtracking to b is necessary to assure that no potential solution to the rule head is missed. The computation of redo(b, f, σ) in r' can be done using the gc-lists of the variables of r'.

The redo-list, redo(b, f), can be recursively defined as:

1. b and f belong to redo(b, f).
2. If j ∈ redo(b, f) and j ∈ con(r', X), then gc(r', X) ∩ [b..f] belongs to redo(b, f, σ).

Basically, redo(b, f) consists of all subgoals in the range [b..f] whose re-execution can potentially affect the continuation of the computation from the subgoal b through the subgoal f.

Example

Consider the following rule and its gc-lists:


\[ gc(X) = [2, 4, 7, 9, 11] \quad gc(U) = [4, 9, 10] \]
\[ gc(Y) = [1, 4, 8, 10] \quad gc(V) = [6, 7, 9] \]
\[ gc(Z) = [1, 3, 5, 11] \quad gc(W) = [5] \]

Figure 20 illustrates the possible values of redo(2, 9) and redo(2, 11) based on the various consumer lists.

† The definition of redo(b, f) which dynamically changes with different states was developed in conjunction with Kundu, [33 - 34].

§ Kundu, [34].
<table>
<thead>
<tr>
<th>Current gc-lists in state ( \sigma ) on backtracking to 2</th>
<th>redo(2, 9, ( \sigma ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10] )</td>
<td>([2, 4, 6, 7, 8, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Y) = [4, 8, 10], \text{con}(\sigma, V) = [6, 7, 9] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Z) = [3, 5, 11], \text{con}(\sigma, W) = [5] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10] )</td>
<td>([2, 4, 6, 7, 8, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Y \leftarrow y0) = \emptyset, \text{con}(\sigma, V) = [6, 7, 9] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Z) = [3, 5, 11], \text{con}(\sigma, W) = [5] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10] )</td>
<td>([2, 4, 6, 7, 8, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Y) = [4, 8, 10], \text{con}(\sigma, V) = [6, 7, 9] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\Sigma, Z \leftarrow z0) = \emptyset, \text{con}(\sigma, W) = [5] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Y \leftarrow y0) = \emptyset, \text{con}(\sigma, V) = [6, 7, 9] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
<tr>
<td>( \text{con}(\sigma, Z \leftarrow z0) = \emptyset, \text{con}(\sigma, W) = [5] )</td>
<td>([2, 4, 6, 7, 9])</td>
</tr>
</tbody>
</table>

Figure 20 (i). Possible values of redo(2, 9) for different consumer lists.
<table>
<thead>
<tr>
<th>Current gc-lists in state $\sigma$ on backtracking to $2$</th>
<th>$\text{redo}(2, 11, \sigma)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10]$</td>
<td>$[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$</td>
</tr>
<tr>
<td>$\text{con}(\sigma, Y) = [4, 8, 10], \text{con}(\sigma, V) = [6, 7, 9]$</td>
<td></td>
</tr>
<tr>
<td>$\text{con}(\sigma, Z) = [3, 5, 11], \text{con}(\sigma, W) = [5]$</td>
<td></td>
</tr>
<tr>
<td>$\text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10]$</td>
<td>$[2, 3, 4, 5, 6, 7, 9, 10, 11]$</td>
</tr>
<tr>
<td>$\text{con}(\sigma, Y) = \emptyset, \text{con}(\sigma, V) = [6, 7, 9]$</td>
<td></td>
</tr>
<tr>
<td>$\text{con}(\sigma, Z) = [3, 5, 11], \text{con}(\sigma, W) = [5]$</td>
<td></td>
</tr>
<tr>
<td>$\text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10]$</td>
<td>$[2, 4, 6, 7, 8, 9, 10, 11]$</td>
</tr>
<tr>
<td>$\text{con}(\Sigma, Z \leftarrow z0) = \emptyset, \text{con}(\sigma, W) = [5]$</td>
<td></td>
</tr>
<tr>
<td>$\text{con}(\sigma, X) = [2, 4, 7, 9, 11], \text{con}(\sigma, U) = [4, 9, 10]$</td>
<td>$[2, 4, 6, 7, 9, 10, 11]$</td>
</tr>
<tr>
<td>$\text{con}(\sigma, Y) = \emptyset, \text{con}(\sigma, V) = [6, 7, 9]$</td>
<td></td>
</tr>
<tr>
<td>$\text{con}(\sigma, Z \leftarrow z0) = \emptyset, \text{con}(\sigma, W) = [5]$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 20 (ii) Possible values of $\text{redo}(2, 11)$ for different consumer lists.

Note that if a literal $\sigma$ is orthogonal to $b$ and in the range $[b..f]$, then $\sigma$ is orthogonal to $b$ for any range $[b..f1]$, $f1 \leq f$. The orthogonality is to be recomputed for any other range.

The states in which the dependent goals of $b$ are executed are directly affected by the execution of the goal $b$. The states in which the goals which are parallel to $b$ are executed are not affected by the execution of $b$. The solutions of the goals dependent on $b$ and the goals that are parallel to $b$ are contributory to the state in which $f$ is executed.

The states in which the goals that are orthogonal to $b$ are executed are not affected by the execution of $b$. The solutions of the goals that are orthogonal to $b$ in the range $[b..f]$ do not affect the state in which $f$ is executed.
In the context of sequential Prolog execution, \( f \) is the failed goal, \( b \) is the goal backtracked to, and dependent and parallel goals form the redo-list. The recomputation of the orthogonal goals is redundant. The relationship between the subgoals in a rule is dynamic during execution.

Consider an application \( r' \) of a rule \( r \). When substitution of the form \( X \leftarrow a \) is applied to \( \text{head}(r) \), the relationships among the subgoals in \( r \) which are based on the variable \( X \) are affected. This is because \( gc(r', X) = \{0\} \). Also, consider a situation where the subgoals \([1..j]\) have succeeded. Also assume that the subgoal \( i, 0 \leq i \leq j \), has generated a binding for a variable \( X \). In this case, all members in \( gc(r', X) \) use the binding for \( X \) by \( X \). Hence, any subgoal \( g > i, g \in gc(r', X) \), is dependent on \( i \). For a subgoal \( g', i < g' \leq g \) and \( g' \in gc(r', X) \), we can also say that the subgoal \( g \) is dependent on \( g' \). But this dependence relation is of no consequence, due to the single assignment property of logic programs. Dynamic dependence relations are used in finding \( \text{redo}(b, f) \).

Consider an execution state in which the goals \([1..f]\) have succeeded. Also assume that a subgoal \( p, 1 \leq p \leq f \), is orthogonal to a subgoal \( b \) in the range \([b..f]\). As execution proceeds, a subgoal \( f1 > f \), which is related to \( b \), as well as \( p \), may succeed. Now \( p \) is parallel to \( b \) in the range \([b..fk], fk \geq f1 \). Thus the relationship between two subgoals in a rule may dynamically change from being orthogonal to parallel to orthogonal, and so on. As described earlier, only the dynamic parallel and orthogonal relationships, based on the dynamic dependence relationship are used in finding \( \text{redo}(b, f) \).

The literals in the body of the rule are denoted by the corresponding indices. Algorithm R1 is the redo-list generation list algorithm, which uses the dynamic relationships among the subgoals in a rule. The dynamic relationships can be known from the generators and consumers of the variables in the rule.
Let $f$ be the failed subgoal and $b$ be the selected backtrack literal.

\begin{verbatim}
redo-set = {b, f};
temp-set = {b, f};
var-set = {};
temp-var-set = var-set;

repeat
    repeat
        G = remove(temp-set); /* get an element from temp-set */
        temp-var-set = \{X | G ∈ con(X)\}
    until (temp-set) = \emptyset);
    if ((temp-var-set - var-set) = \emptyset) then break;
    var-set = var-set \cup temp-var-set;
    repeat
        X = remove(temp-var-set);
        if (b ≤ gen(X) ≤ f)
            then
                temp-set = temp-set \cup (con(var) \cap [b..f]);
    until (temp-var-set = \emptyset or temp-set = [b..f]);
    if ((temp-set - redo-set) = \emptyset) then break;
    redo-set = redo-set \cup temp-set;
    if (redo-set = [b..f]) then break;
until;

redo-list = order(remove-duplicates(redo-set));
\end{verbatim}

Algorithm R1: An algorithm to generate dynamic redo($b, f$).
Algorithm R1 is finite and terminates, the termination conditions being i) 4 when the redo-list contains all the subgoals in the range \([b..f]\), ii) 4 when there are no more new goals to be added to redo\((b, f)\) iii) 4 when there are no more new variables that are affected by the re-execution of \(b\).
6.4. Forward Jumping

The redo(b, f) is used for recomputation of intermediate subgoals during the forward-jump phase. Once this list is exhausted, the forward-execution algorithm, as described previously, is employed. After backtracking to b from f, the goals in redo(b, f) are re-executed. The goal b is executed for further solutions, and the rest of the goals in redo(b, f) are executed from their first alternative. This is effected as follows: after selecting b and finding redo(b, f), the search spaces of the goals in [b..f] - redo(b, f) are saved. After the re-execution of b, the execution continues by executing the goals in redo(b, f) and interleaving the search spaces, bringing back the saved search space, of the goals in [b..f] - redo(b, f) in the order of their appearance in the rule.

Example

Consider redo(σ, 2, 11) of Figure 20. (ii). After backtracking to 2, computation continues as follows.

<table>
<thead>
<tr>
<th>compute/overlay</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Continue Computation of 2</strong></td>
</tr>
<tr>
<td>Interleave the search space of 3</td>
</tr>
<tr>
<td>Compute 4</td>
</tr>
<tr>
<td>Interleave the search space of 5</td>
</tr>
<tr>
<td>Compute 6</td>
</tr>
<tr>
<td>Compute 7</td>
</tr>
<tr>
<td>Interleave the search space of 8</td>
</tr>
<tr>
<td>Compute 9</td>
</tr>
<tr>
<td>Compute 10</td>
</tr>
<tr>
<td>Compute 11</td>
</tr>
</tbody>
</table>
In the event of a failure of a goal $f' \in \text{redo}(b, f)$, $b \leq f' \leq f$, select a backtrack goal $b'$. The backtrack procedure takes us to $b'$. The net result of backtracking from $f$ has resulted in backtracking to $b'$. Compute $\text{redo}(b', f)$. Discard the search space of the subgoals that belongs to $\text{redo}(b', f)$. Then continue forward execution at $b'$.

Figure 21 shows the search tree based on from intelligent backtracking and forward jumping. Table V shows the trace of execution from intelligent backtracking and forward jumping.
Figure 21. Goal-search tree explored by Intelligent Backtracking and Forward Jumping.
<table>
<thead>
<tr>
<th>goal</th>
<th>success/fail</th>
<th>binding</th>
<th>failure-binding</th>
<th>backtrace literal</th>
<th>redo-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1(Y, Z), (1))</td>
<td>success</td>
<td>YA1; Zc1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p2(X), (1))</td>
<td>success</td>
<td>XA1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), (Zc1))</td>
<td>success</td>
<td>XA1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p4(X, Y, U), (Xa61, Ya61))</td>
<td>fail</td>
<td>Xa61</td>
<td>p2</td>
<td>(p2, p4)</td>
<td>-</td>
</tr>
<tr>
<td>(p5(X), (1))</td>
<td>success</td>
<td>XA2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p6(X, Y, U), (Xa2, Ya2))</td>
<td>success</td>
<td>UA1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p7(Z, X, U, W), (Zc1, Xa2, UA11))</td>
<td>fail</td>
<td>Zc1</td>
<td>p1</td>
<td>(p1, p3, p5)</td>
<td>-</td>
</tr>
<tr>
<td>(p1(Y, Z), (1))</td>
<td>success</td>
<td>YA1; Zc1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), (Zc11))</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p5(Z, X, U, W), (Zc1, Xa2, UA11))</td>
<td>fail</td>
<td>Zc1</td>
<td>p1</td>
<td>(p1, p3, p5)</td>
<td>-</td>
</tr>
<tr>
<td>(p7(Y, Z), (1))</td>
<td>success</td>
<td>YA2; Zc2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), (Zc2))</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p5(Z, X, U, W), (Zc2, Xa2, UA11))</td>
<td>success</td>
<td>WA1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p8(V), (1))</td>
<td>success</td>
<td>V/f1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p9(X, U, V), (Xa2, UA11, V/f1))</td>
<td>fail</td>
<td>V/f1</td>
<td>p6</td>
<td>(p6, p7, p9)</td>
<td>-</td>
</tr>
<tr>
<td>(p6(V), (1))</td>
<td>success</td>
<td>V/f2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p7(X, Y, (Xa2, V/f1))</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p9(X, U, V), (Xa2, UA11, V/f1))</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p10(Y, U), (YA2, UA11))</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p11(X, Z), (Xa2, Zc2))</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table V: Trace of Execution by Intelligent Backtracking and Forward Jumping

† Identified from FBT(p4) search.
6.5. Soundness and Completeness of $\zeta''$

The search based on forward jumping, $\zeta''$, will be proved to be sound and complete. It is also shown that the order of solutions obtained by $\zeta''$ is the same as obtained by Prolog.

**Theorem 6.1.** $\zeta''$ is sound.

*Proof.* Solutions are obtained only if a goal unifies with a fact or all the subgoals in a rule applicable to the goal are true. Hence, every solution obtained by $\zeta''$ is sound.

**Theorem 6.2.** $\zeta''$ is complete.

*Proof.* In the default case, where $\text{redo}(b, f) = [b..f]$, the search is complete. The recomputation of the goals in the $\text{redo}(b, f)$ (other than $b$) from their first alternative is necessary, because the solutions of these goals can potentially affect the state in which $f$ is executed. As observed, the computation of the goals that are not in the redo-list are independent of the corrective actions taken to cure the failure of $f$. This ensures that no unique search path which would succeed, or fail, is missed. Hence, $\zeta''$ is complete.

**Theorem 6.3.** $\zeta$ preserves the order of results obtained.

*Proof.* The default case when $\text{redo}(b, f) = [b..f]$, yields the same results as those obtained by Prolog. Otherwise, the solutions of the different subgoals are interleaved during forward execution in the order of their appearance in the rule. Since the selection of control is still the same as in Prolog, the order of solutions obtained by $\zeta''$ is unaffected.
6.6. Difficulty in Implementation of $\xi$

The search space of a subgoal cannot be associated with a subgoal during execution due to the nature of the linear local stack. It is not possible to isolate and store the search space of intermediate goals while backtracking and to interleave them during the forward jumping process. The next chapter on the Segmented Stack presents a data structure that facilitates association of search spaces with goals. This can help forward jumping during sequential execution.
Chapter 7

7. Execution of Logic Programs on a Segmented Stack

7.1. Introduction

A new data structure, called the segmented stack, for execution of logic programs is introduced. The segmented stack can be maintained as a right-threaded tree with a heap pointer. A detailed illustration of the segmented stack environment is presented. It can be used as the data structure for the local-stack used by Prolog. The use of a segmented stack as the data structure to maintain the local-stack enables association of search spaces and solutions with goals. This in turn results in implementation of \( \zeta, \zeta' \) and \( \zeta'' \). Forward and backward execution algorithms are given. The space and time complexity analysis of the segmented stack is analyzed.

Prolog programs are evaluated on a linear stack of activation records, called the local-stack or the environment-stack, together with the goal-list. The goal-list is a list of goals which remain to be solved in order to solve the initial query. It attempts to unify with the goal, \( t \), at the top of the local-stack with its alternatives. If \( t \) unifies with a fact, then \( t \) succeeds. If \( t \) unifies with a rule-head of a rule \( r \), then the body(\( r \)) is inserted at the head of the goal-list. If the local-stack is empty or if the goal at the top of the local-stack succeeds, then the goal at the head of the goal-list, \( h \), is selected for solving. This modifies the goal-list in that \( h \) is removed from the goal-list. A record of \( h \) is stacked on top of the local stack of activation records. The goal \( h \) is tried for unification with its first alternative. If \( h \) unifies with a fact, then \( h \) succeeds. If \( h \) unifies with a rule-head of the rule \( r \), then the body(\( r \)) is inserted at the head of the goal-list. The local-stack is initialized empty. The local-stack grows with the invocation of goals by forward execution. It contains the activation records of all the goals that are active. The records on the local-stack constitute the current path in the goal-search tree that is being tried to solve the initial query. The goal-list is initialized with
the query goal. It is empty on successful completion of the query. When the query
goal succeeds, the local-stack contains the goals falling in the current path that is
being tried for a solution. Thus, forward execution modifies the goal-list and the
local-stack.

A goal $f$ fails when no more alternatives are left for unification. Backward exe-
cution is as follows:

The goal $f$ is inserted at the head of the goal-list.

A backtrack literal is selected.

Goals are popped off the local-stack, their alternatives reset, and inserted at
the head of the goal-list - the unification substitution created by this goal is
undone - till the backtrack literal is encountered at the top of the stack.

All the subgoals in body($r$), of the rule $r$ which unified with a goal $g$, and
which are in the goal-list are removed from the goal-list when the goal $g$ is
encountered at the top of the local-stack.

The local-stack shrinks during backtracking. The goal-list grows and
shrinks during computation. Forward execution is resumed with the goal at the
top of the local-stack. When the query goal fails, the goal-list contains the query
goal and the local-stack is empty. Operations on the local-stack and the goal-list
occur hand-in-hand in order to implement, and execute, exhaustive search for
Prolog programs. A record on the local stack contains all necessary information
to implement, correctly, backtracking and continuation of search. A goal is
active as long as its activation record is present on the local-stack.
7.2. Inapplicability of Linear Stack to Implement $\zeta'$ and $\zeta''$

A goal may be solved by way of unification with a fact or by a rule which succeeds. The rules that are applicable for a goal, in general, have more than one subgoal, and one or more of these subgoals may be solved by using the rules applicable to them. Hence, in general, the successful search path of one goal contains more than one record on the local-stack.

The search paths of different subgoals are stacked on top of one another. While backtracking, the records are popped off the stack until the backtrack point is reached. The records of the different subgoals in a rule are not readily accessible without a linear popping of the records from the local-stack. Continuation of search of an intermediate subgoal going back to where the system left off, cannot be implemented on a linear stack. Also, the information on a linear stack is popped off during backtracking, the search space. Subsequently, these popped records are not retrievable.

A solution to a goal is obtained only when the rule succeeds. The record of the goal which succeeds is embedded deep in the stack. The solution of a goal cannot be associated with the record of the goal. Hence, it is not possible to associate FST's with a goal. Independent search spaces cannot be saved and interleaved in a straightforward fashion. Hence, $\zeta'$ and $\zeta''$ cannot be implemented on a linear stack of records as data the structure for the local-stack.
7.3. Segmented Stack

A linear stack structure for the local-stack does not facilitate maintenance of FST's with goals or for saving and interleaving search spaces for forward-jumping over subgoals. A data structure facilitating depth-first search, as well as breadth-first search, is required. A new data structure, the segmented-stack, for the local-stack is proposed in order to enable both depth-first search and breadth-first search during execution of Prolog programs. Depth-first search is employed when unification succeeds and during forward execution, and breadth-first search is employed during intelligent backward execution, intelligent forward execution and forward jumping.

Definition of a Segmented Stack:

A node in a segmented stack, SS, has three fields. These are

1) a rule-pointer,

2) a heap-pointer and

3) data.

The rules defining a segmented stack are,

1) A segmented-stack, SS, of zero nodes is an empty segmented-stack.

2) A segmented stack, SS, with a single node, which is the root of SS, is a segmented-stack.

3) A node, which is the root, together with its rule-pointer, pointing to a list of segmented-stacks, which form the body, is a segmented-stack.
A segmented-stack, SS_g can be used as a local-stack for execution of a goal g as follows:

1. A node for g is created when g is executed.

2. If g unifies with a fact, then the rule-pointer in the node for g is null, the tree of SS_g is empty and the heap-pointer in the node for g points to itself.

3. If g unifies with a rule r, then each subgoal, g_i in body(r) is solved on a separate segmented-stack, SS_i. All such segmented-stacks, SS_i, 1 ≤ i ≤ n, where n is the number of subgoals in body(r) which has been solved, form the tree of the segmented-stack rooted at g. The rule-pointer in the node for g points to the list {SS_1, SS_2, ..., SS_n}. The heap-pointer points to the node pointed to by the heap-pointer of SS_n. The list of segmented-stacks together with the root g is the segmented-stack SS_g. Thus the heap-pointer points to the last record formed on the segmented-stack.

In the context of Prolog execution, this corresponds to a goal unifying with head(r) of a rule r. The subgoals in body(r) are solved over the segmented-stacks pointed to by the list, pointed to by the rule-pointer.

A node is an activation record of a goal. The list implementation of the segmented-stacks for different subgoals in a rule enables operation on any of these subgoals. Hence, the search space of any of the subgoals in body(r) can be deleted. Also, the heap-pointer of the segmented stack of a subgoal can be used to continue execution of the subgoal. Thus, a subgoal in a rule solved on a segmented-stack can be deleted or continued for further solutions without affecting the search space of other subgoals in the rule. The logical connectivity among the subgoals in a rule is maintained by the list of segmented-stacks over which the subgoals are solved. The logical connectivity between a goal and the body(r) of the rule r applied to solve the goal is maintained by the rule-pointer of the record of the goal.
Example

Consider the facts and rules in Figure 11 and a goal r(a1, b1). A node for r(a1, b1) is created. The rule-pointer and heap-pointer are undefined. When r(a1, b1) unifies with the first rule of r the rule-pointer points to a list of two segmented stack. r11(a1) is executed on the first segmented stack and r12(b1) on the second. r11(a1) unifies with a fact and so the corresponding segmented stack is a single node pointing to itself.

The segmented-stack needs to hold all the information that would be held in the equivalent local-stack during the execution of logic programs. Since there is a record for every goal that is active, this information would be consistent with that of a linear stack implementation of the local-stack.

A segmented-stack is active as long as its root is present. An active segmented-stack can be in either of the two states: Open or Closed. If the heap-pointer of an active segmented-stack, SS_a, is null, then it is said to be Open. Otherwise it is said to be Closed.

In the context of Prolog execution

1. An open segmented-stack with a single node corresponds to the segmented-stack for a goal whose unification is in progress.
2. A closed segmented-stack with a single node corresponds to a segmented-stack for a goal which has succeeded by unification with a fact.
3. An open segmented-stack with multiple nodes corresponds to a segmented-stack for a goal for which a rule application has not yet terminated.
4. A closed segmented-stack with multiple nodes corresponds to a segmented-stack for a goal which has succeeded by a rule application.
The operations on a segmented-stack are:

\texttt{mk\_seg}

Segmented-stacks are created by the \texttt{mk\_seg()} procedure. This procedure opens a segmented-stack by creating a node for the goal to be executed on this segmented-stack.

\texttt{close\_seg}

Segmented-stacks are closed by the \texttt{close\_seg()} procedure. This procedure closes a segmented-stack by recording the heap-pointer at the of the segmented-stack.

\texttt{get\_top(arg)}

Segmented-stacks are accessed by the \texttt{get\_top(arg)} procedure. This procedure goes to the top of the segmented-stack referred to by the argument via the heap-pointer of the segmented-stack, referred to by \texttt{arg}.

\texttt{mk\_null(arg)}

A segmented-stack is deleted by the \texttt{mk\_null(arg)} procedure. This procedure deletes the tree of the segmented-stack referred to by \texttt{arg}. It also undefines, makes null, the rule-pointer and the heap-pointer at the root of this segmented-stack.

A segmented-stack is active during the interim of the \texttt{mk\_seg()} which created it and the \texttt{mk\_null(arg)} call referring to its parent. The above procedures are sufficient to create and maintain a segmented-stack.
7.4. Forward Execution on a Segmented Stack

The two main data-structures other than the goal-list and the segmented-stack are the heap and trail. The heap maintains the address of the choice-points [32] and the trail maintains the unification substitution informations. The goal-list contains the goals to be computed to solve the primary goal, and the segmented-stack contains the nodes for the goals that have been computed, partially or fully, constituting the path from root to the present node in the goal-search-tree. Algorithm A3 gives the Forward Execution Algorithm on a segmented stack.

1. A segmented-stack is created when a goal, G, is first tried for solution.

2. If G unifies with a fact, then the segmented-stack corresponding to this goal is closed using the address of the node of G itself for the heap-pointer of G.

3. If G unifies with a rule, then a list of segmented-stacks are created to solve each of the subgoals in the rule-body. When each of these segmented-stacks that solve these subgoals are closed, then the segmented-stack of G is closed by updating the heap-pointer of the segmented-stack of G with the heap-pointer of the last segmented-stack which solved the last goal in the rule.

Algorithm A3 is used to build the segmented-stack as the forward execution proceeds. If any of the subgoals fail, then backward execution is applied.
Phase I

if (goal-list = ∅)
then
  report success
  Apply Algorithm B3
else
  t_goal = get(goal-list); /* get first element from goal-list */
  if t_goal = '.'
  then
    close_seg(); /* record the pointer to the top of the stack ***/
  else
    mk_seg(); /* create a new segmented stack and record the 
                pointer of its root ***/

Phase II

t_list = unify(t_goal)
if (t_list = fail)
then
  report failure
  Apply Algorithm B3
else
  cons(t_list, goal_list);
  Apply Algorithm A3 at Phase I

Phase III

t-goal = get-top(b);
Apply Forward Execution at Phase I

Algorithm A3: Forward Execution Algorithm on Segmented-stack.
7.5. Backward Execution on a Segmented Stack

Backward execution allows search for other alternate solutions. During backward execution, many subgoals in a rule are to be redone. After the detection of the cause of failure of a subgoal, and subsequent detection of the backtrack-literal and the redo-list, the segmented-stack is updated to perform backtracking and forward jumping.

When the backward execution algorithm is invoked, a backtrack literal, b, is selected together with the redo-list. The search space on the segmented-stack of the goals in the redo-list are deleted, the tree of the corresponding segmented-stack is deleted, and the rule-pointer and heap-pointer set to null. Forward execution is resumed, at Phase II, at b. If b fails, then backward execution is applied. If b succeeds, then the forward-jumping algorithm is applied. Algorithm B3 gives the backward execution algorithm for the segmented-stack.

Let f be the failed literal.

\[ b = \text{select(backtrack-literal)} \]

\[ \text{generate redo}(b, f) \]

\[ \text{for } (i \in \text{ redo-list}) \text{ do} \]

\[ \text{mk-null}(i); \]

\[ \text{backtrack}(b); \]

\[ \text{modify}(\text{FST}(b)); \]

\[ \text{repeat} \]

\[ \text{Apply Algorithm A3 Phase II to } b \]

\[ \text{until (result is not subsumed in FST}(b)) \]

\[ \text{if (result = failure) then} \]

\[ \text{report failure} \]

\[ \text{Apply Algorithm B3} \]
else

for (i ∈ redo(b, f)) do

Apply Algorithm A3 Phase II to i as t_goal.

Algorithm B3: Backward Execution Algorithm on a Segmented-stack.

The segmented-stack representation of the local-stack for two instances from Figure 1. are shown in Figure 22. It shows the state where p0 has unified with the rule-head and when the execution of the subgoals p1, p2, p3, p4 and p5 have succeeded. Note that until all the subgoals in the rule succeed, the heap-pointer with the node for p0 will remain dangling, i.e., undefined.

Figure 22. Execution of Logic programs on a segmented stack, a case from Figure 1.
7.6. **Space Complexity of Segmented Stack**

There is an additional usage of space in implementing the segmented stack in the form of two additional fields in each frame. These are used for maintaining the rule-pointer and the segment pointer. The space complexity of the local-stack is unaltered. With minimal overhead in space, significant efficiency in execution can be obtained. The maintenance of the segmented stack is simple and efficient.

In implementing forward jumping, there is a dynamic overhead in space. This overhead in the form of saving the search subspace of the successful subgoals that do not fall in the redo-list, while backtracking and forward jumping are taking place. This dynamic overhead in the use of space is directly proportional to the savings in computations derived by forward jumping.

Also, since a subgoal is computed on a separate segmented stack, the solutions of the subgoal can be associated with it. This facilitates association of FST's with a goal, so we can implement intelligent forward execution.

When space is at premium, forward jumping can be avoided by simply backtracking to the appropriate goal and performing the normal operations of popping the stack and only implementing \( \zeta' \).
Chapter 8

8. Effects of Non-logical Operators

8.1. Introduction

Non-logical operators in Prolog, such as =, >, <, assert, write, cut, fail, facilitate programming. These operators have side-effects while performing mathematical operations, while avoiding search using control operations like cut and fail, and while performing input/output operations. These operators are built on top of the systematic control facility of the Prolog interpreter. It is essential that every new paradigm for Prolog, or logic programming, incorporates these facilities in a transparent manner.

The operators +, -, *, /, =, <, >, write and read do not have any effect on the paradigms developed in chapters 4, 5, and 6. The operator "consult" modifies the program and hence all the identified failure-bindings, failure-solutions and gc-lists stand void. The effects of other non-logical operators on the different paradigms and data structures is described below.
8.2. Effect of Cut and Fail

The cut is a control facility provided to the programmer to help prune certain searches. The predicate cut always succeeds. When forward execution encounters the predicate cut, all subsequent searches of the subgoals to the left of the cut in the rule, and subsequent alternatives that are applicable to the goal that unified with the head, are avoided.

The effect of the cut on the identification of failure-bindings is contributory to its incompleteness. This is because not all the alternatives of a goal are tried when any of the alternative applicable to the goal execute a cut. Hence, no failure-bindings can be identified.

There is no effect of the cut on identification of failure-solutions. Similarly, the forward jumping procedure is unaltered if any of the subgoals in a rule applied to a goal contains a cut. The use of the segmented stack as a data structure for the local stack does not hinder the control implemented by the cut.

The control operator "fail" does not have any effect on the paradigms developed in the earlier chapters.
8.3. Effect of Assert and Retract

Retract removes information from the database and hence does not affect either the identified failure-bindings or failure-solutions, though it could lead to not identifying some failure-bindings during later execution. Assert, however, adds to the information content of the database. This voids all identified failure-bindings and failure-solutions. Hence, FBT's and FST's of all predicates and goals are made void.
Chapter 9

9. Summary

9.1. Conclusions

We have presented three paradigms and a new data structure for implementing these paradigms. The framework implementing these paradigms on the new data structure leads to very efficient execution of logic programs.

1. The identification and use of failure-bindings during execution leads to alleviation of extensive oscillation of computation between backtracking and forward execution. Although the identification of failure-bindings is not complete, the late detection of failure is cured to a great extent by the bit-vector implementation and the use of failure-binding-tables. The time and space incurred in the identification of failure-bindings of a goal is identical to the nodes in the goal-search tree of a goal.

2. The identification and use of failure-solutions during execution leads to elimination of retracing of identical paths in the goal-search of the primary goal. In other words, the states encountered by the use of failure-solutions are unique. Thus, the resulting search is deterministic in nature. The time and space incurred in identifying a failure-solution of a goal equals the number of nodes in the goal-search tree of the initial query which is under a subgoal.

3. The use of forward jumping leads to further pruning of the goal-search tree. Forward jumping leads to elimination of computations with combinations of bindings which have been identified to cause failure. The time and space incurred in determining redo-list for forward jumping equals $N_{\text{var}} \times N_{\text{subgoals}}$, where $N_{\text{var}}$ is the maximum number of variables used in a rule and $N_{\text{subgoals}}$ is
the number of goals in the rule.

4. As described in chapter 7, the additional space and time incurred in executing logic programs on a segmented-stack, as the data structure for the local-stack, equals two additional fields in each frame and their updating. Thus, additional savings in computation can be found by implementing the techniques described in this dissertation.

5. The effect of non-logical operators is straightforward so that these operators can be implemented properly in the context of efficient search.

Figure 23 presents a flow chart of the execution box of a goal for the framework which would implement intelligent backtracking based on failure-bindings identified by the bit-vector method, intelligent forward execution based on failure-solutions, and forward jumping.

Figure 24 shows the goal-tree which uses all the three paradigms for the example in Figure 1. Table VI gives the trace of execution for the example in Figure 1 based on the use of all the three paradigms.
Figure 23. Execution Box of a goal implementing Intelligent Backtracking, Intelligent Forward Execution and Forward Jumping
Figure 24. Goal-tree by Intelligent Backtracking, Intelligent Forward Execution and Forward Jumping
<table>
<thead>
<tr>
<th>goal</th>
<th>success/fail</th>
<th>binding</th>
<th>fb</th>
<th>∈ FST?</th>
<th>b</th>
<th>redo-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1(Y, Z), {Y, Z}, {})</td>
<td>success</td>
<td>Y/a1; Z/c1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p2(X), {})</td>
<td>success</td>
<td>X/a1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p3(Z), {Z/c2})</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p4(X, Y, U), {X/a1, Y/b1})</td>
<td>fail</td>
<td>-</td>
<td>X/a1</td>
<td>p2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p5(Z, X, U), {Z/c1, X/a2, U/d1})</td>
<td>success</td>
<td>X/a2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p6(V), {})</td>
<td>success</td>
<td>U/d1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p7(X, V), {X/a2, V/f1})</td>
<td>success</td>
<td>-</td>
<td>Z/c1</td>
<td>p1</td>
<td>p2</td>
<td>-</td>
</tr>
<tr>
<td>(p8(Y), {Y/b2})</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p9(X, U, V), {X/a2, U/d1, V/f1})</td>
<td>fail</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p10(Y, U), {Y/b2, U/d1})</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(p11(X, Z), {X/a2, Z/c2})</td>
<td>success</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table VI: Trace of Execution by Intelligent Backtracking. Intelligent Forward Execution and Forward Jumping

† Identified from FBT(p4) search. fb and b in the header row is an acronym for failure-binding and backtrack literal, respectively.
Each of these paradigms is defined and systematic evaluation of these methods is presented in detail. Forward- and backward-execution algorithms that implement these paradigms are given. It is proved that the application of these paradigms is sound and complete in that the resulting execution based on these paradigms are sound and complete. Also, the application of these paradigms preserves the order of results obtained by Prolog. The properties of the use of these paradigms and data structure are:

**Failure Binding (Chapter 4)**
- Precise cause of failure.
- State independent. Intelligent backtracking farthest to the left than has been previously possible.
- Global learning of information about predicates.

**Failure Solution (Ch. 5)**
- Use of history of computation.
- State dependent.
- Intelligent forward execution eliminates repetition of unsuccessful search.
- Local learning about the solutions of goals.

**Forward Jumping (Ch. 6)**
- Use of generator consumer approach.
- State dependent.
- Eliminates repetition of successful search after backtracking.

**Segmented Stack (Ch. 7)**
- Facilitates association of information about a goal with the goal.
- Keeps space and time complexity identical to that of linear stack.
- Used for interleaved depth-first and breadth-first search.

A simple comparison of the number of steps performed by the application of different
paradigms and that by Prolog, for the example in Figure 1, from the Tables I, II, III, IV and V is given below.

<table>
<thead>
<tr>
<th>Standard Prolog</th>
<th>$\zeta$</th>
<th>$\zeta'$</th>
<th>$\zeta''$</th>
<th>Unified framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>34</td>
<td>34</td>
<td>22</td>
<td>21</td>
</tr>
</tbody>
</table>

This project was started in conjunction with Kundu, [33] and [59]. The concepts in Kundu, [34], differ in that the failure-bindings described in this dissertation are precise causes of failure and that by Kundu are composite failure-bindings. The use of FBT is similar, if not identical. The representation schemes for the dependencies among the literals in a rule are different. The generation of redo-list is based on the representation schemes.

An implementation of an interpreter based on the paradigms described in this chapter was undertaken by Mr. Pramod Korwar, under the direction of Kundu. This interpreter implemented a prototype of the segmented stack and the bit-vector method for identification of "precise" failure-binding, as described in chapter 4. For more details on this implementation contact Kundu, [33].
9.2. Future Directions

The area of efficient execution of logic programs needs to be further explored. Some of the open issues that can be stated are as follows:

a) Extension of the paradigms introduced in this dissertation to logic programs which contain generalized terms containing lists$^*$. sup

b) Is it possible to find a method which is complete in identifying failure-bindings of a goal?

c) Can the identification of failure-bindings lead further pruning than has been described in this dissertation?

d) How to make the FST and FBT table look up procedures more efficient?

e) A method to identify failure-bindings of a negated goal.

f) A user friendly, declarative style to control paradigm to specify control of search.

g) A formal and accurate time and complexity analysis model for comparing the execution models for logic programs.

† The application of failure-bindings to the case of logic programs with generalized terms is being investigated independently Kundu, [33-34].
References


2 Lloyd, J. W., Mathematical foundations of logic programming, Research Monograph, Dept. of Computer Science, Univ. of Melbourne, Australia, 1983.


7 Kowalski, R. A., Predicate logic as a programming language, Proc. of IFIP-74 North Holland Publ., Amsterdam, pp. 569-574.


10 Kowalski, R., in, Logic for Problem Solving, Elsevier, North Holland, New


14 Bruynooghe, M., Analysis of Dependencies to Improve the Behaviour of Logic Programs, Conference on Automated Deduction, 1980, pp. 293-305.


18 Chang, J. H., Despain, A. M., Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis, Proceedings of IEEE Sympo-


26 Woo, N., Choe, K., Selecting the Backtrack Literal in the AND/OR Process

27 Rajasekhar, S, Narasimha Murthy, Variable Based Backtracking Method, Conference on
28 The memory management of Prolog implementations, Clark and Tarnlund.
33 Kundu, S., private communications and different versions of reports exchanged, October, 1987 through March, 1989.
34 Kundu, S., paper under preparation pertaining to generalized failure-bindings and their use for backtracking and forward jumping.
35 Mellish, C. S., An alternative to structure sharing in the implementation of Prolog interpreter, Clark and Tarnlund.
36 Clark, K., Gregory, S., PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and


DeGroot, D., Restricted And-Parallelism, ICOT International Conference


50 Byrd, L., Understanding the control flow of Prolog programs, Logic programming workshop 1980.


52 Li, P. P., Martin, A. J., The Sync Model: A Parallel Execution Method for


Vita

Sudharsan R. Iyengar was born to Pankajam and Rengaswamy Aravamudhan on First of November, 1961, in the town of Srirangam, Madras, India. After finishing Secondary School Certificate at Fatima High School, Bombay, and Higher Secondary Certificate, at CHMC, Bombay, he went on to study for the Bachelor in Engineering Degree, at Regional Engineering college, Durgapur. After obtaining the B.E. in Electrical Engineering, in May 1984, he went on to pursue higher education at Louisiana State University, Baton Rouge. He got his M.S. in Systems Science and M.S. in Electrical Engineering in May 1987. He is currently a candidate for his Ph.D. degree in the Department of Computer Science, at LSU, Baton Rouge.
DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Sudharsan R. Iyengar
(first) (last)

Major Field: COMPUTER SCIENCE

Title of Dissertation: A Framework for Efficient Execution of Logic Programs,

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination: 6/29/89