INFORMATION TO USERS

This manuscript has been reproduced 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.

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.
Automatic complexity analysis of logic programs

Lin, Nai-Wei, Ph.D.
The University of Arizona, 1993

Copyright ©1993 by Lin, Nai-Wei. All rights reserved.
As members of the Final Examination Committee, we certify that we have read the document prepared by Nai-Wei Lin entitled Automatic Complexity Analysis of Logic Programs and recommend that it be accepted as fulfilling the requirements for the Degree of Doctor of Philosophy.

Saumya K. Debray
Date 6 May 1993

Peter J. Denley
Date 6 May 1993

Richard T. Snodgrass
Date 6 May 1993

William G. Faris
Date 6 May 1993

Daniel Madden
Date

Final approval and acceptance of this document is contingent upon the candidate's submission of the final copy of the document to the Graduate College.

I hereby certify that I have read this document prepared under my direction and recommend that it be accepted as fulfilling the requirement.

Saumya K. Debray
Director
Date June 1, 1993
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: Mei-Wei Lin 4/6/93
ACKNOWLEDGMENTS

I would first like to thank my advisor Saumya Debray for his continuous support, encouragement, and guidance. His knowledge on various aspects of research provided crucial advices through this work. His style of research and enthusiasm also established a model for me to follow.

I would also like to thank the members of the Committee, Peter Downey, Richard Snoegrass, William Faris, and Daniel Madden for their interest, encouragement, and invaluable comments and suggestions to this document. Udi Manber and Paul Fan provided their encouragement and guidance in early stages of this work.

Special thanks are due to Manuel Hermenegildo for many perceptive discussions through this work and providing the experimental measurements for &-Prolog system. I am also grateful to Laxmikant Kalé and Balkrishna Ramkumar for making available their ROLOG system for my experiments, and for many helpful discussions and suggestions.

The work benefited from discussions with Jacques Cohen and Timothy Hickey on the problem of automatic complexity analysis of programs, with Yves Deville on the analysis of number of solutions complexity, and with Evan Tick on the issue of granularity analysis of logic programs. I am indebted to their illuminating comments and suggestions.

I would also like to thank many fellow students for helpful discussions, notably, Peter Bigot, David Gudeman, Herman Rao, and Sun Wu. The Department of Computer Science at The University of Arizona provided an excellent environment for research. Thanks to the faculty and staff of the Department for establishing a pleasant and supportive environment.

This work was supported in part by the National Science Foundation under grants number CCR-8702939 and CCR-8901283.
# TABLE OF CONTENTS

LIST OF FIGURES ........................................... 9

LIST OF TABLES ........................................... 11

ABSTRACT .................................................... 12

CHAPTER 1: INTRODUCTION ..................................... 13
  1.1 Complexity Analysis of Programs .......................... 13
  1.2 Automatic Complexity Analysis of Programs ............. 14
    1.2.1 Imperative Programs ............................. 14
    1.2.2 Functional Programs ............................. 15
    1.2.3 Logic Programs ................................. 17
  1.3 Applications of Automatic Complexity Analysis .......... 18
  1.4 Contributions ......................................... 19
  1.5 Organization of the Dissertation ........................ 20

CHAPTER 2: PRELIMINARIES .................................. 21
  2.1 Graph Theory ......................................... 21
  2.2 Logic Programming .................................... 24

CHAPTER 3: A FRAMEWORK FOR COMPLEXITY ANALYSIS ............. 31
  3.1 Logic Programs vs. Imperative and Functional Programs 31
  3.2 The Framework ........................................ 34
  3.3 Overview of an Implementation .......................... 36
  3.4 Summary ............................................... 38

CHAPTER 4: THE PREPROCESSOR ................................ 39
  4.1 Mode Declarations .................................... 39
  4.2 Measure Declarations .................................. 42
LIST OF FIGURES

2.1 Examples of graphs ........................................... 22
2.2 An example of a computation ................................. 27
2.3 An example of a finite search tree ........................... 28
2.4 An example of an infinite search tree ....................... 29

3.1 Organization of CASLOG ....................................... 37

4.1 An example of a type graph ................................... 43
4.2 The definition for some common size measures ............. 45
4.3 An example of a call graph and a strongly connected components graph .......... 50
4.4 Examples of argument dependency graphs and literal dependency graphs . 54
4.5 An algorithm for constructing the argument dependency graph of a clause 55

5.1 The definition of size for some common size measures .... 60
5.2 The definition of diff for some common size measures ...... 61
5.3 An algorithm for argument-size complexity analysis of a clause ............ 71
5.4 An algorithm for argument-size complexity analysis of a predicate .......... 74

6.1 An algorithm for number-of-solutions complexity analysis of a clause .... 87
6.2 An algorithm for number-of-solutions complexity analysis of a predicate ... 91

7.1 An algorithm for relation-size complexity analysis of a clause ............ 105
7.2 An algorithm for solution-size complexity analysis of a clause ............ 108

8.1 An example of a consistency graph ............................ 121
8.2 An example of graph addition .................................. 123
8.3 An algorithm for estimating the number of n-cliques in a weighted
    consistency graph .............................................. 126
8.4 Experimental measurements for the n-cliques algorithms .................. 131
8.5 Average behavior of experimental measurements for the $n$-cliques algorithms ........................................ 132
9.1 An example graph .................................................. 136
9.2 The accumulating and interfacing subgraphs of the graph in Figure 9.1 .... 136
9.3 An algorithm for estimating the chromatic polynomial of a graph .......... 141
9.4 Graph $K_{m,2}$ ......................................................... 144
9.5 Experimental measurements for Algorithm 9.1 .............................. 146
10.1 An algorithm for time complexity analysis of a clause .................... 149
10.2 An algorithm for time complexity analysis of a predicate ................. 152
11.1 An algorithm for computing a nondecreasing upper bound solution to a set of difference equations ..................................... 167
12.1 Execution time vs. task granularity on ROLOG .......................... 179
12.2 Execution time vs. task granularity on &-Prolog .......................... 180
LIST OF TABLES

5.1 Built-in predicates used for the inference of the size of a variable . . . . . 72
12.1 Execution times for benchmarks on ROLOG (times in milliseconds) . . . . 181
12.2 Execution times for benchmarks on &-Prolog (times in milliseconds) . . . 181
13.1 Measurements for global goal ordering transformation (times in seconds) . 191
13.2 Measurements for local goal ordering transformation (times in seconds) . 194
13.3 Measurements for domain reducing optimization (times in seconds) . . . . 197
13.4 Measurements for clause ordering transformation (times in milliseconds) . 199
ABSTRACT

This dissertation describes research toward automatic complexity analysis of logic programs and its applications. Automatic complexity analysis of programs concerns the inference of the amount of computational resources consumed during program execution, and has been studied primarily in the context of imperative and functional languages. This dissertation extends these techniques to logic programs so that they can handle nondeterminism, namely, the generation of multiple solutions via backtracking. We describe the design and implementation of a (semi)-automatic worst-case complexity analysis system for logic programs. This system can conduct the worst-case analysis for several complexity measures, such as argument size, number of solutions, and execution time.

This dissertation also describes an application of such analyses, namely, a runtime mechanism for controlling task granularity in parallel logic programming systems. The performance of parallel systems often starts to degrade when the concurrent tasks in the systems become too fine-grained. Our approach to granularity control is based on time complexity information. With this information, we can compare the execution cost of a procedure with the average process creation overhead of the underlying system to determine at runtime if we should spawn a procedure call as a new concurrent task or just execute it sequentially. Through experimental measurements, we show that this mechanism can substantially improve the performance of parallel systems in many cases.

This dissertation also presents several source-level program transformation techniques for optimizing the evaluation of logic programs containing finite-domain constraints. These techniques are based on number-of-solutions complexity information. The techniques include planning the evaluation order of subgoals, reducing the domain of variables, and planning the instantiation order of variable values. This application allows us to solve a problem by starting with a more declarative but less efficient program, and then automatically transforming it into a more efficient program. Through experimental measurements we show that these program transformation techniques can significantly improve the efficiency of the class of programs containing finite-domain constraints in most cases.
CHAPTER 1
INTRODUCTION

1.1 Complexity Analysis of Programs

Computer users usually have a strong desire to understand the behavior of the programs they use. In particular, they are concerned with the computational complexity of the programs, namely, the amount of computational resources, such as time and memory space, consumed during execution. Apart from correctness of the programs, they often demand that programs are executed as efficiently as possible. Complexity analysis of programs aims to infer information about the computational complexity of programs by examining the program text.

Information about the computational complexity of programs is useful in a number of ways. When several programs are available for solving a particular problem, the computational complexity of the programs provides a natural criterion for determining which of them is the best. Moreover, the computational complexity of a program may also indicate how the computational resources are consumed by the components of the program. This information allows us to identify the most expensive components in the program, and thereby to effectively improve the efficiency of the program by focusing on optimizing these crucial components.

Complexity analysis of programs has appeared in the earliest days of computer programming. In the classic report of Goldstine and von Neumann in 1948 [GvN48], each program is accompanied by the estimates of the time needed for executing each statement and the whole program. The discipline of complexity analysis of programs became better understood after the monumental work of Knuth [Knu68, Knu69, Knu72]. In this work Knuth comprehensively presented a large amount of techniques for designing and analyzing programs. In particular, he illustrated in detail how we can use the techniques of discrete mathematics to analyze the computational complexity of programs.

Although most of the mathematical techniques required can be carried out in a rea-
sonably systematic way, complexity analysis can be nontrivial and tedious even for very simple programs. For example, for a simple program of half page that randomly inserts nodes into and deletes nodes from a binary tree of size at most 3, Jonassen and Knuth [JK78] spent 15 pages to illustrate the analysis of its average behavior. But most of the analysis should be performed automatically via a computer. Many researchers have, therefore, made numerous efforts to automate the task of complexity analysis.

1.2 Automatic Complexity Analysis of Programs

Automatic complexity analysis has been studied primarily in the context of imperative and functional languages; we first review some of these studies.

1.2.1 Imperative Programs

Cohen and Zuckerman [CZ74] pioneered this area by implementing an interactive system that can estimate program complexity for a subset of Algol 60 [Nau63]. In this system a program is first translated into a symbolic formula expressing the average execution time of the program. A user can interactively provide more information for the variables in the formula. A variable in the formula may represent the average execution time of a basic operation, the probability of a boolean expression evaluating to true, or the number of times a while loop is executed. The provided information can then be used to simplify the symbolic formula.

In Cohen and Zuckerman's system, the probabilities of boolean expressions evaluating to true are treated as constants and are statistically independent. In order to handle boolean expressions more realistically, Wegbreit [Weg76b] presented a system that can formally verify the specification of program complexity based on the method of inductive assertions [Flo67, Hoa69]. Unlike usual formal verification systems, which are concerned with the properties of the values of program variables, the assertions in this system specify the probabilities of the properties of the values of program variables. The system tries to prove that if the input assertion describes the probability distribution of the input data, then the inductive assertions describe the probability distributions of the values taken on by program variables during the computation. Once the inductive assertions are shown to be correct, the average execution time can be computed according to the probability distributions specified in the assertions.
In Wegbreit's verification system, the assertions specify the probabilities of the properties of the values of program variables, which naturally leads to the following difficulty. To keep the sum of the probabilities in an assertion to be 1, the probabilities for the out-branches of a conditional statement are obtained simply by taking conditional probability with respect to the probabilities for the in-branch of the conditional statement. This makes the determination of the probability for the out-branch of the join of the conditional statement problematic because the information about the relative probabilities for the in-branches of the join have been thrown away by the action of taking conditional probability. Ramshaw [Ram79] discovered this difficulty and remedied it by using the assertions that specify the frequencies, instead of the probabilities, of the properties of the values of program variables.

1.2.2 Functional Programs

Wegbreit [Weg75] pioneered automatic complexity analysis of functional programs by implementing a system that can automatically analyze program complexity for a subset of Lisp [McC60]. Since loops are generally realized by recursion in functional programs, the primary contribution of this system is the development of techniques for dealing with recursion. Because the time taken to execute a recursion depends on the depth of the recursion, the execution time of a recursive procedure depends on the size difference between the input of consecutive recursive calls in the recursion. In this system a recursive procedure is first translated into a difference equation [LL59], which expresses either the worst-case or the average-case execution time of the procedure, in terms of the size of the input to the procedure. The system then tries to solve the difference equation and produce a closed-form expression. Like Cohen and Zuckerman's system [CZ74], however, the probabilities of boolean expressions evaluating to true are also treated as constants and are statistically independent in this system.

Hickey and Cohen [HC88] later extended the work of Ramshaw [Ram79] to handle recursion and complex data structures in functional programs. In their work, the probability distributions of data structures are represented by attributed probabilistic grammars, which are context-free grammars augmented with probability information. An attributed probabilistic grammar can be viewed as a generator of parse trees representing data structures of a given program.
Flajolet, Steyaert, Salvy and Zimmermann approach the average-case complexity analysis from another point of view [FS81, FS87, FSZ91]. They try to identify the class of programs and data structures whose analysis can be performed automatically. Based on the uniform distribution assumption, they can apply a set of formal translation rules to translate programs into generating functions. More specifically, each data type definition is translated into a generating function, each of whose coefficients represents the total number of data structures of a certain size in that data type. In the meantime, each procedure definition is translated into a generating function, each of whose coefficients represents the total amount of time needed for executing the procedure on all the input data structures of a certain size. The average execution time with respect to an input data structure of a certain size is, therefore, the ratio of the coefficient in a second type of generating function over the coefficient, corresponding to the same size, in a first type of generating function.

Instead of resorting to difference equations or generating functions, Le Métayer [LM88] presented a system that can analyze the worst-case complexity for the language FP [Bac78] based on the techniques of program transformation. This system first transforms the initial recursive function into a recursive function whose value is the worst-case complexity of the initial function. It then transforms the recursive complexity function into a nonrecursive complexity function based on MacCarthy's recursion induction principle [Mac63]. A main feature of this method is that all these program transformations are performed within the source language.

Rosendahl later presented another system that analyzes the worst-case complexity based on program transformation techniques [Ros89]. Rosendahl's method, however, is performed in the framework of abstract interpretation [CC77], which makes the proof of correctness easier. In this system the translated recursive complexity functions manipulate abstract data rather than concrete data, for example, a representation denoting all lists of a certain length instead of a particular list instance of that length.

Several efforts to deal with more advanced features in functional languages have also been proposed. Shultis [Shu85], Le Métayer [LM87], and Sands [San90] have reported techniques for handling higher-order functions. Wadler [Wad88], Bjerner and Holmström [BH89], and Sands [San90] have described methods for dealing with lazy evaluation.
1.2.3 Logic Programs

One of the key features that distinguish logic programs from imperative and functional programs is nondeterminism. Logic programs are generally nondeterministic, namely, a procedure may have more than one definition and any of them may return a solution to a procedure call. As a result, a procedure in logic programs usually defines a relation, a more general notion than function. In most logic programming languages, nondeterminism is simulated by backtracking [BR75]. Informally, the solution space for a call to a procedure is modeled by a search tree. The solutions to the call are generated by traversing the search tree using backtracking.

The study on automatic complexity analysis of logic programs has been very limited. Kaplan described a method for estimating the average-case complexity of logic programs [Kap88]. However, the class of procedures dealt with in this work is restricted to deterministic procedures. Therefore, the treatment of nondeterminism is not addressed in this work.

Knuth [Knu75] and Purdom [Pur78] have developed algorithms to estimate the size of the search tree generated by a backtracking program. The size of the search tree can be considered to be an estimate of the complexity of the backtracking program. Nevertheless, their algorithms are based on the method of random sampling [Vit84], and often need to run hundreds or thousands of times in order to achieve an acceptable estimate.

To analytically estimate the complexity of a backtracking program, we need to estimate the number of alternative branches at each backtrack point. In the computation of logic programs, this means that we need to estimate the number of solutions produced by each procedure. Lipton and Naughton have presented algorithms for estimating the number of solutions produced by procedures in a logic program [LN90]. However, their algorithms are also based on random sampling and need to be performed at run time.

Not surprisingly, it turns out that both the time complexity and the number-of-solutions complexity of a procedure depend on the size of the input to the procedure. As a result, we also need to keep track of the size of arguments at each procedure entry. In the context of proving the termination of logic programs, a number of researchers have investigated the inference of the linear size relationships among the arguments of a procedure [UVG88, Plii90, VG90, SVG91, VDS91]. This work is intended for application of
proving program termination; therefore, it is generally enough to infer linear size relationships among arguments. In practice, however, many programs reveal nonlinear behavior in the size relationships among arguments. In these cases, a precise complexity analysis should be able to uncover the nonlinearity among the argument sizes.

The discussion above demonstrates that to realize automatic complexity analysis of logic programs, we need new techniques for inferring respectively the argument-size complexity, the number-of-solutions complexity, and the time complexity.

1.3 Applications of Automatic Complexity Analysis

Automatic complexity analysis of programs has applications in many areas. In software engineering, programmers often take advantage of execution profile information to optimize their programs. Several techniques for profiling execution have been proposed [Ing70, Knu71, KS73]. However, analytical methods can indeed provide more complete information for optimizing programs because symbolic formulas are usually more informative than numbers.

In automatic program synthesis and transformation systems [MW71, LCW74, Dar82], executable programs are derived automatically from formal specifications, and more declarative but inefficient programs are transformed automatically into more efficient programs. In these systems, complexity information can be used to identify the most expensive components of a program, and thereby improvements can be conducted more effectively by focusing on these crucial components [Weg76a]. In addition, complexity information can also be employed as a criterion for choosing among several possible alternatives.

In compilers for parallel systems, knowledge about the complexity of different procedures can be applied to guide the partitioning, allocation and scheduling of parallel processes to ensure that the gains from parallelism outweigh the overhead associated with the management of concurrent processes [KS90, RM90, Tic90, ZTD*92].

In deductive databases, information about the number of solutions produced by procedures in a program is used to optimize the query evaluation, for example, to plan the order in which subgoals are evaluated [War81, SG85, Nat86]. In proving program termination, the techniques based on the size relationships among arguments in a procedure are very important [UVG88, Plü90, VG90, SVG91, VDS91].
1.4 Contributions

This dissertation describes research toward automatic complexity analysis of logic programs and its applications. The contributions of this research are as follows.

First, we extend the techniques developed for analyzing complexity of imperative and functional programs to logic programs by handling nondeterminism, namely, the generation of multiple solutions via backtracking. The techniques are based on the automatic solution of difference equations. We present the design and implementation of a (semi)-automatic worst-case complexity analysis system for logic programs. This system can conduct the worst-case analysis for several complexity measures, specifically, argument size, number of solutions, and execution time, for a large class of programs.

Second, we describe an application of such analyses, namely, a runtime mechanism for controlling task granularity in parallel logic programming systems. The performance of parallel systems often starts to degrade when the concurrent tasks in the systems become too fine-grained. Our approach to granularity control is based on time complexity information. With this information, we can compare the execution cost of a procedure with the average process creation overhead of the underlying system to determine at runtime if we should spawn a procedure call as a new concurrent task or just execute it sequentially. Through experimental measurements, we show that this mechanism can substantially improve the performance of parallel systems in many cases.

Third, we develop several source-level program transformation techniques for optimizing the evaluation of logic programs containing finite-domain constraints. These techniques are based on number-of-solutions complexity information. The techniques include planning the evaluation order of subgoals, reducing the domain of variables, and planning the instantiation order of variable values. This application allows us to solve a problem by starting with a more declarative but less efficient program, and then automatically transforming it into a more efficient program. Through experimental measurements we show that these program transformation techniques can significantly improve the efficiency of the class of programs containing finite-domain constraints in most cases.
1.5 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 summarizes the preliminary definitions and notions used in this dissertation. It includes a brief introduction to graph theory and logic programming. Chapter 3 presents a framework for the worst-case complexity analysis of logic programs and gives an overview of a complexity analysis system that implements the framework.

Chapters 4–11 describe in detail the major components of the complexity analysis system. Chapter 4 summarizes a number of preliminary analyses that produce information shared by complexity analyses dealing with distinct complexity measures. Chapter 5 presents the worst-case analysis for argument-size complexity. Chapters 6–9 present the worst-case analysis for number-of-solutions complexity. Chapter 10 presents the worst-case analysis for time complexity. Chapter 11 describes the scope of symbolic manipulations and difference equation solving performed in the system.

Chapter 12 describes the application for controlling task granularity in parallel logic programming systems. Chapter 13 describes the application for optimizing logic programs with finite-domain constraints. Finally, Chapter 14 concludes this research and outlines the directions for future research.
CHAPTER 2
PRELIMINARIES

This chapter summarizes the preliminary definitions and notions used in this dissertation. It first introduces the terminology of graph theory, then gives a brief introduction to logic programming.

2.1 Graph Theory

In this section we briefly introduce the terminology of graph theory. For a more complete account, the reader is referred to general books [Tut84, CL86].

A graph \( G = (V, E) \) consists of a set \( V \) of vertices, and a set \( E \subseteq V \times V \) of edges. The order \( |G| \) of \( G \) is the number of vertices in \( G \). Each edge corresponds to a pair of distinct vertices in \( G \). A graph can be directed or undirected.

In an undirected graph \( G \), the edges are unordered pairs. For each edge \( e = (v, w) \), \( e \) is said to connect (or be incident to) vertices \( v \) and \( w \), and \( v \) and \( w \) are said to be adjacent to each other. The degree \( d_G(v) \) of a vertex \( v \) is the number of edges incident to \( v \). Graph \( G_1 \) in Figure 2.1 is an undirected graph of order 4 with \( d_{G_1}(v) = 2 \), and \( d_{G_1}(w) = 3 \).

In a directed graph, the edges are ordered pairs. For each edge \( e = (v, w) \), \( e \) is said to be leading from \( v \) and coming into \( w \), \( v \) a predecessor of \( w \), and \( w \) a successor of \( v \). The indegree of a vertex \( v \) is the number of edges coming into \( v \), and the outdegree of \( v \) is the number of edges leading from \( v \). Graph \( G_2 \) in Figure 2.1 is a directed graph of order 4 with the indegree of \( w \) being 1, and the outdegree of \( w \) being 2.

A path from \( v_1 \) to \( v_n \) is a sequence of vertices \( v_1, \ldots, v_n \) such that \( (v_i, v_{i+1}) \) is an edge, for \( 1 \leq i < n \). A cycle is a path with all its vertices being distinct except for the first and last vertices, which are the same. Both \( v, w, x \) and \( v, w, y, x, v \) are cycles of graphs \( G_1 \) and \( G_2 \) in Figure 2.1.

An undirected graph is called connected if there is a path from any vertex to any other vertex. Graph \( G_1 \) in Figure 2.1 is a connected graph. A directed graph is said to be
Figure 2.1: Examples of graphs
**strongly connected** if there is a path from any vertex to any other vertex. Graph $G_2$ in Figure 2.1 is a strongly connected graph. A directed graph is called **acyclic** (or **DAG**) if there is no cycle in the graph. Graph $G_3$ in Figure 2.1 is a DAG. A **rooted tree** is a DAG with one distinguished vertex, called the **root**, such that the root is the only vertex that has the indegree 0 and all other vertices have indegree 1. In other words, all the edges in a rooted tree are pointing away from the root, and except for the root, every vertex has exactly one incoming edge. Graph $G_4$ in Figure 2.1 is a rooted tree with the root $v$. A vertex $v$ in a rooted tree is called a **leaf** if it has the outdegree 0, that is, there is no edge leading from $v$. Vertices $x$ and $y$ in graph $G_4$ in Figure 2.1 are leaves of $G_4$. A **branch** from a vertex $v$ in a rooted tree is a maximal path (in the sense of the number of edges in the path) leading from $v$. A branch from $v$ is **finite** if it is from $v$ to a leaf; otherwise, it is **infinite**. In graph $G_4$ in Figure 2.1, the two branches from the root $v$ are $v, w, x$ and $v, w, y$, both of which are finite.

A subgraph of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$. The **induced subgraph** of $G$ by a set of vertices $U \subseteq V$ is a subgraph $H = (U, F)$ such that $F$ consists of all the edges in $E$ both of whose vertices belong to $U$. In Figure 2.1, graph $G_6$ is the subgraph of graph $G_5$ induced by the set of vertices $\{w, x, y\}$.

Let $G = (V, E)$ be an undirected graph and $U$ be a subset of $V$. The graph $G - U$ resulting from the removal of the set of vertices in $U$ is the subgraph induced by $V - U$. In Figure 2.1, $G_6 = G_5 - \{v\}$. The **neighbors** $N_G(v) = \{w \in V \mid (v, w) \in E\}$ of a vertex $v$ is the set of vertices adjacent to $v$. In Figure 2.1, $N_{G_6}(v) = \{w, x, y\}$. The **adjacency graph** $\text{Adj}_G(v)$ of $v$ is the subgraph of $G$ induced by $N_G(v)$. In Figure 2.1, $N_{G_6}(v) = G_6$.

A **clique** $C$ of an undirected graph $G$ is a subgraph of $G$ such that the vertices in $C$ are pairwise adjacent. An **$n$-clique** is a clique of order $n$. In Figure 2.1, $G_5$ is a 4-clique and $G_6$ is a 3-clique. An **$n$-partite graph** is an undirected graph whose vertices can be partitioned into $n$ mutually disjoint sets such that every edge connects vertices from distinct sets. In Figure 2.1, $G_7$ is a bipartite graph with the two mutually disjoint sets $\{v, w\}$ and $\{x, y\}$. A **complete $n$-partite graph** is an $n$-partite graph in which every pair of vertices from distinct sets is adjacent. In Figure 2.1, $G_8$ is a complete bipartite graph with the two mutually disjoint sets $\{v, w\}$ and $\{x, y\}$. 
2.2 Logic Programming

This section gives a brief introduction to logic programming. For a more thorough discussion, the reader is referred to general books [SS86, Llo87, MW88].

Most logic programming languages are based on a subset of first order predicate calculus known as Horn clause logic. Such a language has a countably infinite set of variables, and countable sets of function and predicate symbols; these sets are mutually disjoint. Each function symbol \( f \) and each predicate symbol \( p \) is associated with a unique natural number \( n \), referred to as the arity of the symbol; \( f \) and \( p \) are said to be \( n \)-ary symbols, and written \( f/n \) and \( p/n \) respectively. A 0-ary function symbol is referred to as a constant.

A term in such a language is a variable, a constant, or a compound term \( f(t_1, \ldots, t_n) \), where \( f \) is an \( n \)-ary function symbol and \( t_i \) is a term, for \( 1 \leq i \leq n \). The function symbol \( f/n \) of a non-variable term \( f(t_1, \ldots, t_n) \), \( n \geq 0 \), is called the principal functor of the term. The set of terms of the language is denoted by \( \mathcal{T} \). A term is ground if it contains no variable.

The set of ground terms of the language is called the Herbrand universe of the language and denoted by \( \mathcal{H} \).

An atom \( p(t_1, \ldots, t_n) \) consists of an \( n \)-ary predicate symbol \( p \) and \( n \) terms \( t_i \), for \( 1 \leq i \leq n \), as the arguments of \( p \). An atom is ground if all its arguments are ground terms. A literal is either an atom or the negation of an atom. In the first case, the literal is said to be positive; in the second case, it is negative. A clause is the disjunction of a finite number of literals, and is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a definite clause. The positive literal in a definite clause is its head, and the remaining literals constitute its body. A definite clause with empty body is called a fact. A predicate definition consists of a finite number of definite clauses, all whose heads have the same predicate symbol. A logic program consists of a finite set of predicate definitions. A goal is a Horn clause with only negative literals.

We adhere to the syntax of Edinburgh Prolog [Bow81] and write a definite clause as

\[ p : - q_1, \ldots, q_n \]

read declaratively as "\( p \) if \( q_1 \) and \( \ldots \) and \( q_n \)". Names of variables begin with upper case letters, while names of function and predicate symbols begin with lower case letters. For readability, some compound terms and atoms are written in the infix notation instead of the prefix notation. The empty list is written \([\ ]\), and a nonempty list with head \( H \) and
tail $T$ is written $H . T$ or $[H | T]$, where $./2$ is the principal functor of a nonempty list. Note that head is used to refer to both the head of a clause and the head of a list. In situations where ambiguities may arise, we will explicitly distinguish them.

An example of a logic program is as follows:

\[
\text{append}([], L, L).
\]
\[
\text{append}([H|L1], L2, [H|L3]) :- \text{append}(L1, L2, L3).
\]

Predicate \text{append}/3 consists of two clauses and inductively specifies the relation that its third argument is the list resulted from appending its first and second arguments. The first clause says that if the first argument is the empty list, then the third argument is the same as the second argument. The second clause says that if the first argument is a nonempty list, then the head of the third argument is the same as the head of the first argument, and the tail of the third argument is the same as the list resulted from appending the tail of the first argument and the second argument.

A \textit{substitution} is a mapping from variables to terms that is the identity mapping at all but finitely many points. A substitution $\sigma_1$ is said to be \textit{more general} than a substitution $\sigma_2$ if there is a substitution $\theta$ such that $\sigma_2 = \theta \circ \sigma_1$, where $\circ$ is the function composition operator. Two terms $t_1$ and $t_2$ are said to be \textit{unifiable} if there exists a substitution $\sigma$ such that $\sigma(t_1) = \sigma(t_2)$; in this case, $\sigma$ is said to be a \textit{unifier} for the terms. If two terms $t_1$ and $t_2$ have a unifier, then they have a \textit{most general unifier}, denoted by $\text{mgu}(t_1, t_2)$, that is unique up to variable renaming. Thus, \{X $\rightarrow$ $f(1), Y \rightarrow [2], Z \rightarrow 1$\} is a unifier for terms $[X|Y]$ and $[f(Z), 2]$, and $\text{mgu}([X|Y],[f(Z), 2]) = \{X \rightarrow f(Z), Y \rightarrow [2]\}$.

The operational behavior of logic programs can be described by means of \textit{goal reduction}, usually called \textit{SLD-resolution}. A \textit{computation}, also called \textit{SLD-derivation}, for a goal $G$ with respect to a program $P$ is a (possibly infinite) sequence of goals

\[ G = G_0, \ldots, G_i, G_{i+1}, \ldots \]

such that each goal $G_i$ is reduced to the goal $G_{i+1}$ via a resolution step. Let

\[ G_i = a_1, \ldots, a_j, \ldots, a_n, \]

and

\[ b := b_1, \ldots, b_m \]
be an alphabetic variant of a clause in \( P \) that has no variable in common with any of the goals \( G_0, \ldots, G_i \). If \( \theta \) is the most general unifier of \( a_j \) and \( b_i \), and

\[
G_{i+1} = \theta(a_1, \ldots, a_{j-1}, b_1, \ldots, b_m, a_{j+1}, \ldots, a_n),
\]

then we say the goal \( G_i \) is reduced to the goal \( G_{i+1} \) via a resolution step, and \( a_j \) is said to be the resolved atom. For each variable \( v \) in \( G_{i+1} \), if \( \theta(v) = w \), we say \( w \) is the binding of \( v \) at \( G_{i+1} \). Intuitively, each resolution step corresponds to a procedure call.

Let \( G_0, \ldots, G_n \) be a computation for a goal \( G = G_0 \) with respect to a program \( P \), and let \( \theta_i \) be the unifier obtained when resolving the goal \( G_i \) to obtain \( G_{i+1}, 0 \leq i < n \). If this computation is finite and maximal, that is, a computation in which it is not possible to resolve the goal \( G_n \) with any of the clauses in \( P \), then it corresponds to a terminating computation for \( G \). In this case, if \( G_n \) is the empty goal, denoted by \( \emptyset \), then the computation is said to succeed with answer substitution \( \theta \), where \( \theta \) is the substitution obtained by restricting the substitution \( \theta_0 \circ \cdots \circ \theta_0 \) to the variables occurring in \( G \); otherwise, if \( G_n \) is not the empty goal, then the computation is said to fail. On the other hand, if the computation is infinite, then the computation does not terminate. The success set of a program \( P \) is the set of ground atoms \( A \) such that there is a successful computation for \( A \) with respect to \( P \). Let \( p(\tilde{f}) \) be a resolved atom in some computation of a goal \( G \) with respect to a program \( P \), then we say that \( p(\tilde{f}) \) is a call that arises in a computation of \( G \).

As an example, consider the following program [Llo87]:

1. \( p(X, Z) :\neg q(X, Y), p(Y, Z). \)
2. \( p(X, X). \)
3. \( q(a, b). \)

A computation of the goal \( p(R, b) \) with respect to this program is shown in Figure 2.2. For each resolution step, the resolved atom is underlined, and the clause chosen to unify with the resolved atom is specified by the number appearing on the left hand side of the arrow denoting the step. Therefore, this computation is said to succeed with the answer substitution \( \theta = \{ R \rightarrow a \} \); namely, this computation successfully terminates with the variable \( R \) being bound to the constant \( a \).

The resolved atom in a goal is chosen via computation rules. A computation rule is a mapping from goals to atoms such that the value of the mapping for a goal is the resolved
atom in the goal. Given a computation rule, we can represent the computations for a goal $G$ with respect to a program $P$ as a rooted tree, called a search tree, also called SLD-tree. Each vertex in the search tree corresponds to a goal; in particular, the root corresponds to the goal $G$. There is an edge from a goal $G_i$ to a goal $G_j$ if $G_i$ can be reduced to $G_j$ via a resolution step under the particular computation rule. Thus, each edge leading from a goal is associated with a clause whose head is unifiable with the resolved atom in the goal, and each branch from the root corresponds to a computation of the goal $G$. Generally, different computation rules generate different search trees. Figures 2.3 and 2.4 show the search trees for two different computation rules for the example above. In Figure 2.3, the rule that always chooses the leftmost atom in a goal as the resolved atom is used, while in Figure 2.4, the rule that always chooses the rightmost atom in a goal is used.
Figure 2.3: An example of a finite search tree

Notice that the tree in Figure 2.3 is finite, but the tree in Figure 2.4 is infinite.

Given a search tree for a goal G with respect to a program P, the solutions of the goal G are exactly the answer substitutions associated with the successful branches in the search tree. Therefore, the search for the solutions involves traversing the search tree to find the successful branches. Despite differences of search trees, different computation rules generate the same set of answer substitutions in their search trees; consequently, the soundness of a computation is independent of the computation rule used [Llo87]. On the other hand, because the size of a search tree is decided by its computation rule, computation rules play an important role on the efficiency of the computations. Most logic programming systems adopt the computation rule that always chooses the leftmost atom
Figure 2.4: An example of an infinite search tree
in a goal as the resolved atom; accordingly, our complexity analyses are also primarily based on this computation rule.

A search rule is a strategy for traversing search trees to find successful branches. Search rules do not change search trees, but they may affect the order in which the solutions are produced. In particular, when there are infinite branches in a search tree, some or all of the solutions may not be produced if the search is trapped in an infinite branch and cannot traverse the remainder of the search tree. Most logic programming systems adopt the search rule that traverses search trees in a depth-first manner; hence, our complexity analysis are also primarily based on this search rule.

Using the depth-first search rule, the search will go along a branch until the corresponding computation succeeds or fails. In the first case, a solution is found and produced, and the search will backtrack to the predecessor of the current goal and continue to traverse the other alternative branches. In the second case, the search will directly backtrack to the preceding goal to traverse the other branches. When the search proceeds along an infinite branch, we have a nonterminating computation; in this case, the search for the parts of the tree after the infinite branch can never be performed. Therefore, a precise complexity analysis should ignore the parts of the tree after the infinite branch. However, since the infiniteness of computations is an undecidable property, our complexity analyses will make a conservative estimation by considering all the branches in the search tree.

In the discussions above, we have not specified the order in which the edges leading from a vertex in a search tree are traversed. Such an order can be reduced to an ordering on the clauses in the program. Since our complexity analyses will consider all the branches in the search trees, this ordering is generally not crucial. However, when the language analyzed contains system predicates like cut ('!') in Prolog, which can dynamically affect the search procedure on the search tree, we can often take advantage of the ordering on the clauses to obtain a more practical analysis. In these cases, we will use the ordering that chooses the clauses in the textual top-bottom order, as adopted in Prolog.
CHAPTER 3

A FRAMEWORK FOR COMPLEXITY ANALYSIS

This chapter first discusses the issues arising in the complexity analysis of logic programs. It then presents a framework for the worst-case complexity analysis of logic programs. Finally, it gives an overview of a complexity analysis system that implements the framework.

3.1 Logic Programs vs. Imperative and Functional Programs

With regard to complexity analysis, there are three major features that distinguish logic programs from conventional programs. First, logic programs are generally nondirectional, namely, there is no rigid sense of "input" or "output" for the argument positions in a procedure. That is because unification is used as the parameter passing mechanism in logic programs. An argument position in a procedure call is said to be an input position if the argument occurring at that position is ground, that is, there is no variable in that argument. Otherwise, we say it is an output position. As an example, consider the behavior of the following procedure:

\[ \text{father\_son(ted, derek).} \]

This procedure simply states the fact that Ted is Derek's father and Derek is Ted's son. If this procedure is invoked by the call 'father\_son(ted, X)', which asks who Ted's son is, and which also indicates that the first argument position is an input position since the argument ted is ground, while the second argument position is an output position since X is a variable, then this procedure will return derek as the output binding to the variable X. On the other hand, if this procedure is invoked by the call 'father\_son(X, derek)', which inquires who Derek's father is, and which also shows that the second argument

\(^1\)We will follow the terminology used by Reddy [Red84].
position is an input position since the argument derek is ground, while the first argument position is an output position since X is a variable, then this procedure will return ted as the output binding to the variable X. Notice that the same procedure definition can be invoked by two calls with distinct input and output characterizations. Informally, each input and output characterization associated with a call to a procedure is referred to as a mode of that procedure. Therefore, we can generally associate a set of modes with each procedure in a logic program, whereas every procedure in a conventional program can be considered to have only a single mode.

Second, logic programs are generally nondeterministic, that is, a procedure may have more than one definition and any of them may return a solution to a procedure call. As an example, consider the behavior of the following procedure:

\begin{verbatim}
father_son(ted, derek).
father_son(ted, dustin).
\end{verbatim}

This procedure has two definitions and states the fact that Ted is the father of both Derek and Dustin, and both Derek and Dustin are sons of Ted. If this procedure is invoked by the call ‘father_son(X, derek)’, or the call ‘father_son(X, dustin)’, namely, the second position is the input position and the first is the output position. then only one of the definitions can successfully complete the execution and ted will be returned as the output binding to the variable X in any of the two cases. On the other hand, if this procedure is invoked by the call ‘father_son(ted, X)’, namely, the first position is the input position and the second is the output position, then both of the definitions can successfully complete the execution, and both derek and dustin will be returned as the output binding to the variable X. We say a mode of a procedure is determinate if the relation induced by this mode is a function (namely, for each input at most one solution is produced); otherwise, we say it is indeterminate.

Third, the output arguments of a procedure in a logic program may be nonground, that is, the arguments at output positions may contain variables when a call to the procedure successfully returns. The variables in the output arguments may later unify with other terms via other procedure calls. As an example, consider the behavior of the following procedure:
lookup(Name, [(Name,Number)|PhoneBook], Number).
lookup(Name, [(Name1,Number1)|PhoneBook], Number) :-
    Name \== Name1, lookup(Name, PhoneBook, Number).

This procedure defines the relation `lookup(name, phonebook, number)`, which is true if the entry under the name `name` in the telephone book `phonebook` has the telephone extension `number`. If this procedure is invoked by the call `lookup(ted, [(ted,3541), (derek,7326)|P], X)`, which asks what Ted's extension number is, then 3541 will be returned as the output binding to the variable `X`. On the other hand, if this procedure is invoked by the call `lookup(dustin, [(ted,3541), (derek,7326)|P], 2163)`, then since the pair `(dustin,2163)` does not appear in the original telephone book, after traversing the current entries of the book, variable `P` will unify with the term `[(dustin,2163)|PhoneBook]`, and thus `(dustin,2163)` will be inserted into the book. Notice that in both cases the telephone book maintained is an incomplete list. This incomplete list contains a list of `(name, number)` pairs and has a variable as its tail, which may later unify with the empty list, another list, or any other term. Nevertheless, this procedure shows that both the retrieval and insertion operations on the telephone book can be defined in logic program in a unified way. We say a mode of a procedure is `definite` if for every call in that mode, whenever it successfully returns, it only produces ground output arguments; otherwise, we say it is `indefinite`.

With regard to nondirectionality, we can in fact handle each mode of a procedure separately because in practice each procedure has only a small number of modes. Given the specifications of the modes for the procedures exported by a module, it is possible to infer the set of modes associated with each procedure in the module. The automatic inference of mode information has already been well studied [Red84, Mel85, Bru84, MU87, Deb89]. In the system presented in this dissertation, however, we require the users to provide mode information using annotations. We plan to incorporate the automatic inference of mode information into the system in the future.

Thus, we are left to consider the treatment of nondeterminism and nongroundness (or indefiniteness). In this dissertation, we only address the issues arising in dealing with nondeterminism. The handling of nongroundness is left to future research. In most logic programming languages, nondeterminism is simulated by backtracking; therefore, we mainly investigate computational complexity of programs executed using backtracking.
3.2 The Framework

Loops in logic programs are usually realized by recursion. This is due to the single assignment nature of the language, which makes iterative constructs such as "while" unnatural, since there is no loop index that can be updated during iteration. The time complexity of a recursive predicate generally depends on the depth of the recursion during the computation, which in turn depends on the size of the input. As an example, consider the following program, which permutes a list of elements given as its first argument and returns the result as its second argument:

```
perm([], []).
perm([X|Y], [R|Rs]) :- select(R, [X|Y], Z), perm(Z, Rs).
```

```
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

Given the empty list [] as its first (input) argument, perm/2 simply returns the empty list as its second (output) argument. Alternatively, given a nonempty list [X|Y] as its input argument, perm/2 generates a permutation of [X|Y] by nondeterministically selecting an element R of [X|Y] via the predicate select/3, permuting the remainder Z of the input list into a list Rs, and returning the list [R|Rs] as its output argument.

The time complexity of the predicate perm/2 depends on the number of times the literal perm/2 in the clause body is recursively called, which in turn depends on the length difference between the input list [X|Y] to the clause head and the list Z to the literal. The larger the difference in length, the smaller the number of recursive calls. To know the length of the list Z in the literal perm/2, which is equal to the length of the third (output) argument of the literal select/3, we need to know the length relationships between the input and output arguments of the literal select/3, the length of whose input argument is in turn equal to the length of the input list [X|Y] in the clause head. Therefore, for each clause we need to infer the size of arguments at each literal entry and exit, relative to the size of the input arguments in the clause head. To this end, for each predicate we also need to infer the size of its output arguments with respect to the size of its input arguments.
Due to nondeterminism, the literal \texttt{select/3} in the clause body may produce multiple solutions; in such case, the literal \texttt{perm/2} will be recursively called once for each of the solutions via backtracking. As a result, in addition to the size of the input, the time complexity of \texttt{perm/2} also depends on the number of solutions generated by the literal \texttt{select/3}. Consider the definition of \texttt{select/3}. Since each element in the input list is selected once by predicate \texttt{select/3}, it turns out that the number of solutions generated by a call to \texttt{select/3} depends on the number of times the literal \texttt{select/3} in the clause body is recursively called, which again depends on the length difference between the input list to the clause head and the input list to the literal. Consequently, for each clause we need to infer the number of solutions generated by each literal in the clause body. To this end, for each predicate we also need to infer the number of solutions generated with respect to the size of its input arguments.

Our framework for automatic worst-case complexity analysis of logic programs consists of the following steps:

1. Compute the worst-case upper bound on the relative size of arguments at each literal entry and exit for each clause, and compute the worst-case upper bound on the size of the output arguments in terms of the size of the input arguments for each predicate.

2. Based on the argument size information, compute the worst-case upper bound on the number of solutions generated by each literal in each clause, and compute the worst-case upper bound on the number of solutions generated by each predicate in terms of the size of its input arguments.

3. Based on the argument size and the number of solutions information, compute the worst-case upper bound on the time complexity for each clause and predicate in the program.

In this framework the complexity analyses for distinct measures share several common features. First, the predicates in the program are processed in such a way that the analyses for a callee are performed before the analyses for its caller. If two predicates call each other, they are processed at the same time. Second, the complexity function for a predicate is derived from the complexity functions of its clauses by using the information about the mutual exclusion relationships between its clauses. Third, the complexity function for
a clause is inferred based on the data dependency relationships between its variables and between its literals. Fourth, the complexity functions for recursive predicates are in the form of difference equations and are transformed into closed form functions where possible using difference equation solving techniques. These common features allow the analyses to be performed in a unified framework that simplifies proofs of correctness and the implementation of the algorithms.

3.3 Overview of an Implementation

CASLOG (Complexity Analysis System for LOGic) is an implementation of the framework described in last section. CASLOG can perform the worst-case complexity analyses for several complexity measures, specifically, argument-size complexity, number-of-solutions complexity, and time complexity, for the class of programs whose predicates have definite modes.

CASLOG consists of five major components: a preprocessor, an argument-size complexity analyzer, a number-of-solutions complexity analyzer, a time complexity analyzer, and a difference equation solver. The organization of CASLOG is shown in Figure 3.1.

The preprocessor reads program predicates and declarations, and prepares preliminary information for various complexity analyses. This includes caller-callee relationships between the predicates of a program, mutual exclusion relationships between the clauses of a predicate, and data dependency relationships between the variables and between the literals of a clause.

The argument-size complexity analyzer infers the argument size complexity function associated with each clause and predicate; the number-of-solution complexity analyzer infers the number of solutions complexity function associated with each clause and predicate; and the time complexity analyzer infers the time complexity function associated with each clause and predicate.

The difference equation solver receives complexity functions expressed as difference equations from various complexity analyzers and solves them if possible. It returns closed form functions to complexity analyzers if the difference equations are solvable; otherwise, the infinity, denoted by $\infty$, is returned as a conservative solution.
logic program

Preprocessor

Argument-Size Complexity Analyzer

Number-of-Solutions Complexity Analyzer

Time Complexity Analyzer

argument-size complexity

number-of-solutions complexity

time complexity

Difference Equation Solver

Figure 3.1: Organization of CASLOG
3.4 Summary

This chapter discussed the issues regarding nondirectionality, nondeterminism, and non-groundness that arise in the complexity analysis of logic programs. It presented a framework for the worst-case complexity analysis of logic programs that can handle nondeterminism or generation of multiple solutions via backtracking. It also gave an overview of CASLOG—a complexity analysis system—that implements the framework. The remainder of this dissertation will describe the components of CASLOG in more detail.
CHAPTER 4
THE PREPROCESSOR

One of the functions of the preprocessor is reading in program predicates and declarations. CASLOG can handle a subset of Prolog. The list of built-in predicates handled by CASLOG is included in Appendix A. In addition to such predicates and user-defined predicates, the input programs to CASLOG may also contain three types of declarations: mode, measure, and domain declarations. This chapter will describe the syntax and usages of these declarations. The preprocessor also prepares preliminary information for various complexity analyses. The preliminary information includes caller-callee relationships between the predicates of a program, mutual exclusion relationships between the clauses of a predicate, and data dependency relationships between the argument positions of the literals of a clause and between the literals of a clause. This chapter will discuss the analyses for this preliminary information.

4.1 Mode Declarations

For the computations corresponding to a class of initial goals of interest, when a literal of a predicate is the resolved atom, each of its arguments is either a ground term or a nonground term. A mode of a call (or resolved atom) is an assertion about which of its arguments are ground terms and which are nonground terms. In CASLOG we require that each predicate is associated with a single mode; that is, in terms of the ground and nonground characterization of its arguments, there is only one way the predicate can be called. Thus, if a predicate are called in \( n \) different modes in a program, the programmer should rename the definition of the predicate into \( n \) different versions such that each of them corresponds to a different mode.

A mode declaration designates each argument position of a predicate as an input or output position, depending on whether or not it is bound to a ground term when that predicate is called. In the current implementation of CASLOG, mode declarations are
mandatory: every predicate in the program must have an associated mode declaration. The requirement for mode declarations can be removed if we can automatically infer the mode information. Automatic mode inference has been well studied [Red84, Mel85, Bru84, MU87, Deb89]. Given the specifications of the modes for the initial goals of interest, the techniques proposed in these studies can be applied to automatically generate mode declarations.

The syntax of a mode declaration is as follows:

```
:- mode(p/n, mode_list)
```

where `p/n` is a predicate symbol and `mode_list` is a list of `n` mode symbols. The set of mode symbols is `{+, -}`, where `+` denotes an input position, while `-` denotes an output position. As an example, consider the list permutation program:

```prolog
perm([], []).
perm([Y|Ys], [R|Rs]) :- select(R, [Y|Ys], Z), perm(Z, Rs).
```

```prolog
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

The appropriate mode declarations for the predicates in the program can be given as follows:

```
:- mode(perm/2, [+,-]).
:- mode(select/3, [-,+,-]).
```

The first declaration specifies that the first position of `perm/2` is input and the second position is output. The second declaration specifies that the second position of `select/3` is input, while the first and the third positions are output.

A mode of a predicate is said to be definite if for each call to the predicate in that mode, whenever it successfully returns, the terms appearing at the output positions are always ground. CASLOG can handle predicates with definite modes. In addition, we also require that every clause is acyclically well-moded [Red84]. A clause is said to be acyclically well-moded if there exists an irreflexive partial order over the body literals so that
1. every variable appearing in an input position in a body literal \( l \) also appears either in an input position in the clause head, or in an output position of some other body literal preceding \( l \); and

2. every variable occurring in an output position in the clause head also appears either in an input position in the head, or in an output position in the body.

Since Prolog uses a leftmost computation rule, that is, executes body goals from left to right, we require that the clauses in the program are acyclically well-moded under the left-to-right order of the body literals. The clauses in both \( \text{perm}/2 \) and \( \text{select}/3 \) in the example above are acyclically well-moded under the left-to-right order. However, the following clause is not acyclically well-moded under the left-to-right order:

\[
\text{:- mode(p/2, [+,-]).}
\]
\[
\text{:- mode(q/2, [+,-]).}
\]
\[
\text{:- mode(r/3, [+,-]).}
\]
\[
\text{:- p(X, Y) :- q(X, V), r(V, W, Y).}
\]

This is because variable \( W \) appears in an input position of literal \( r/3 \), but it appears neither in an input position in the clause head nor in an output position of literal \( q/2 \). Intuitively, given that all the literals in an acyclically well-moded clause have a definite mode, any call to the clause behaves as follows. If all the input arguments of the call are ground, then

1. all the input arguments of the clause head are given ground,

2. all the input arguments of each body literal become ground when the literal is called,

3. all the output arguments of each literal become ground when the call to the literal successfully returns, and

4. all the output arguments of the clause head become ground when the call to the clause successfully returns.

The detection of acyclically well-moded clauses under the left-to-right order can be performed using an algorithm (Algorithm 4.1) described in Section 4.6. Because every predicate has a single and definite mode, and every clause is acyclically well-moded under the
left-to-right order, the programs currently handled by CASLOG do not have the features of nondirectionality and nongroundness discussed in Section 3.1.

4.2 Measure Declarations

For the computations corresponding to a class of initial goals of interest, when a literal of a predicate is the resolved atom, the complexity associated with the computation of the resolved atom usually depends on the size of its input arguments. Various measures (or term norms) can be used to determine the "size" of a term. Let $\mathcal{T}$ denote the set of terms in the language, and $\mathcal{N}_\bot$ denote the set of natural numbers $\mathcal{N}$ augmented with a special symbol $\bot$, denoting "undefined." A size measure is a function $|\cdot| : \mathcal{T} \rightarrow \mathcal{N}_\bot$. Examples of such functions are "list-length," which maps lists to their lengths and all other terms to $\bot$, and "term-size," which maps every term to the number of constants and function symbols appearing in it. Thus, $|[a, b]|_{\text{list-length}} = 2$, and $|[a, b]|_{\text{term-size}} = |a \cdot (b \cdot [])|_{\text{term-size}} = 5$.

The size measures appropriate in a given situation can, in most cases, be determined by examining the types of argument positions [Plis90, VDS91]. The type of an argument position denotes the set of terms that can appear at that argument position during execution, and can be represented by a directed graph, called a type graph. As an example, suppose the type $\tau$ of an argument position is the set of lists of integers, defined recursively as $\tau = \{ \text{ListOfInt} = [] \mid \text{Int}\text{ListOfInt} \}$, meaning a term of type $\tau$ is either the empty list or a list with an integer as its head and another term of type $\tau$ as its tail. Then the type graph for $\tau$ can be represented as the graph shown in Figure 4.1, where $/2$ denotes the principal functor of a nonempty list. The general idea for determining size measures is to use the cycles in the type graphs of argument positions to determine how a predicate recursively traverses its input terms or constructs its output terms, thereby synthesizing the size measures for the argument positions. The function symbols appearing in some cycle of the type graph are referred to as the reflexive constructors, representing the parts of the term on which the predicate recursively operates. For example, $/2$ is a reflexive constructor for the type $\tau = \{ \text{ListOfInt} = [] \mid \text{Int}\text{ListOfInt} \}$. Therefore, the size of a term can usually be determined by measuring the number of reflexive constructors in it.

Size measures based on type information can be defined using the type norm as follows [Plis90]. Let $\tau$ be a type. The type norm under the type $\tau$ is a function $|\cdot| : \mathcal{T} \rightarrow \mathcal{N}$ defined as
Figure 4.1: An example of a type graph

\[
\begin{align*}
|t|_\tau & = 0 \quad \text{if } t \text{ is a constant or a variable} \\
|f(t_1, \ldots, t_n)|_\tau & = 1 + \sum_{i=1}^{n} |t_i|_\tau \quad \text{if } f/n \text{ is a reflexive constructor for } \tau \\
|f(t_1, \ldots, t_n)|_\tau & = \sum_{i=1}^{n} |t_i|_\tau \quad \text{otherwise.}
\end{align*}
\]

For example, for the type \(\tau = \{\text{ListOfInt} = [\_] \mid [\text{Int}|\text{ListOfInt}]\}\), \(|t|_\tau\) gives the list length of the term \(t\). Thus, \(|[]|_\tau = 0\), \(|[a, b]|_\tau = |a \cdot (b \cdot [\_])|_\tau = 2\), and \(|[X,b]|_\tau = 2\).

Type information can be inferred by means of program analysis [Mis84, YS87, PR89, HJ90].

A measure declaration assigns a size measure to each argument position of a predicate. Each such size measure is used to determine the size of the terms at the corresponding argument position for calls to the predicate. At the present state of CASLOG, measure declarations are also mandatory: every predicate must be accompanied by an associated measure declaration.

The syntax of a measure declaration is as follows:

\[
:- \text{measure}(p/n, \text{measure\_list})
\]

where \(p/n\) is a predicate symbol and \text{measure\_list} is a list of \(n\) size measures. In the
current implementation of CASLOG, we abstract general types into the following set of commonly used size measures \( \Gamma = \{ \text{nat}, \text{length}, \text{size}, \text{depth}(f, r), ? \} \), where \( f \) is a function symbol and \( r \) is a list of positive integers. Since we only deal with predicates with definite modes, we can restrict our size measures to ground terms, namely, each of them is a function \( \cdot |_m : \mathcal{H} \to \mathcal{N}_1 \), where \( m \in \Gamma \) and \( \mathcal{H} \) is the Herbrand universe of the language. The definition for the size measures in \( \Gamma \) is given in Figure 4.2.

The size measure \text{nat} is used for an argument position of a predicate \( p \) if the type of this argument position is any set of natural numbers and the complexity of \( p \) depends on the value of the term at this argument position. \( |t|\text{nat} \) determines the value of a ground natural number expression \( t \); thus, \( |2|\text{nat} = 2 \) and \( |3 - 1|\text{nat} = 2 \). For example, consider the following predicate that computes the factorial of a natural number:

\[
\text{fact}(0, 1).
\text{fact}(N, M) :- N > 0, \text{N1 is } N-1, \text{fact(N1, M1)}, M \text{ is } N \times M1.
\]

The type for both argument positions of \text{fact}/2 is the set of natural numbers. Given a natural number as the input argument of a call to \text{fact}/2, the complexity of this call depends on the value of that number.

The size measure \text{length} is used for an argument position of a predicate \( p \) if the type of this argument position is any set of ground lists and the complexity of \( p \) depends on the length of the term at this argument position. \( |t|\text{length} \) determines the length of a ground list \( t \); thus, \( |[a, b]|\text{length} = 2 \). For example, consider the following predicate that appends two lists into a list:

\[
\text{append}([], L, L).
\text{append}([\text{H}], L1, [\text{H}|R]) :- \text{append}(L, L1, R).
\]

The type for all the argument positions of \text{append}/3 is the set of ground lists. Given two ground lists as the input arguments of a call to \text{append}/3, the complexity of this call depends on the length of the list at the first argument position and the length of the output list depends on the length of the two input lists.
If $m$ is $\text{nat}$, then for any ground term $t$,

\[
|t|_m = \begin{cases} 
  t & \text{if } t \text{ is a natural number} \\
  \odot(|t_1|_m, \ldots, |t_n|_m) & \text{if } t = \odot(t_1, \ldots, t_n) \text{ for some arithmetic operator } \odot \\
  \bot & \text{otherwise.}
\end{cases}
\]

If $m$ is $\text{length}$, then for any ground term $t$,

\[
|t|_m = \begin{cases} 
  0 & \text{if } t \text{ is the empty list} \\
  1 + |t_2|_m & \text{if } t = [t_1|t_2] \\
  \bot & \text{otherwise.}
\end{cases}
\]

If $m$ is $\text{depth}(f/n, r)$, where $f/n$ is a function symbol and $r$ is a list of positive integers, then for any ground term $t$,

\[
|t|_m = \begin{cases} 
  0 & \text{if } t \text{ is a constant} \\
  1 + \max \{|t_i|_m \mid i \in r\} & \text{if } t = f(t_1, \ldots, t_n) \text{ and } 1 \leq i \leq n, \text{ for all } i \in r \\
  \bot & \text{otherwise.}
\end{cases}
\]

If $m$ is $\text{size}$, then for any ground term $t$,

\[
|t|_m = \begin{cases} 
  1 & \text{if } t \text{ is a constant} \\
  1 + \sum_{i=1}^{n}(|t_i|_m) & \text{if } t = f(t_1, \ldots, t_n) \\
  \bot & \text{otherwise.}
\end{cases}
\]

If $m$ is `?`, then for any ground term $t$,

\[|t|_m = \bot.\]

Figure 4.2: The definition for some common size measures
The size measure \textit{size} is used for an argument position of a predicate \( p \) if the type of this argument position is any set of ground terms and the complexity of \( p \) depends on the number of constants and function symbols in the term at this argument position. \(|t|_{\text{size}}\) determines the number of constants and function symbols in a ground term \( t \); thus \(|t(a, t(b, \text{nil}, \text{nil}), \text{nil})|_{\text{size}} = 7\). For example, consider the following predicate that flattens nested list into a flat list, namely, a list whose elements do not contain lists:

\[
\text{:- mode}(	ext{flatten/2}, [+,-]).
\]
\[
\text{:- measure}(	ext{flatten/2}, [\text{size}, \text{length}]).
\]
\[
\text{flatten([], []).}
\]
\[
\text{flatten}(X, [X]) :- \text{atomic}(X), X \neq \text{[]}.
\]
\[
\text{flatten}([X|Xs], Ys) :-
\]
\[
\text{flatten}(X, Ys1), \text{flatten}(Xs, Ys2), \text{append}(Ys1, Ys2, Ys).
\]

The type for both argument positions of \text{flatten/2} is the set of ground lists. Given a ground list as the input argument of a call to \text{flatten/2}, the complexity of this call depends on the number of constants and function symbols in that list.

The size measure \textit{depth}(f, r) is used for an argument position of a predicate \( p \) if the type of this argument position is any set of ground terms with a reflexive constructor \( f \) and \( f \) recursing at argument positions in \( r \), and the complexity of \( p \) depends on the height of the tree representation of the term at this argument position by restricting the measuring to the set of argument positions in \( r \). \(|t|_{\text{depth}}(f, r)\) determines the height of the tree representation of a ground term \( t \) by restricting the measuring to the set of argument positions in \( r \); thus, \(|t(f(g(a)), t(b, \text{nil}, \text{nil}), \text{nil})|_{\text{depth}}(t/3, [2,3]) = 2\). For example, consider the following predicate that traverses a ground term represented as a binary tree.

\[
\text{:- mode}(	ext{traverse/1}, [+]).
\]
\[
\text{:- measure}(	ext{traverse/1}, [\text{depth}(t/3, [2,3])]\).
\]
\[
\text{traverse}([]).
\]
\[
\text{traverse}(t(X,L,R)) :- \text{traverse}(L), \text{traverse}(R).
\]

The type for the argument position of \text{traverse/1} is the set of ground terms that have a reflexive constructor \( t/3 \) and recurse at the second and the third argument positions. Given a ground term in the type as the input argument of a call to \text{traverse/1}, the
complexity of this call depends on the depth of the tree representation of that term by restricting the measuring to the argument positions in \([2, 3]\). Notice that the measure length is a special case of the measure depth\((f, r)\), and can be regarded as the measure depth\((./2, [2])\). For example, \(|[a, b]|_{\text{length}} = |[a, b]|_{\text{depth}(./2, [2])} = 2\). We choose to have a separate measure for length because manipulating length is more efficient than manipulating depth\((./2, [2])\) and the measure length arises frequently in practice.

The size measure \(\top\), indicating "irrelevant," returns \(\bot\) for any given ground term. It is usually used for an argument position of a predicate \(p\) when the size of the term at that argument position is irrelevant to the complexity of \(p\).

We will assume that all the measure declarations given by the user are correct in the sense that when the term at an argument position of a predicate \(p\) becomes ground, it is of a type consistent with its associated size measure (namely, it is a natural number if the measure is \(\text{nat}\), a ground list if the measure is length, and so on), and the complexity of \(p\) can be represented as a function in terms of its size (possibly also the size of the terms at other argument positions) under its associated measure. If there are incorrect measure declarations in the program, the complexity inferred may also be incorrect. If at run time every ground term occurring at an argument position is consistent with the type of the position, then the size measured at run time will always be a natural number. This allows us to discuss various (argument-size, number-of-solutions, and time) complexities over the domain \(\mathcal{N}_{\bot, \infty}\), where \(\mathcal{N}_{\bot, \infty}\) denotes the set of natural numbers \(\mathcal{N}\) augmented with two special symbols \(\bot\), denoting undefined, and \(\infty\), denoting infinity. Because of that, we can define various complexity functions as a function from \(\mathcal{N}_{\bot, \infty}\) to \(\mathcal{N}_{\bot, \infty}\). In addition, hereafter, when we say \(\theta\) is a substitution such that \(\theta(t)\) is ground for a term \(t\) at an argument position with a size measure \(m\), we really mean it is a substitution such that \(\theta(t)\) is ground and \(|\theta(t)|_m\) is defined.

### 4.3 Domain Declarations

The domain (or type) of an argument position denotes the set of terms that can appear at that argument position during execution. In many situations, the clauses of a predicate consist of constraints, namely, literals involving comparison operators \((=, \neq, >, \geq, <, \leq)\), over finite domains. In these situations, for the reasons that will be discussed in the following chapters, we can often use domain information to improve complexity analysis.
Domain information can be obtained by means of type inference [Mis84, YS87, PR89, HJ90].

A domain declaration specifies a finite domain for each argument position of a predicate. In the current implementation of CASLOG, domain declarations are optional: they are usually used for predicates whose clauses consist of constraints over finite domains.

The syntax of a domain declaration is as follows:

```
:- domain(p/n, domain_list)
```

where \( p/n \) is a predicate symbol and \( domain_list \) is a list of \( n \) domain expressions. A domain expression is either a list of constants, denoting the elements of a domain, or a natural number interval, written \( I-h \), denoting the set of natural number between (and including) the number \( I \) and the number \( h \).

As an example, consider the following predicate that specifies a set of precedence constraints among the jobs in a project:

```
:- mode(precedence_constraints/4, [+,-,-,-]).
:- measure(precedence_constraints/4, [nat,nat,nat,nat]).
precedence_constraints(A,B,C,D) :-
    B >= A + 1, C >= A + 1, D >= B + 3, D >= C + 2.
```

where variables \( A, B \) and \( C \) are the starting time of the jobs in the project, and variable \( D \) is the starting time of a dummy job, denoting the end of the project. If the project is planned to be completed within 4 units of time, then an appropriate domain declaration for this predicate can be either

```
:- domain(precedence_constraints/4, [0-4,0-4,0-4,0-4]).
```

or

```
:- domain(precedence_constraints/4,
            [[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4]]).
```

both of which specify that the domain for the variables \( A, B, C \) and \( D \) is the set of natural numbers between 0 and 4.

CASLOG currently requires that domain declarations be associated with predicate definitions. However, a predicate involving finite-domain constraints may be called in
several different contexts, and thereby the corresponding domains may be different. In these cases, the programmer should rename the definition of the predicate into different versions such that each of them corresponds to a different context or domain. For example, the predicate `precedence_constraints/4` may be called with the domain `0-4` at one point of execution, and is called with the domain `0-5` at another point of execution. To make use of domain information in both situations, we need to provide two versions of definition for `precedence_constraints/4`. Domain declarations can be generated automatically and be made to be associated with calls instead of predicate definitions after automatic type inference is incorporated into CASLOG.

### 4.4 Call Graph

The **call graph** for a program is a directed graph that represents the caller-callee relationships between the predicates in the program. Each vertex in the call graph denotes a predicate symbol. There is an edge from a vertex `p` to a vertex `q` if a literal with predicate symbol `q` appears in the body of a clause defining the predicate `p`.

As an example, consider the following program:

```
    p :- q.
    q :- r, s.
    r :- q.
    s.
```

The call graph for this program is shown in Figure 4.3(a).

A body literal in a clause is called a **recursive** literal if its predicate symbol is part of a (directed) cycle that contains the predicate symbol of the clause head in the call graph. A clause is called **nonrecursive** if no body literal is recursive, and is called **recursive** otherwise. A recursive clause is called **direct recursive** if it contains recursive literals and all the recursive literals have the same predicate symbol as the clause head; otherwise, it is called **indirect recursive**. A predicate is called **(direct or indirect) recursive** if it contains (direct or indirect) recursive clauses.

The caller-callee relationships between predicates reveal the computational dependencies between predicates. To understand the behavior of a caller, we usually need to first understand the behavior of its callees. This naturally imposes an order on the predicates of
Figure 4.3: An example of a call graph and a strongly connected components graph

A strongly connected component of a directed graph is a maximal subset of vertices such that its induced subgraph is strongly connected. Recall that, in a strongly connected directed graph, there is a path from any vertex to any other vertex. A linear time algorithm for finding the strongly connected components of a directed graph is described by Tarjan [Tar72].

To generate an appropriate order for analyzing strongly connected components, we transform the call graph into a directed graph, called strongly connected components graph. Each vertex in the strongly connected components graph corresponds to a strongly connected component in the call graph. There is an edge from a vertex \( v \) to a vertex \( w \) in the strongly connected components graph if, in the call graph, there is an edge from a vertex...
in the component corresponding to $v$ to a vertex in the component corresponding to $w$. The strongly connected components graph is acyclic because no cycle in the call graph can appear in two distinct strongly connected components. A desired order can now be generated by topologically sorting the strongly connected components graph. Such an order is called a topological order of the call graph. A linear time algorithm for topologically sorting a directed graph is described by Knuth [Knu72].

As an example, consider again the program given above. The strongly connected components graph transformed from the call graph for the program is shown in Figure 4.3(b). From the strongly connected component graph, an appropriate order for analyzing the predicates in this program is to analyze $s$ first, then $q$ and $r$, and finally $p$.

4.5 Mutual Exclusion between Clauses

Because logic programs can be nondeterministic, there may, in general, be more than one clause whose head can unify with a call. The results of complexity analysis can be greatly improved if situations where this cannot happen, namely, where clauses are mutually exclusive, are detected and dealt with specially.

Informally, two clauses of a predicate are mutually exclusive if at most one of the clauses can succeed for any call to that predicate. The following three propositions, modified from the work of Debray and Warren [DW89], provide some rules for detecting mutual exclusion between clauses.

**Proposition 4.5.1** Two clauses of a predicate are mutually exclusive if there is a subset of input argument positions in their heads such that the terms appearing at the positions are not unifiable. □

A predicate is called a test if all its argument positions are input positions. Two $n$-ary tests $p$ and $q$ are complementary if $p(\bar{t})$ and $q(\bar{t})$ cannot succeed simultaneously for any $n$-tuple $\bar{t}$ of ground terms.

**Proposition 4.5.2** Let two clauses of a predicate $p$ be in the form of

\[
p(X, Y_1) := G_{11}, \quad r_1(Z), \quad G_{12}.
\]

\[
p(X, Y_2) := G_{21}, \quad r_2(Z), \quad G_{22}.
\]
where $\vec{X}$ and $\vec{Y}_i$ are respectively the tuple of terms at the input and output positions in the heads, and each of the $G_{ij}$ consists of zero or more literals. The two clauses are mutually exclusive if every literals in $G_{11}$ and $G_{21}$ are tests, and $r_1$ and $r_2$ are complementary tests.

As an example, consider the following predicate $\text{part/4}$, which partitions a list of numbers into two sublists based on the pivot number given at the first argument position.

\begin{verbatim}
:- mode(part/4,[+,-,-,+-,+-,-]).
:- measure(part/4,[void,length,length,length]).
part(F,[],[],[]).
part(F,[X|Y],[X|Y1],Y2) :- X =< F, part(F,Y,Y1,Y2).
part(F,[X|Y],Y1,[X|Y2]) :- X > F, part(F,Y,Y1,Y2).
\end{verbatim}

The first two clauses can be shown to be mutually exclusive by Proposition 4.5.1 because the two terms $[]$ and $[X|Y]$ appearing at the second (input) argument positions are not unifiable. The last two clauses can be shown to be mutually exclusive by Proposition 4.5.2 since the two tests ‘$X =< F$’ and ‘$X > F$’ in their bodies are complementary.

The predicate cut in Prolog is often used to increase the degree of mutual exclusion between the clauses of a predicate. The following proposition states its effect in this respect.

**Proposition 4.5.3** Two clauses of a predicate are mutually exclusive if there is a cut in the body of the textually antecedent clause. □

Operationally, if the textually antecedent clause contains a cut and the computation for it succeeds, then the system will prune the computation for the other clause.

The clauses of a predicate can be partitioned into “clusters” such that two clauses are in the same cluster if and only if they are not mutually exclusive. We call each such cluster a mutually exclusive cluster. The partition of the clauses of a predicate can be done as follows. We first construct an undirected graph. Each vertex in the graph denotes a clause. There is an edge between two vertices $p$ and $q$ if $p$ and $q$ are not mutually exclusive. Then each mutually exclusive cluster corresponds to a maximal connected subgraph of this graph. The maximal connected subgraphs of an undirected graph can be found in linear time using a depth-first traversal of the graph.
As an example, consider the following clauses:

\[- \text{mode}(p/2, [+,-]).\]

1. \(p(a, x).\)
2. \(p(a, y).\)
3. \(p(b, z).\)
4. \(p(c, y).\)
5. \(p(c, z).\)

These clauses can be partitioned into three clusters: \(\{1, 2\}, \{3\}, \text{ and } \{4, 5\}\). For any call to the predicate, only clauses in one of the three clusters can succeed.

Given the set of mutually exclusive clusters of clauses of a predicate, complexity analysis can be performed separately for each cluster, and the total complexity for the predicate is given in terms of the complexity of the most expensive cluster.

### 4.6 Data Dependency Graphs

Operationally, given two argument position \(a\) and \(b\) in a clause, \(b\) is dependent on \(a\) if the variable bindings generated at \(a\) are used to construct the term occurring at \(b\). To be precise, \(a\) is an input position in the head or an output position in a body literal, \(b\) is an output position in the head or an input position in a body literal, and the terms occurring at positions \(a\) and \(b\) have variables in common.

The data dependencies between argument positions can be represented by a DAG, called an argument dependency graph. Each vertex in the graph denotes an argument position. There is an edge from an argument position \(a\) to an argument position \(b\) if \(b\) is dependent on \(a\). Notice that the vertices denoting the input positions in the head have no predecessor, and the vertices denoting the output positions in the head have no successor. Examples of argument dependency graphs are given in Figure 4.4. In the graphs each circle represents a vertex, \(h_i\) denotes the \(i^{th}\) argument position in the head, and \(b_{ij}\) denotes the \(i^{th}\) argument position in the \(j^{th}\) body literal.

Argument dependency graphs are induced by the control strategy of the system. For programs containing only predicates with definite modes and clauses that are acyclically well-moded in the left-to-right order, argument dependency graphs can be easily con-
:- mode(perm/2, [+,-]).

perm([], []).

perm([X|Y], [R|Rs]) :- select(R, [X|Y], Z), perm(Z, Rs).

Figure 4.4: Examples of argument dependency graphs and literal dependency graphs

structured using mode information. Basically, we use mode information to determine where each of the variables in a clause becomes bound (or defined). A variable can become bound in an input position in the head or in an output position in a body literal. The other occurrences of the variable, namely, ones in an input position in a body literal or in an output position in the head, merely use the value of the variable. Hence, we can construct an edge from each position that defines the variable to each position that uses the variable. Also, because the output positions of a body literal depend on the input positions of the literal, we construct an edge from each input position of a body literal to each output position of the literal. If the value of any variable is used (in an input position in a literal or an output position in head) but not defined (in an input position in the head or an output position in a literal), then the clause is not acyclically well-moded and the system will print an error message and terminate. An algorithm for constructing the argument dependency graph of a clause is given in Figure 4.5. For more general programs, argument dependency graphs may be inferred via dataflow analysis [CDD85, Deb89].

It is sometimes convenient to abstract an argument dependency graph into a graph
Algorithm 4.1

\textit{arg\_dep\_graph}(C): Construct the argument dependency graph of a clause.

\textbf{Input:} A clause $C: L_0 : - L_1 , \ldots , L_m$.

\textbf{Output:} The argument dependency graph $G$ of $C$.

\textbf{Method:}

begin

$G := (\emptyset, \emptyset)$;

add a vertex for each argument position in a literal in $C$ to $G$;

for each input position $a$ in $L_0$ do

for each variable $v$ appearing in $a$ do mark $v$ becoming bound at $a$ od od

for $i := 1$ to $m$ do

for each input position $a$ in $L_i$ do

for each variable $v$ appearing in $a$ do

for each position $b$ at which $v$ becomes bound do

add an edge from the vertex for $b$ to the vertex for $a$ od od od

for each output position $a$ in $L_i$ do

for each input position $b$ in $L_i$ do

add an edge from the vertex for $b$ to the vertex for $a$ od od

for each variable $v$ appearing in $a$ do mark $v$ becoming bound at $a$ od od

for each output position $a$ in $L_0$ do

for each variable $v$ appearing in $a$ do

for each position $b$ at which $v$ becomes bound do

add an edge from the vertex for $b$ to the vertex for $a$ od od od

return $G$

end

Figure 4.5: An algorithm for constructing the argument dependency graph of a clause
that represents the data dependencies between literals. A literal dependency graph is a DAG. Each vertex in the literal dependency graph denotes a literal. There is an edge from a vertex \( p \) to a vertex \( q \) in the literal dependency graph if, in the argument dependency graph, there is an edge from a vertex corresponding to an argument position in \( p \) to a vertex corresponding to an argument position in \( q \). The head of the clause is treated specially. It corresponds to two vertices: one consists of the input positions in the head, and the other consists of the output positions in the head. Examples of literal dependency graphs are also shown in Figure 4.4, where the vertices in the graphs are represented by rectangles.

### 4.7 Complexity

This section discusses the complexity of the preprocessor. Let \( p \) be the number of predicates in the program, \( c \) the maximum number of clauses in a predicate, \( l \) the maximum number of literals in a clause, \( a \) the maximum arity of a predicate, and \( t \) the maximum term size of a term in the program. These parameters are also listed in Appendix B for later reference.

The program predicates and declarations can be read in in time \( O(p \cdot c \cdot l \cdot a \cdot t) \). The call graph can be constructed in time \( O(p \cdot c \cdot l) \). Let \( m \) be the number of edges in the call graph. Both the algorithm for finding the strongly connected components of the graph and the algorithm for topologically sorting the strongly connected components graph requires time \( O(p + m) \).

The detection of mutual exclusion between the clauses in a predicate can be performed in time \( O(p \cdot c \cdot l \cdot a) \) [DW89]. Let \( n \) be the number of edges in the graph representing the mutual exclusion between clauses. The algorithm for finding the mutually exclusive clusters of clauses of a predicate requires time \( O(c + n) \). To handle \( p \) predicates, it requires time \( O(p \cdot c + p \cdot n) \).

We now consider the construction of argument dependency graphs. In Prolog, since each variable can only become bound in a single literal, the number of positions at which a variable becomes bound is \( O(a) \). Because there are \( O(a \cdot t) \) variables in a literal, constructing edges between argument positions in distinct literals requires \( O(l \cdot a^2 \cdot t) \). Edges between argument positions in the same literal can be constructed in time \( O(a^2) \). Altogether, the algorithm for constructing the argument dependency graph of a clause requires
time $O(l \cdot a^2 \cdot t)$. To handle $p$ predicates, it requires time $O(p \cdot c \cdot l \cdot a^2 \cdot t)$. Literal dependency graphs can be constructed in the same complexity.

In summary, the time complexity of the preprocessor is $O(p \cdot c \cdot l \cdot a^2 \cdot t + p \cdot n + m)$. Constructing the argument dependency graph dominates the time complexity.

4.8 Summary

The preprocessor performs four functions. First, it reads in program predicates and the declarations associated with each predicate. Second, it constructs the call graph for the program, and generates an appropriate order for analyzing predicates. Third, it partitions the clauses of each predicate into mutually exclusive clusters. The analysis for each cluster can then be performed separately, and the most expensive complexity among them is taken to be the complexity for the predicate. Fourth, it constructs the argument dependency graph and the literal dependency graph for each clause. Various complexity analyses for a clause will be performed based on these data dependency graphs.
CHAPTER 5

THE ARGUMENT-SIZE COMPLEXITY ANALYZER

One of the main functions of the argument-size complexity analyzer is the inference of the argument-size complexity functions of predicates. An argument-size complexity function $Sz_p^{(i)}: \mathcal{N}_n \to \mathcal{N}_{n,\infty}$ of a predicate $p$ with $n$ input positions is a function representing a worst-case upper bound on the size of the $i^{th}$ (output) argument position of $p$ in terms of the size of the input positions of $p$. The argument-size complexity analyzer also uses the argument-size complexity functions of predicates to infer the input size of each literal in each clause body. These input sizes are important for the inference of the number-of-solutions complexity and the time complexity of predicates.

We will first discuss how various size measures introduced in Section 4.1 can be extended to handle general terms. We then describe a method for inferring, for each clause, a worst-case upper bound on the size of (the term at) each argument position in a body literal or each output position in the head, relative to the sizes of (the terms at) the input positions in the head. This is followed by a discussion of how a worst-case upper bound on the output size of each predicate can be derived from the worst-case upper bounds on the output size obtained from its clauses. Finally, we give an example and discuss the complexity of the argument-size complexity analyzer.

5.1 Size Measures for General Terms

In Section 4.1, we introduce the following set $\Gamma = \{\text{nat, length, size, depth}(f, r), ?\}$ of measures, where $f$ is a function symbol and $r$ is a list of positive integers. For each $m \in \Gamma$, the size measure $|\cdot|_m$ is defined only on ground terms. But this is not adequate for our purposes because we are doing static analysis and need the ability to handle general terms. Let $\mathcal{T}$ be the set of terms in the language. For any given size measure $|\cdot|_m$, the size properties of general terms can be described using two functions $\text{size}_m: \mathcal{T} \to \mathcal{N}_\perp$ and $\text{diff}_m: \mathcal{T} \times \mathcal{T} \to \mathcal{N}_\perp$ that are defined in terms of $|\cdot|_m$. The function $\text{size}_m(t)$ defines the
size of a term $t$ under a measure $m$:

$$size_m(t) = \begin{cases} 
  n & \text{if } |\theta(t)|_m = n \text{ for every substitution } \theta \text{ such that } \\
  & \theta(t) \text{ is ground} \\
  \bot & \text{otherwise.}
\end{cases} \quad (5.1)$$

Thus, $size_{\text{length}}([L, a]) = 2$ since $|\theta([L, a])|_{\text{length}} = 2$ for every substitution $\theta$ such that $\theta([L, a])$ is ground, but $size_{\text{length}}([a|L]) = \bot$ because $|\theta([a|L])|_{\text{length}}$ is different for substitutions $\theta$ that bind $L$ to ground lists of different lengths. A detailed realization of the size functions for the measures in $\Gamma$ is given in Figure 5.1.

**Proposition 5.1.1** For every function $size_m$ defined in Figure 5.1, if $size_m(t)$ is defined for a term $t$, then $size_m(t) = |\theta(t)|_m$ for every substitution $\theta$ such that $\theta(t)$ is ground. $\square$

The function $diff_m(t_1, t_2)$ gives a bound on the size difference between two terms $t_1$ and $t_2$ under a measure $m$:

$$diff_m(t_1, t_2) = \begin{cases} 
  n & \text{if } |\theta(t_2)|_m - |\theta(t_1)|_m \leq n \text{ for every substitution } \theta \text{ such that } \\
  & \theta(t_1) \text{ and } \theta(t_2) \text{ are ground} \\
  \bot & \text{otherwise.}
\end{cases} \quad (5.2)$$

Thus, $diff_{\text{length}}([a|L], L) = -1$. But $diff_{\text{length}}([L, a], L) = \bot$ because $|\theta([L, a])|_{\text{length}} = 2$ for every substitution $\theta$ such that $\theta([L, a])$ is ground, while depending on $\theta$, $|\theta(L)|_{\text{length}}$ may be any natural number. A detailed realization of the diff functions for the measures in $\Gamma$ is given in Figure 5.2.

**Proposition 5.1.2** For every function defined in Figure 5.2, if $diff_m(t_1, t_2)$ is defined for two terms $t_1$ and $t_2$, then $diff_m(t_1, t_2) \geq |\theta(t_2)|_m - |\theta(t_1)|_m$ for every substitution $\theta$ such that $\theta(t_1)$ and $\theta(t_2)$ are ground. $\square$

## 5.2 Clause Analysis

This section describes a method for the inference of an upper bound on the size of each argument in a clause based on data dependency information. The size of each argument position is represented by an expression in terms of the size of the input positions in the head.
If $m$ is nat, then for any given term $t$

$$size_m(t) = \begin{cases} 
    t & \text{if } t \text{ is a natural number} \\
    \odot(size_m(t_1), \ldots, size_m(t_n)) & \text{if } t = \odot(t_1, \ldots, t_n) \text{ for some arithmetic operator } \odot \\
    \bot & \text{otherwise}.
\end{cases}$$

If $m$ is length, then for any given term $t$

$$size_m(t) = \begin{cases} 
    0 & \text{if } t \text{ is the empty list} \\
    1 + size_m(t_2) & \text{if } t = [t_1 | t_2] \\
    \bot & \text{otherwise}.
\end{cases}$$

If $m$ is depth$(f/n, r)$, where $f/n$ is a function symbol and $r$ is a list of positive integers, then for any given term $t$

$$size_m(t) = \begin{cases} 
    0 & \text{if } t \text{ is a constant} \\
    1 + \max\{size_m(t_i) \mid i \text{ in } r\} & \text{if } t = f(t_1, \ldots, t_n) \text{ and } 1 \leq i \leq n, \text{ for all } i \text{ in } r \\
    \bot & \text{otherwise}.
\end{cases}$$

If $m$ is size, then for any given term $t$

$$size_m(t) = \begin{cases} 
    1 & \text{if } t \text{ is a constant} \\
    1 + \sum_{i=1}^{n} \{size_m(t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \\
    \bot & \text{otherwise}.
\end{cases}$$

Figure 5.1: The definition of size for some common size measures
If $m$ is \texttt{nat}, then for any two terms $t_1$ and $t_2$
\[
diff_m(t_1, t_2) = \begin{cases} 
0 & \text{if } t_1 \equiv t_2 \\
\bot & \text{otherwise.}
\end{cases}
\]

If $m$ is \texttt{length}, then for any two terms $t_1$ and $t_2$
\[
diff_m(t_1, t_2) = \begin{cases} 
0 & \text{if } t_1 \equiv t_2 \\
\diff_m(t_2, t_2) - 1 & \text{if } t_1 = [t_1|t_2] \\
\bot & \text{otherwise.}
\end{cases}
\]

If $m$ is \texttt{depth}(f/n, r), where $f/n$ is a function symbol and $r$ is a list of positive integers, then for any two terms $t_1$ and $t_2$
\[
diff_m(t_1, t_2) = \begin{cases} 
0 & \text{if } t_1 \equiv t_2 \\
\min\{\diff_m(t_i, t_2) \mid i \in r\} - 1 & \text{if } t_1 = f(t_1, \ldots, t_n) \text{ and } 1 \leq i \leq n, \text{ for all } i \in r \\
\bot & \text{otherwise.}
\end{cases}
\]

If $m$ is \texttt{size}, then for any two terms $t_1$ and $t_2$
\[
diff_m(t_1, t_2) = \begin{cases} 
0 & \text{if } t_1 \equiv t_2 \\
\arg(sz(t_1), i) - sz(t_1) & \text{if } t_1 = f(t_1, \ldots, t_n) \text{ and } t_i \equiv t_2 \text{ for some } i, 1 \leq i \leq n \\
\bot & \text{otherwise.}
\end{cases}
\]

where $\arg(sz(t_1), i)$ is a symbolic variable denoting $|\theta(t_2)|_m$ and $sz(t_1)$ is a symbolic variable denoting $|\theta(t_1)|_m$ for substitutions $\theta$ such that $\theta(t_1)$ and $\theta(t_2)$ are ground.

Figure 5.2: The definition of \texttt{diff} for some common size measures
Basically, if possible, we will use the size functions to infer the size of an argument position and use the diff functions to infer the size relationship between dependent argument positions. For example, consider the following clause:

\[
\begin{align*}
&\text{:- mode}(\text{nrev}/2, [+,*]). \\
&\text{:- measure}(\text{nrev}/2, [\text{length},\text{length}]). \\
&\text{:- mode}(\text{append}/3, [+,*,-]). \\
&\text{:- measure}(\text{append}/3, [\text{length},\text{length},\text{length}]).
\end{align*}
\]

\[\text{nrev}([H|L], R) :- \text{nrev}(L, R1), \text{append}(R1, [H], R).\]

Using the \text{diff}_{\text{length}} function, we can infer that at run time, the list length of the argument \(L\) at the first argument position of the first body literal \text{nrev}/2 is at most 1 less than the list length of the argument \([H|L]\) at the first argument position of the clause head because \(\text{diff}_{\text{length}}([H|L], L) = -1\) and from Proposition 5.1.2, this is independent of substitutions. On the other hand, using the \text{size}_{\text{length}} function, we can infer that at run time, the list length of the argument at the second argument position of the second body literal \text{append}/3 is always 1 since \(\text{size}_{\text{length}}([H]) = 1\) and from Proposition 5.1.1, this is independent of substitutions.

In brief, we will apply the size and diff functions in the following way. If the size of the term occurring at an argument position \(a\) can be determined using a size function, then the result of the size function is used as the size of \(a\) because it is independent of substitutions from Proposition 5.1.1. Otherwise, the size of \(a\) is determined using a diff function. More specifically, the size of \(a\) is determined based on the size differences between \(a\) and its predecessors in the argument dependency graph. If the size of \(a\) can be determined using a diff function, then the result of the diff function is used to compute the size of \(a\) because it ensures an upper bound estimate independent of substitutions from Proposition 5.1.2. Otherwise, the size of \(a\) is given as \(\perp\).

Following data dependency relationships between argument positions, the analysis proceeds as follows. We first consider the input positions in the head. We then consider the body literals from left to right. For each literal, we consider its input positions ahead of its output positions. Finally, we consider the output positions in the head.

We will use \$\alpha a\$ to denote the term occurring at an argument position \(a\) in a literal
of a clause $C$, $m_a$ to denote the measure associated with $a$, and $sz(\theta a)$ to denote an expression representing the size of $\theta a$ under the measure $m_a$ in terms of the size of the input positions in the head of $C$. For example, consider the nrev/2 clause above. If $a$ is the first argument position of the clause head, then $\theta a$ is $[H|L]$, $m_a$ is length, and $sz(\theta a)$ is a symbolic variable representing the length of $[H|L]$, which depends on substitutions computed at run time.

Let $C$ be a clause $L_0 : = L_1, \ldots, L_m$ of a program $P$. Then, we will also use $\Theta_C$ to denote the set of substitutions $\theta$ such that the input arguments of $\theta(L_0)$ are ground, and $\Theta_C(\theta, i)$ to denote the bag of answer substitutions of $\theta(L_1, \ldots, L_i)$ with respect to $P$. In some cases, we also use $\Theta_C(\theta, x)$ to denote $\Theta_C(\theta, m)$, that is, the bag of answer substitutions for the clause. Since the expression $sz(\theta b)$ representing the size of an input position $b$ in the head of $C$ is indeed a function $\lambda \theta. |\theta(\theta b)|_{m_b}$, for $\theta \in \Theta_C$, the size expression $sz(\theta a)$ for argument positions $a$ that are not an input position in the head of $C$ will also be considered to be a function with domain $\Theta_C$.

5.2.1 Input Positions in the Head

The size of each input position in the head is intended to serve as an independent variable so that the size of an argument position that is not an input position in the head can be represented in terms of this variable. Sometimes, however, the size of an input position is actually a constant independent of substitutions. In this case, we will use the size functions to compute the constant size. Also, in some situations, we may want to ignore the size of an input position, as specified by the measure '?', denoting irrelevant. In this case, the size of the input position is given as $\bot$.

An algorithm for the inference of the size of an input position in the head is described as follows. Let $a$ be an input position in the head of a clause $C$. The size expression $sz(\theta a)$ can be determined as follows.

1. If $m_a$ is '?', then $sz(\theta a) = \bot$.
2. Otherwise, if $size_{m_a}(\theta a) \neq \bot$, then $sz(\theta a) = size_{m_a}(\theta a)$.
3. Otherwise, use $sz(\theta a)$ as a symbolic variable, denoting the function $\lambda \theta. |\theta(\theta a)|_{m_a}$, for $\theta \in \Theta_C$. (Note that size expressions for argument positions that are not input position in the head my be represented in terms of this symbolic variable.)
As an example, consider the following predicate:

```prolog
:- mode(nrev/2, [+,-]).
:- measure(nrev/2, [length,length]).
nrev([], []).
nrev([H|L], R) :- nrev(L, R1), append(R1, [H], R).
```

Since the input term in the first clause is the empty list, whose length is 0 regardless of substitutions, the size of the input position in the first clause is given as 0. On the other hand, the size of the input term [H|L] in the second clause cannot be determined by the function \(size_{\text{length}}\), the size of the input position in the second clause is given as a symbolic variable.

**Proposition 5.2.1** Let \( a \) be an input position in the head of a clause \( C \). If \( sz(\@a) \) computed by the algorithm above is defined, then \( sz(\@a)(\theta) = |\theta(\@a)|_{m_\theta} \), for every \( \theta \in \Theta_C \).

**Proof** There are two cases in which \( sz(\@a) \) is defined. First, \( sz(\@a) = size_{m_\theta}(\@a) \). From Proposition 5.1.1, \( sz(\@a)(\theta) = |\theta(\@a)|_{m_\theta} \), for every \( \theta \in \Theta_C \).

Second, \( sz(\@a) \) is used as a symbolic variable. Since \( sz(\@a) \) denotes \( \lambda \theta. |\theta(\@a)|_{m_\theta} \), for \( \theta \in \Theta_C \), we have \( sz(\@a)(\theta) = |\theta(\@a)|_{m_\theta} \), for every \( \theta \in \Theta_C \). \( \square \)

### 5.2.2 Input Positions in the Body Literals

The body literals in a clause are considered from left to right. Since every clause is acyclically well-moded under the left-to-right order, the size of an input position in a body literal can be determined by examining either the term occurring at that position (using the \( size \) functions) or the terms occurring at its predecessors (using the \( diff \) functions).

Consider the following clause:

```prolog
:- mode(nrev/2, [+,-]).
:- measure(nrev/2, [length,length]).
:- mode(append/3, [+,-]).
:- measure(append/3, [length,length,length]).
nrev([H|L], R) :- nrev(L, R1), append(R1, [H], R).
```
Using the \texttt{diff\_length} function, we can infer that at run time, the list length of the argument \texttt{L} at the first argument position of the body literal \texttt{nrev/2} is at most 1 less than the list length of the argument \texttt{[H|L]} at the first argument position of the clause head because \( \text{diff\_length}([H|L], L) = -1 \) and it is independent of substitutions from Proposition 5.1.2. Similarly, using the \texttt{diff\_length} function, we can infer that at run time, the list length of the argument \texttt{R1} at the first argument position of the literal \texttt{append/3} is the same as the list length of the argument \texttt{R1} at the second argument position of the literal \texttt{nrev/2}. On the other hand, using the \texttt{size\_length} function, we can infer that at run time, the list length of the argument at the second argument position of the literal \texttt{append/3} is always 1 since \( \text{size\_length}([H]) = 1 \) and it is independent of substitutions from Proposition 5.1.1.

An algorithm for the inference of an upper bound on the size of an input position in a body literal is described as follows. Let \( b \) be an input position in a body literal, and let \texttt{preds(b)} denote the set of predecessors of \( b \) in the argument dependency graph. The size expression \( \text{sz}(\mathbb{Q}b) \) can be determined as follows.

1. If \( m_b \) is `?', then \( \text{sz}(\mathbb{Q}b) = \bot. \)
2. Otherwise, if \( \text{size}_{m_b}(\mathbb{Q}b) \neq \bot \), then \( \text{sz}(\mathbb{Q}b) = \text{size}_{m_b}(\mathbb{Q}b). \)
3. Otherwise, if there is an argument position \( a \in \text{preds}(b) \) such that \( m_a = m_b \) and \( \text{diff}_{m_b}(\mathbb{Q}a, \mathbb{Q}b) \neq \bot \), then \( \text{sz}(\mathbb{Q}b) = \text{sz}(\mathbb{Q}a) + \text{diff}_{m_b}(\mathbb{Q}a, \mathbb{Q}b). \) (Note that \( \text{sz}(\mathbb{Q}a) \) has already been computed.)
4. Otherwise, if \( \text{size}_{m_b}(\mathbb{Q}b) \) can be expanded using the definition in Figure 5.1, then:
   (a) expand \( \text{size}_{m_b}(\mathbb{Q}b) \) one step accordingly;
   (b) recursively compute \( \text{sz}(t_i) \) for the appropriate subterms \( t_i \) (depending on the measure involved) of \( \mathbb{Q}b \) with respect to the same set of predecessors \( \text{preds}(b) \);
   (c) if each of these recursive size computations has a defined result, then use them to compute \( \text{sz}(\mathbb{Q}b) \) as appropriate (depending on the particular size measure under consideration); if the result of any of the recursive size computations is undefined, \( \text{sz}(\mathbb{Q}b) = \bot. \)
5. Otherwise, \( \text{sz}(\mathbb{Q}b) = \bot. \)
Note that, in step 3, there may be more than one argument position \( a \in \text{preds}(b) \) such that \( m_a = m_b \) and \( \text{diff}_{m_a}(@a, @b) \neq \perp \). A more precise estimation should take the minimum \( \text{diff}_{m_a}(@a, @b) \) as its result.

Note also that for each size expression \( sz(@b) \) obtained using the \text{diff} functions, because the size obtained for each of the predecessors of \( b \) is an upper bound estimate, \( sz(@b) \) has to be a nondecreasing expression in terms of these sizes in order to guarantee an upper bound estimate for the size of \( b \).

**Proposition 5.2.2** Let \( C \) be a clause \( L_0 : - L_1, \ldots, L_m \), the input positions in the head of \( C \) be \( c_1, \ldots, c_n \), and \( b \) be an input position in \( L_i \). Suppose for each \( a \in \text{preds}(b) \), \( sz(@a) \) is nondecreasing with respect to \( sz(@c_1), \ldots, sz(@c_n) \), and \( sz(@a)(\theta) \geq |\sigma(@a)|_{m_a} \), for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i-1) \). If every natural number expression in \( C \) is nondecreasing, and \( sz(@b) \) computed by the algorithm above is defined, then \( sz(@b) \) is nondecreasing with respect to \( sz(@c_1), \ldots, sz(@c_n) \), and \( sz(@b)(\theta) \geq |\sigma(@b)|_{m_b} \), for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i-1) \).

**Proof** There are two cases in which \( sz(@b) \) is defined. First, \( sz(@b) = size_{m_a}(@b) \). From Proposition 5.1.1, \( sz(@a)(\theta) = |\sigma(@a)|_{m_a} \), for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i-1) \).

Second, \( sz(@b) \) is determined using a \text{diff} function and possibly an expansion of a \text{size} function. We consider measure \text{nat} first. For measure \text{nat}, the expansion of its \text{size} function depends on the operators in the natural number expression under consideration. If every natural number expression in \( C \) is nondecreasing, then an upper bound on the value of a natural number expression can be obtained from upper bounds on the value of its subexpressions. Consider each subterm \( t \) of \( @b \) such that \( sz(t) \) is determined using \( sz(t) = sz(@a) + \text{diff}_{m_a}(@a, t) \), for some \( a \in \text{preds}(b) \). By Proposition 5.1.2, \( \text{diff}_{m_a}(@a, t) \geq |\sigma(t)|_{m_a} - |\sigma(@a)|_{m_a} \), for every \( \sigma \in \Theta_C(\theta, i-1) \). If for each \( a \in \text{preds}(b) \), \( sz(@a)(\theta) \geq |\sigma(@a)|_{m_a} \), for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i-1) \), then \( sz(t)(\theta) \geq |\sigma(t)|_{m_a} \), for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i-1) \). If for each \( a \in \text{preds}(b) \), \( sz(@a) \) is nondecreasing with respect to \( sz(@c_1), \ldots, sz(@c_n) \), then \( sz(t) \) is also nondecreasing with respect to \( sz(@c_1), \ldots, sz(@c_n) \). Since every natural number expression in \( C \) is nondecreasing, we have \( sz(@b) \) is nondecreasing with respect to \( sz(@c_1), \ldots, sz(@c_n) \), and \( sz(@b)(\theta) \geq |\sigma(@b)|_{m_b} \), for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i-1) \).
For each measure other than nat, its size function involves only summation or the maximum function, thereby being nondecreasing. As a result, the rest of the proof follows in the same way as for measure nat. □

5.2.3 Output Positions in the Body Literals

The size of an output position in a body literal usually depends on the size of the input positions in the literal. Consider the following clause:

\[- \text{mode(nrev/2, [+],+}).\]
\[- \text{measure(nrev/2, [length],length])}.\]
\[- \text{mode(append/3, [+],+,-]).\]
\[- \text{measure(append/3, [length],length,length])}.\]

\[\text{nrev}([H|L], R) :- \text{nrev}([H], R1), append(R1, [H], R).\]

From the previous section, we can infer that at run time, the list length of the argument R1 at the first argument position of the literal append/3 is the same as the list length of the argument R1 at the second argument position of the literal nrev/2. However, to know the list length of the argument at the second argument position of the literal nrev/2, we need to infer the size relationship between the input and output arguments of nrev/2. Since the literal nrev/2 is recursive, its argument-size function is exactly the one we are trying to infer. Hence, we can only represent its argument-size function in symbolic form initially (this will lead to a difference equation) and will try to solve it later (by solving the difference equation). If the literal under consideration is nonrecursive, then since the predicates are processed in a topological order of the call graph, its argument-size functions have already been computed. In this case, we can immediately use these argument-size functions to infer the size of its output positions.

An algorithm for the inference of an upper bound on the size of an output position in a body literal is described as follows. Let \( a \) be the \( i \)th (output) argument position in a body literal \( l \) with input positions \( b_1, \ldots, b_n \). The size expression \( sz(\!@a) \) can be determined as follows.
1. If \( m_a \) is '?', then \( \text{sz}(a) = \bot \).

2. Otherwise, if \( l \) is recursive, then \( \text{sz}(a) \) is symbolically expressed as

\[
\text{sz}(a) = \text{Sz}_{l}(\text{sz}(b_1), \ldots, \text{sz}(b_n)).
\]

3. Otherwise, if \( l \) is nonrecursive, and assume that the argument-size complexity function for \( a \) has been computed as \( \text{Sz}_{l}(x_1, \ldots, x_n) \), then \( \text{sz}(a) \) can be expressed as

\[
\text{sz}(a) = \text{Sz}_{l}(\text{sz}(b_1), \ldots, \text{sz}(b_n)).
\]

Note that, in step 3, since the predicates are processed in a topological order of the call graph, the argument-size complexity functions giving the size relationship between the input and output positions of \( l \) have already been computed; we are able, in this case, to express the relationship between \( \text{sz}(a) \) and \( \{\text{sz}(b_1), \ldots, \text{sz}(b_n)\} \) explicitly in terms of this function.

Note also that for each size expression \( \text{sz}(a) \) obtained using the argument-size function \( \text{Sz}_l \), because the size obtained for each of the input positions of \( l \) is an upper bound estimate, \( \text{Sz}_l \) must be a nondecreasing function in terms of these sizes in order to guarantee an upper bound estimate for the size of \( a \).

**Proposition 5.2.3** Let \( C \) be a clause \( L_0 : - L_1, \ldots, L_m \), the input positions in the head of \( C \) be \( c_1, \ldots, c_n \), and \( a \) be the \( j \)th (output) position in \( L_i \). Suppose for each of the input positions \( b \) of \( L_i \), \( \text{sz}(b) \) is nondecreasing with respect to \( \text{sz}(c_1), \ldots, \text{sz}(c_n) \), and \( \text{sz}(b)(\theta) \geq |\sigma(b)|_{m_b} \) for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i - 1) \). If \( \text{Sz}_{L_i}(j) \) is nondecreasing, and \( \text{sz}(a) \) computed by the algorithm above is defined, then \( \text{sz}(a) \) is nondecreasing with respect to \( \text{sz}(c_1), \ldots, \text{sz}(c_n) \), and \( \text{sz}(a)(\theta) \geq |\sigma(a)|_{m_a} \) for every \( \theta \in \Theta_C \) and every \( \sigma \in \Theta_C(\theta, i) \).

**Proof** Let \( L_i \) have input positions \( b_1, \ldots, b_k \). Then \( \text{sz}(a) \) is symbolically or explicitly expressed as

\[
\text{sz}(a) = \text{Sz}_{L_i}(\text{sz}(b_1), \ldots, \text{sz}(b_k)).
\]

If \( \text{Sz}_{L_i}(j) \) is nondecreasing, and for each of the input positions \( b \) of \( L_i \), \( \text{sz}(b) \) is nondecreasing with respect to \( \text{sz}(c_1), \ldots, \text{sz}(c_n) \), then \( \text{sz}(a) \) is also nondecreasing with respect to
If $sz_b^j(i)$ is nondecreasing, and for each of the input positions $b$ of $L_i$, $sz_b^j(i)(\theta) \geq |\sigma(\@b)|_{m_i}$, for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, i - 1)$, then

$$sz(\@a)(\theta) = sz^j_L(i)(sz_b^j(\@b_1)(\theta), \ldots, sz_b^j(\@b_k)(\theta))$$
$$\geq sz^j_L(i)(|\beta(\@b_1)|_{m_i}, \ldots, |\beta(\@b_k)|_{m_i})$$
$$\geq |\beta(\@a)|_{m_a}$$

for every $\theta \in \Theta_C$ and every $\beta \in \Theta_C(\theta, i). \ \Box$

5.2.4 Output Positions in the Head

The size of an output position in the head can be determined in the same way as the input positions in the body literals, by examining either the term occurring at that position or the terms occurring at its predecessors. Consider the following clause:

```
:- mode(nrev/2, [+]+).
:- measure(nrev/2, [length,length]).
:- mode(append/3, [+,+,-]).
:- measure(append/3, [length,length,length]).

nrev([H|L], R) :- nrev(L, R1), append(R1, [H], R).
```

Using the $\text{diff}$ functions, we can infer that at run time, the list length of the argument $R$ at the second argument position of the clause head is the same as the list length of the argument $R$ at the third argument position of the literal $\text{append}/3$. This is because $\text{diff}_{\text{length}}(R, R) = 0$ and it is independent of substitutions from Proposition 5.1.2. Thus, we can use the algorithm for the inference of an upper bound on the size for an input position in a body literal to infer an upper bound on the size for an output position in the head.

**Proposition 5.2.4** Let $C$ be a clause with the input positions $c_1, \ldots, c_n$ in the head, and $b$ be an output position in the head of $C$. Suppose for each $a \in \text{preds}(b)$, $sz(\@a)$ is nondecreasing with respect to $sz(\@c_1), \ldots, sz(\@c_n)$, and $sz(\@a)(\theta) \geq |\sigma(\@a)|_{m_a}$, for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, \ast)$. for every feasible substitution $\theta$ of $C$. If $sz(\@b)$ computed by the algorithm above is defined, then $sz(\@b)$ is nondecreasing with respect to $sz(\@c_1), \ldots, sz(\@c_n)$, and $sz(\@b)(\theta) \geq |\sigma(\@b)|_{m_b}$, for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, \ast)$. 
Proof It is the same as the proof of Proposition 5.2.2. □

The algorithms described in this section are summarized into the algorithm in Figure 5.3.

**Proposition 5.2.5** Let $C$ be a clause $L_0 : - L_1, \ldots, L_m$, the input positions in the head of $C$ be $c_1, \ldots, c_n$, and $b$ be an output position in the head of $C$. If every natural number expression in $C$ is nondecreasing, the function $Sz_{L_i}$ for every output position of $L_i$ is nondecreasing, and $sz(\sigma b)$ computed by Algorithm 5.1 is defined, then $sz(\sigma b)$ is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$, and $sz(\sigma b)(\theta) \geq |\sigma(\sigma b)|_{m_0}$ for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, \ast)$.

**Proof** From Propositions 5.2.1, 5.2.2, 5.2.3 and 5.2.4. □

### 5.2.5 Size Information from Built-in Literals

In addition to size and diff functions, we can often infer useful size information from built-in literals in the clause body. We consider two classes of built-in predicates.

First, we consider built-in predicates that are type tests, for example, atom/1, atomic/1, float/1, integer/1, and number/1. Each literal of this class tests, at run time, the membership of its argument in a type, and thus implicitly constrains the size of its argument. Size information from this class of built-in predicates will only be used when the size measure under consideration is either $\text{depth}(f, r)$ or size. Since only constants can satisfy these predicates, the size inferred is either 0 or 1 under these measures.

As an example, consider a clause

$$p(X) : - \text{atomic}(X), \ldots$$

Suppose the argument position of $p/1$ is an input position and has a measure size. Before the call to atomic/1 succeeds, we can only infer that the size of $X$ is a variable that depends on the binding of $X$. However, after the call to atomic/1 succeeds, we can safely say that the size of $X$ is 1. This is because the call `atomic(X)` can succeed only if $X$ is bound to a constant, and in this case, the binding of $X$ has size 1 under the measure size.

Second, we consider the built-in predicate that are "equality" tests, namely, =:=/2 and =/=/2. It is often that these tests are in the form of
Algorithm 5.1

sz_clause(C): Argument-size complexity analysis of a clause.

Input: A clause C: L₀ := L₁, . . . , Lₘ.

Output: The size expressions sz(}@a) for all the output positions @a in L₀.

Method:

begin

for each input position a in L₀ do

    if sizeₘₙ(}@a) ̸= ⊥ then sz(}@a) := sizeₘₙ(}@a) else sz(}@a) is a variable od

for i := 1 to m do

    for each input position a in Lᵢ do

        if sizeₘₙ(}@a) ̸= ⊥ then sz(}@a) := sizeₘₙ(}@a)
        else if diffₘₙ(}@b, @a) ̸= ⊥ for an argument position b ∈ preds(@a) then
            sz(}@a) := sz(}@b) + diffₘₙ(}@b, @a)
        else recursively compute sz(}@i) for subterms @i of @a and
            compute sz(}@a) using sz(}@i) od

    for each output position a in Lᵢ do

        let a be the jth position and b₁, . . . , bₙ be the input positions of Lᵢ;

        if Lᵢ is recursive then
            sz(}@a) is symbolically expressed as Szₘₙ(}@a) := Szₘₙ(}@b₁), . . . , Szₘₙ(}@bₙ))
        else sz(}@a) := Szₘₙ(}@b₁), . . . , sz(}@bₙ)) od od

for each output position a in L₀ do

    if sizeₘₙ(}@a) ̸= ⊥ then sz(}@a) := sizeₘₙ(}@a)
    else if diffₘₙ(}@b, @a) ̸= ⊥ for an argument position b ∈ preds(}@a) then
        sz(}@a) := sz(}@b) + diffₘₙ(}@b, @a)
    else recursively compute sz(}@i) for subterms @i of @a and
        compute sz(}@a) using sz(}@i) od

return sz(}@a) for all the output positions @a in L₀

end

Figure 5.3: An algorithm for argument-size complexity analysis of a clause
Measures | Built-in Predicates
--- | ---
**nat** | `=:=/2, ==/2`.
**length** | `==/2`.
**depth(r)** | `atom/1, atomic/1, float/1, integer/1, number/1, =:=/2, ==/2`.
**size** | `atom/1, atomic/1, float/1, integer/1, number/1, =:=/2, ==/2`.

Table 5.1: Built-in predicates used for the inference of the size of a variable

'X =:= t' or 'X == t',

where X is a variable and t is a term such that $size_m(t)$ is defined for the measure m under consideration. In this case, $size_m(t)$ is given as the size of X.

As an example, consider a clause

```
p(X) :- X =:= 1, ...
```

Suppose the argument position of p/1 is an input position and has a measure nat. Before the call to `=:=/2` succeeds, we can only infer that the size of X is a variable that depends on the binding of X. However, after the call to `=:=/2` succeeds, we can safely say that the size of X is 1. This is because the call 'X =:= 1' can succeed only if X is bound to the natural number 1, and in this case, the binding of X has size 1 under the measure nat.

Built-in predicates that are used for the inference of the size of a variable under various measures are given in Table 5.1. By using built-in literals, we can infer adequate size information for both of the following predicates fact/2 and fact1/2.

```
fact(0, 1).
fact(N, M) :- N > 0, N1 is N-1, fact(N1, M1), M is N*M1.
```

```
fact1(0, 1) :- N =:= 0.
fact1(N, M) :- N > 0, N1 is N-1, fact1(N1, M1), M is N*M1.
```

5.3 Predicate Analysis

This section describes how the argument-size complexity function for an output position of a predicate can be derived from the size expressions for the same output position of
its clauses. Basically, since we are considering the worst-case analysis, the argument-size complexity function for an output position of a predicate can be obtained by taking the maximum among the size expressions for the same output position of its clauses.

Let $b$ be the $i^{th}$ (output) argument position of a predicate $p$ with input positions $a_1, \ldots, a_n$. Let $sz(\@b)$ be the size expression computed for $b$ in a clause of $p$. Since the size of an output position usually depends on the size of the input positions, we can also symbolically represent the size of $b$ as $Sz^{(i)}(sz(\@a_1), \ldots, sz(\@a_n))$, namely,

$$Sz^{(i)}_p(sz(\@a_1), \ldots, sz(\@a_n)) = sz(\@b).$$

This equation explicitly represents the size of an output position in the head in terms of the size of the input positions in the head. If $sz(\@b)$ does not contain any symbolic expression except for the size of the input positions in the head, then this equation is a closed form function. On the other hand, if, in addition to the size of the input positions in the head, $sz(\@b)$ also contains symbolic expressions representing the size of the output positions in the recursive literals, then this equation is in the form of difference equation.

An algorithm for deriving an argument-size complexity function of a predicate is given in Figure 5.4. The analysis for nonrecursive predicates is straightforward. Since the functions obtained from the clauses of a nonrecursive predicate are already in closed form, we can simply take the maximum among them as an argument-size complexity function for the predicate. Let $b$ be the $i^{th}$ (output) argument position of a nonrecursive predicate $p$ with input positions $a_1, \ldots, a_n$. Also, let $e_1, \ldots, e_m$ be the size expressions for $b$ obtained from the clauses of $p$. Then, the argument-size complexity function for $b$ can be represented as

$$Sz^{(i)}_p(sz(\@a_1), \ldots, sz(\@a_n)) = \max\{e_1, \ldots, e_m\}. \quad (5.3)$$

The analysis for recursive predicates is more complicated. We consider the clauses in two cases. First, consider a nonrecursive clause. The clause usually serves as a base case of a recursion and produces a solution when the recursion succeeds through this clause. Therefore, its size equation provides a boundary condition for the difference equations obtained from recursive clauses.

Second, consider a recursive clause. While a computation may go through recursive clauses, it only produces solutions through nonrecursive clauses. As a result, the size
Algorithm 5.2
sz_predicate(p): Argument-size complexity analysis of a predicate.

Input: A predicate p.
Output: The argument-size complexity functions Sz_p for all the output positions of p.

Method:

begin
for each clause C_i in p do
    e_i := sz_clause(C_i)
oď
for each output position a of p do
    let a be the jth argument position of p;
    if p is nonrecursive then
        Sz_p(j) := max_i\{e_i\}
    else
        d := max_i\{e_i \mid e_i is a difference equation\};
        D := d \cup \{e_i \mid e_i is not a difference equation\};
        Sz_p(j) := solve_difference_equation(D)
    fi
od
return Sz_p for all the output positions of p
end

Figure 5.4: An algorithm for argument-size complexity analysis of a predicate
equation of a recursive clause is usually in the form of difference equation and needs to be solved together with the boundary conditions provided by nonrecursive clauses.

Since a recursion may go through several distinct recursive clauses, to achieve the worst-case analysis, we need to consider the maximum of the size expressions obtained from distinct recursive clauses. Let \( b \) be the \( i \text{th} \) (output) argument position of a recursive predicate \( p \) with input positions \( a_1, \ldots, a_n \). Also, let \( e_1, \ldots, e_m \) be the size expressions for \( b \) obtained from the recursive clauses of \( p \). Then, the argument-size complexity function for \( b \) can be obtained by solving the following difference equation

\[
S_{p}^{(i)}(sz(a_1), \ldots, sz(a_n)) = \max\{e_1, \ldots, e_m\}
\]  

(5.4)

together with the boundary conditions provided by the nonrecursive clauses of \( p \). Recall that each of the expressions \( e_i \) contains some complexity function \( S_{q} \) represented in symbolic form, where \( q \) can be \( p \) or a predicate indirectly recursive with \( p \). Automatic solution of difference equations is discussed in Chapter 11. For now let us assume that \( solve\_difference\_equation \) is a function that returns a nondecreasing upper bound solution for any given set of difference equations.

As an example, consider the following predicate \( \text{part/4} \), which partitions a list of numbers into two sublists based on the pivot number given at the first argument position.

\[
\text{:- mode(part/4, [+,-,-]).}
\]
\[
\text{:- measure(part/4, [?,length,length,length]).}
\]
\[
\text{part(F, [], [], []).}
\]
\[
\text{part(F, [XY], [XY1], Y2) :- X =< F, part(F, Y, Y1, Y2).}
\]
\[
\text{part(F, [XY], Y1, [XY2]) :- X > F, part(F, Y, Y1, Y2).}
\]

The size expressions for the third (output) argument position of \( \text{part/4} \) obtained from the three clauses can be represented as functions in terms of the size of the second argument position of \( \text{part/4} \) as follows:

\[
f(0) = 0,
\]
\[
f(n) = f(n - 1) + 1,
\]
\[
f(n) = f(n - 1).
\]

Since a computation may recurse through both the second and the third clauses, to achieve the worst-case analysis, we need to solve the following set of difference equations:
\( f(0) = 0, \)
\( f(n) = \max(f(n - 1) + 1, f(n - 1)) = f(n - 1) + 1, \)

which yields the expected solution \( \lambda x. x \), namely, the maximum length of the output list at the third argument position is the length of the input list at the second argument position.

**Theorem 5.3.1** Let \( p \) be a predicate, \( C_i : L_{i,0} : - L_{i,1}, \ldots, L_{i,m} \) be the \( i \)th clause of \( p \), and \( c_{i,1}, \ldots, c_{i,n} \) be the input positions in \( L_{i,0} \). Let \( b \) be the \( j \)th (output) argument position of \( p \). If the function \( \text{solve_difference_equation} \) returns a nondecreasing upper bound solution for any given set of difference equations, every natural number expression in \( p \) is nondecreasing, and \( \text{Sz}_{p}^{(j)} \) computed by Algorithm 5.2 is defined, then \( \text{Sz}_{p}^{(j)} \) is a nondecreasing argument-size complexity function for \( b \).

**Proof** Let \( p \) be a predicate in a program \( P \), and \( a_1, \ldots, a_n \) be the input positions of \( p \). We will prove that by processing the predicates in \( P \) in a topological order of the call graph, every defined complexity function computed by Algorithm 5.2 is a nondecreasing argument-size complexity function. We show by induction on the topological order of the call graph.

We first consider three base cases in which there is no nonrecursive literals in the clauses of a predicate. First, consider a nonrecursive predicate \( p \) consisting of clauses with empty body. For each clause \( C_i \) of \( p \), let \( \text{sz}_i(\@b) \) denote the size expression for \( b \) computed for \( C_i \) by Algorithm 5.1. From Proposition 5.2.5, \( \text{sz}_i(\@b) \) is nondecreasing with respect to \( \text{sz}(\@c_{i,1}), \ldots, \text{sz}(\@c_{i,n}) \), or equivalently, with respect to \( \text{sz}(\@a_1), \ldots, \text{sz}(\@a_n) \). Since Equation (5.3) involves only the maximum function, and the maximum function is nondecreasing, \( \text{Sz}_{p}^{(j)} \) is also nondecreasing.

From Proposition 5.2.5, \( \text{sz}_i(\@b)(\theta) \geq |\sigma(\@b)|_{m_1} \) for every \( \theta \in \Theta_{C_i} \) and every \( \sigma \in \Theta_{C_i}(\theta, \star) \). For each goal \( g \) of \( p \), let \( t_{a_1}, \ldots, t_{a_n} \) be the input arguments of \( p \), \( \theta_i = \text{mgu}(g, L_{i,0}) \) for all \( i \), and \( t_b \) be any output argument bound at \( b \) when \( g \) succeeds. Then, we have
\[
\text{Sz}_{p}^{(j)}(\tau_{a_1}|_{m_{a_1}}, \ldots, \tau_{a_n}|_{m_{a_n}}) = \max_i \{ \text{sz}_i(\@b)(\theta_i) \}
\geq \max_i \{ |\sigma(\@b)|_{m_1} \mid \sigma \in \Theta_{C_i}(\theta_i, \star) \}
\geq |t_b|_{m_4}.
\]
Therefore, \( \text{Sz}_{p}^{(j)} \) is a nondecreasing argument-size complexity function for the argument position \( b \) of \( p \).
Second, consider a direct recursive predicate $p$ that involves only itself, namely, all the body literals in its clauses have the predicate symbol $p$. For each clause $C_i$ of $p$, let $sz_i(\theta_b)$ denote the size expression for $b$ computed for $C_i$ by Algorithm 5.1. From Proposition 5.2.5, if the argument-size complexity functions for $p$ are nondecreasing functions, then each $sz_i(\theta_b)$ is nondecreasing with respect to $sz(\theta_{a1}), \ldots, sz(\theta_{an})$. Since Equation (5.4) involves only the maximum function, the difference equation obtained from using Equation (5.4) is also a nondecreasing function with respect to $sz(\theta_{a1}), \ldots, sz(\theta_{an})$. Let $D$ be the set of difference equations derived for $p$. Then each of the equations in $D$ is also nondecreasing. Because by assumption, the function $solve\_difference\_equation$ returns a nondecreasing solution for any given set of difference equations, $SZ_p^{(i)}$ is also nondecreasing.

From Proposition 5.2.5, if the argument-size complexity functions for $p$ are nondecreasing upper bound functions, then we have $sz_i(\theta_b)(\theta) \geq |\sigma(\theta_b)|_{m_b}$, for every $\theta \in \Theta_{C_i}$ and every $\sigma \in \Theta_{C_i}(\theta, \ast)$. Since Equation (5.4) involves only the maximum function, the difference equation obtained from using Equation (5.4) is also an upper bound function. For each goal $g$ of $p$, let $t_{a1}, \ldots, t_{an}$ be the input arguments of $p$, $\theta_i = mgu(g, L_{i,\theta})$ for all $i$, and $t_b$ be any output argument bound at $b$ when $g$ succeeds. Then, if the function $solve\_difference\_equation$ returns an upper bound solution for any given set of difference equations, we have

$$SZ_p^{(i)}(t_{a1}|m_{a1}, \ldots, t_{an}|m_{an}) = solve\_difference\_equation(D)(t_{a1}|m_{a1}, \ldots, t_{an}|m_{an}) \geq \max_i\{sz_i(\theta_b)(\theta_i)\} \geq \max_i\{|\sigma(\theta_b)|_{m_b} \mid \sigma \in \Theta_{C_i}(\theta_i, \ast)\} \geq |t_b|_{m_b}.$$  

Thus, $SZ_p^{(i)}$ is a nondecreasing argument-size complexity function for the argument position $b$ of $p$. This also verifies the assumptions for applying Proposition 5.2.5 in the proof.

Third, consider a set of indirect recursive predicates that involve only themselves. Following the analogous argument as for the second case, and using the function $solve\_difference\_equation$ to solve a system of simultaneous difference equations, we can derive the same result as the second case.
Since these three cases are the only base cases, after proving these base cases, Proposition 5.2.5 can be applied for the inductive cases as for the base cases. Consequently, the inductive cases can be proved in the analogous way as for the base cases. □

If the set of difference equations of a recursive predicate can be solved to get a closed form expression, we can substitute this closed form expression for symbolic expressions in various size expressions for argument positions in the body literals of its recursive clauses, and transform them into closed form size expressions. In particular, for each clause, we want to have a closed form size expression for each input position in each body literal, whose input size is important for the inference of the number-of-solutions and time complexity for the clauses and for the predicate. However, for tail recursive clauses (namely, ones in which the recursive literal is the rightmost literal), there is no need for the substitution of symbolic expressions because the size expression for each input position of each body literal in such a clause is already in closed form.

**Theorem 5.3.2** Let $C$ be a clause $L_0 : - L_1, \ldots, L_m, c_1, \ldots, c_n$ be the input positions in the head of $C$, and $b$ be an input position in $L_i$, $1 \leq i \leq m$. If every natural number expression in $C$ is nondecreasing, the function solve_difference_equation returns a non-decreasing upper bound solution for any given set of difference equations, and $sz(\Theta b)$ computed by Algorithms 5.1 and 5.2 is defined, then $sz(\Theta b)$ is nondecreasing with respect to $sz(\Theta c_1), \ldots, sz(\Theta c_n)$, and $sz(\Theta b)(\theta) \geq |\sigma(\Theta b)|_{m_\theta}$ for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, i - 1)$.

**Proof** From Proposition 5.2.5 and Theorem 5.3.1. □

### 5.4 An Example

Consider the list permutation program:

```prolog
:- mode(perm/2, [+,-]).
:- measure(perm/2, [length,length]).
perm([], []).
perm([X|Y], [R|Rs]) :- select(R, [X|Y], Z), perm(Z, Rs).
```
Let \( head[i] \) and \( body_j[i] \) denote the sizes of the \( i \)th argument position in the head and in the \( j \)th body literal, respectively. We start with the analysis for the predicate \( \text{select/3} \). We first consider the nonrecursive clause. The size expression for the input position in the head is

\[
head[2] = size_{length}([X|Xs]) = head[2].
\]

The size expressions for the output positions in the head are

\[
head[1] = 1,
\]

\[
\]

Thus, the size functions for the output positions in the head can be represented as

\[
S_{select}^{(1)}(head[2]) = 1, \quad (5.5)
\]

\[
S_{select}^{(3)}(head[2]) = head[2] - 1. \quad (5.6)
\]

We now consider the recursive clause. The size expression for the input position in the head is

\[
head[2] = size_{length}([Y|Ys]) = head[2].
\]

The size expressions for argument positions in the body literal are

\[
body_1[2] = size_{length}(Ys) = head[2] + \text{diff}_{length}([Y|Ys], Ys) = head[2] - 1,
\]

\[
body_1[1] = 1,
\]

\[
\]

The size expressions for the output positions in the head are

\[
head[1] = 1,
\]

\[
head[3] = size_{length}([Y|Zs]) = size_{length}(Zs) + 1
\]

\[
= body_1[3] + \text{diff}_{length}(Zs, Zs) + 1
\]

\[
= body_1[3] + 1
\]

\[
= S_{select}^{(3)}(head[2] - 1) + 1.
\]
Thus, the size functions for the output positions in the head can be represented as

\[ S_{\text{select}}^{(1)}(\text{head}[2]) = 1, \]
\[ S_{\text{select}}^{(2)}(\text{head}[2]) = S_{\text{select}}^{(3)}(\text{head}[2] - 1) + 1. \]  

(5.7)

(5.8)

We now consider the size functions for the predicate. For the first argument position, from Equations (5.5) and (5.7), we have \( S_{\text{select}}^{(1)} \equiv \lambda x.\bot \). For the third argument position, Equations (5.6) and (5.8) can be solved to yield \( S_{\text{select}}^{(3)} \equiv \lambda x.x - 1 \), namely, the size of the third argument is bounded by the size of the second argument minus 1. As the recursive clause is tail recursive, the input size \( \text{head}[2] - 1 \) to the recursive literal is already in closed form, and thus there is no need for the substitution of symbolic expressions.

We now continue with the analysis for the predicate \text{perm}/2. We first consider the nonrecursive clause. The size expression for the input position in the head is

\[ \text{head}[1] = \text{size}_{\text{length}}([1]) = 0. \]

The size expression for the output position in the head is

\[ \text{head}[2] = \text{size}_{\text{length}}([1]) = 0. \]

Thus, the size function for the output position in the head can be represented as

\[ S_{\text{perm}}^{(2)}(0) = 0. \]  

(5.9)

We now consider the recursive clause. The size expression for the input position in the head is

\[ \text{head}[1] = \text{size}_{\text{length}}([X|Y]) = \text{head}[1]. \]

The size expressions for the argument positions in the body literals are

\[ \text{body}_1[2] = \text{size}_{\text{length}}([X|Y]) = \text{head}[1] + \text{diff}_{\text{length}}([X|Y],[X|Y]) = \text{head}[1], \]
\[ \text{body}_1[1] = \bot, \]
\[ \text{body}_1[3] = S_{\text{select}}^{(3)}(\text{body}_1[2]) = \text{body}_1[2] - 1 = \text{head}[1] - 1; \]
\[ \text{body}_2[1] = \text{size}_{\text{length}}(Y) = \text{body}_1[3] + \text{diff}_{\text{length}}(Z,Z) = \text{head}[1] - 1, \]
\[ \text{body}_2[2] = S_{\text{perm}}^{(2)}(\text{body}_2[1]) = S_{\text{perm}}^{(1)}(\text{head}[1] - 1). \]

The size expression for the output position in the head is
Thus, the size function for the output position in the head can be represented as

\[ S_{\text{perm}}^{(2)}(\text{head}[1]) = S_{\text{perm}}^{(2)}(\text{head}[1] - 1) + 1. \] (5.10)

Equations (5.9) and (5.10) can be solved to yield \( S_{\text{perm}}^{(2)} \equiv \lambda x . x \), namely, the size of the output of \text{perm}/2 is bounded by the size of its input. Since the recursive clause is again tail recursive, there is no need for the substitution of symbolic expressions.

### 5.5 Complexity

This section discusses the complexity of Algorithms 5.1 and 5.2. Let \( p, c, l, a, \) and \( t \) be defined as in Appendix B. Also, let \( E \) be the complexity of simplifying a symbolic expression, \( L \) the complexity of evaluating the maximum of two symbolic expressions, and \( D \) the complexity of solving a set of difference equations. These new parameters are also listed in Appendix B for later reference. All of the parameters \( E, L, \) and \( D \) are indeed functions of \( p, c, l, a, \) and \( t. \) However, for simplicity, we will only discuss the complexity of Algorithms 5.1 and 5.2 in terms of these parameters, and will not discuss how these parameters depend on \( p, c, l, a, \) and \( t. \)

We first discuss the complexity of evaluating \text{size} and \text{diff} functions. The value of these function can be determined by traversing the term given as the argument to a \text{size} function or the term given as the first argument to a \text{diff} function. Therefore, the complexity is linear on the term size of the argument, that is, the complexity of evaluating a \text{size} or a \text{diff} functions is \( O(t). \)

We now discuss the complexity of Algorithm 5.1. Consider the complexity of inferring the size of an input position in the head. Since we only use the \text{size} functions, the complexity of inferring the size of an input position in the head is \( O(t). \) It takes time \( O(a \cdot t) \) to consider all the input positions in the head.

Consider the complexity of inferring the size of an input position in a body literal. This complexity is dominated by the complexity of determining the size difference between a
position and its predecessors. Each term with term size $t$ has at most $t$ predecessors. Hence, each expansion of the size function applies the \textit{diff} function at most $t$ times and can be carried out in time $O(t^2)$. There are at most $t$ expansions for a term with term size $t$. Therefore, the complexity of inferring the size of an input position in a body literal is $O(t^3)$. It takes time $O(a \cdot t^3)$ to consider all the input positions in a literal.

Consider the complexity of inferring the size of an output position in a body literal. The size is given as a symbolic expression or is determined by simplifying a size function using given input size expressions. Thus this complexity is dominated by the complexity of simplifying the size function $a$ times, once for each input size expression, that is, by $O(a \cdot E)$. It takes time $O(a^2 \cdot E)$ to consider all the output positions in a literal.

The complexity of inferring the size of an output position in the head is the same as the complexity of inferring the size of an input position in a body literal. Thus, the complexity of analyzing a clause is $O(l \cdot a \cdot t^3 + l \cdot a^2 \cdot E)$.

Finally, we discuss the complexity of Algorithm 5.2. Analyzing all the clauses in the predicate requires time $O(c \cdot l \cdot a \cdot t^3 + c \cdot l \cdot a^2 \cdot E)$. The complexity of constructing Equation (5.3) or (5.4) from the size expressions of the clauses is $O(c \cdot L)$. Since a set of difference equations can be solved in time $D$, the complexity of deriving an argument-size complexity function of a predicate is $O(c \cdot l \cdot a \cdot t^3 + c \cdot l \cdot a^2 \cdot E + c \cdot L + D)$. It takes time $O(c \cdot l \cdot a^2 \cdot t^3 + c \cdot l \cdot a^3 \cdot E + a \cdot c \cdot L + a \cdot D)$ to consider all the output positions in the head.

In summary, the complexity of the argument-size complexity analyzer is $O(p \cdot c \cdot l \cdot a^2 \cdot t^3 + p \cdot c \cdot l \cdot a^3 \cdot E + p \cdot a \cdot c \cdot L + p \cdot a \cdot D)$.

### 5.6 Summary

The argument size complexity analyzer performs three functions. First, for each clause, it infers the size function for each output position in the head in terms of the size of the input positions in the head. Second, for each predicate, it infers the size function for each output position in terms of the size of the input positions. The size functions for the predicate is derived from the size functions for its clauses. Third, the size functions for predicates are used to infer the input size to each literal in a clause body.

With knowledge about the size of the arguments in a clause, the space required by each argument can be estimated. Given a specific computation model and implementation, this information may be used to estimate the space complexity of predicates.
CHAPTER 6
THE NUMBER-OF-SOLUTIONS COMPLEXITY ANALYZER

One of the main functions of the number-of-solutions complexity analyzer is the inference of the number-of-solutions complexity functions of predicates. A number-of-solutions complexity function $\text{Sol}_p : N^\omega_1 \to N^\omega_1$ of a predicate $p$ with $n$ input positions is a function representing a worst-case upper bound on the number of solutions generated by a single input to $p$ in terms of the size of the input positions of $p$. It also uses the number-of-solutions complexity functions of predicates to determine an upper bound on the number of times each body literal (namely, procedure call in the body of a procedure definition) is executed. These bounds are important for the inference of the worst-case time complexity of predicates. This chapter describes these two functions. Other functions of the number-of-solutions complexity analyzer are described in the next three chapters.

We will first classify predicates based on determinacy. A predicate is said to be determinate if it generates at most one solution for any single input to the predicate. Otherwise, it is said to be indeterminate. Determinacy analysis determines which of the predicates in a program are determinate. It has been well studied and can be performed in linear time [Me85, ST85, DW89]. Using determinacy analysis allows us to limit further number-of-solutions complexity analysis to indeterminate predicates.

We next discuss number-of-solutions complexity analysis for indeterminate predicates. To this end, we first describe a method for inferring, for each clause, a worst-case upper bound on the number of solutions generated by each body literal with respect to a single input to the clause, and represent it as an expression in terms of the size of the input positions in the head. We are particularly interested in a worst-case upper bound on the number of solutions generated by the last body literal or, equivalently, a bound on the number of solutions generated by a single input to the clause. This is followed by a discussion of how a worst-case upper bound on the number of solutions generated by
a single input to a predicate can be derived from the worst-case upper bounds on the number of solutions generated by a single input to its clauses.

Finally, we give an example and discuss the complexity of the number-of-solutions complexity analysis.

6.1 Determinacy Analysis

Determinacy analysis determines which of the predicates in a program are determinate. A simple rule for this property is given as follows: A predicate is determinate if

1. its clauses are pairwise mutually exclusive, and
2. every literal in each clause body is determinate.

The analysis for mutual exclusion between clauses was discussed in Section 4.5. Determinacy analysis can be realized via dataflow analysis in linear time [Mel85, ST85, DW89]. The general idea is to use the following iterative process. At the beginning, every predicate is assumed to be determinate. Then, at each iteration of the process, each predicate is tested against the two conditions above; any predicate that fails the test is determined to be indeterminate. This process continues until no further change occurs. It is clear that when the process terminates, all the predicates that are remained to be determinate are indeed determinate. Because the number of predicates in a program is finite, this iterative process is guaranteed to terminate.

In situations where the language allows language features such as Prolog’s cut, we can take into account their specific effects on the control strategy. For example, if cut is included in the language, then the rule for determining the determinacy of predicates can be modified as follows: A predicate is determinate if

1. its clauses are pairwise mutually exclusive, and
2. every literal that occurs after the last occurrence of cut in a clause body is determinate.

Note that the predicate cut can also affect the analysis for mutual exclusion between clauses (see Section 4.5 for details). Because both conditions in this new rule satisfy more predicates than the conditions in the earlier simple rule, this modification allows more predicates to be classified as a determinate predicate.
6.2 Clause Analysis

This section describes a method for the inference of an upper bound on the number of solutions generated by each body literal in a clause. In the discussion that follows, we assume that the size for each of the input positions in each body literal of a clause has already been computed by the argument-size complexity analyzer discussed in Chapter 5.

We first discuss the measure used to determine the number of solutions generated by a goal. Let $G$ denote the set of goals in the language and $P$ a program. A solution measure is a function $\| \cdot \|_P : G \rightarrow \mathcal{N}$. For each goal $g$, $\|g\|_P$ gives the number of solutions (namely, successful computations) in the search tree for $g$ with respect to $P$. Thus, for the following program $P$:

\[
p(1, u). \quad p(1, v). \quad p(2, w).
\]

$\|p(0, X)\|_P = 0$, $\|p(1, X)\|_P = 2$, and $\|p(2, X)\|_P = 1$. Hereafter, when the program under consideration is clear from the context, we will omit the subscript $P$ in $\| \cdot \|_P$.

Let $C$ be a clause $L_0 \leftarrow L_1, \ldots, L_m$. We use $\text{sol}_i$ to denote an expression representing an upper bound on the number of solutions generated by the literal $L_i$ (with respect to a single input to $C$) in terms of the size of the input positions in the head $L_0$. We also use $\text{sz}(\theta(a))$ to denote an upper bound on the size of the term at an argument position $a$. The expression $\text{sz}(\theta(a))$ for each input position $a$ in $L_0$ is indeed a function with domain $\Theta_C$ (namely, the set of substitutions $\theta$ such that the input arguments of $\theta(L_0)$ are ground). Hence, the expression $\text{sol}_i$ for $L_i$ is also considered to be a function with domain $\Theta_C$.

Because literals can produce multiple solutions via backtracking, the number of solutions generated by a literal (with respect to a single input to the clause) depends on the number of times this literal is executed and the number of solutions generated by each execution of this literal. For example, consider the following program:

\[
\text{- mode}(p/2, [+,-]). \\
\text{- mode}(q/2, [+,-]). \\
\text{- mode}(r/2, [+,-]). \\
p(X, Y) \leftarrow q(X, Z), r(Z, Y). \\
q(a, 1). \quad q(a, 2). \\
r(1, u). \quad r(1, v). \quad r(2, v). \quad r(2, w).
\]
We first consider the predicates \( q/2 \) and \( r/2 \). For each call to \( q/2 \), at most two solutions are generated, namely, 1 and 2. Similarly, for each call to \( r/2 \), it also generates at most two solutions, that is, \( u \) and \( v \), or \( v \) and \( w \). We now consider the predicate \( p/2 \). For each input to the clause of \( p/2 \), \( q/2 \) is executed at most once, and thus it can generate at most two solutions. On the other hand, since \( q/2 \) may generate 2 solutions, \( r/2 \) may be executed twice, once for each of the solution generated by \( q/2 \). As a result, \( r/2 \) may generate \( 2 \times 2 = 4 \) solutions, two solutions for each execution. As a matter of fact, given the call \('p(a, Y)'\), \( p/2 \) generates the following four solutions:

\[
\begin{align*}
Y &= u; \\
Y &= v; \\
Y &= v; \\
Y &= w;
\end{align*}
\]

namely, \( ||p(a, Y)|| = 4 \). Notice also that because of backtracking, the number of solutions generated by a literal is exactly the number of times its immediately succeeding literal is executed.

An algorithm for the inference of an upper bound on the number of solutions generated by each body literal in a clause is given in Figure 6.1 and described as follows. Let \( C \) be a clause \( L_0 : - L_1, \ldots, L_m \). We begin with setting \( \text{sol}_0 = 1 \), denoting that given a single input to \( C \), \( C \) can generate at most one solution if it has empty body, or the first literal in the body of \( C \) is executed at most once. Then, for each literal \( L_i \), \( 1 \leq i \leq m \), \( \text{sol}_i \) is given by the product of \( \text{sol}_{i-1} \) (namely, the number of times \( L_i \) is executed) and the number-of-solutions complexity function of \( L_i \) (namely, the number of solutions generated by executing \( L_i \) once). Let \( a_1, \ldots, a_n \) be the input positions of \( L_i \). Then we have

\[
\text{sol}_i = \text{sol}_{i-1} \times \text{Sol}_{L_i}(\text{sz}(a_1), \ldots, \text{sz}(a_n)),
\]

for \( 1 \leq i \leq m \). Since the predicates in a program are processed in a topological order of the call graph as discussed in Section 4.4, we can assume that the number-of-solutions complexity functions for the nonrecursive literals (namely, literals whose predicate symbols do not appear in a cycle involving the predicate symbol of the clause head) have already been computed. On the other hand, the number-of-solutions complexity function for recursive literals is represented in symbolic form, which are the functions we want to infer.
Algorithm 6.1

sol_clause(C, Sz): Number-of-solutions complexity analysis of a clause.

Input: A clause C: L₀ :- L₁, ..., Lₘ and the set Sz of size expressions for the input positions in each literal of C.

Output: The number-of-solutions expression solₘ for the last literal Lₘ of C.

Method:

begin
  sol₀ = 1;
  for i := 1 to m do
    let a₁, ..., aₙ be the input positions of Lᵢ;
    if Lᵢ is nonrecursive then
      solᵢ := solᵢ₋₁ × Solₜᵢₚ(sz(®a₁), ..., sz(®aₙ))
    else
      use Solₜᵢₚ(sz(®a₁), ..., sz(®aₙ)) as a symbolic expression;
      solᵢ := solᵢ₋₁ × Solₜᵢₚ(sz(®a₁), ..., sz(®aₙ))
  od
  return solₘ
end

Figure 6.1: An algorithm for number-of-solutions complexity analysis of a clause

in the analysis. As a result, if Lᵢ is nonrecursive, then solᵢ is a closed form expression that is explicitly expressed in terms of the size of the input positions in the head. Otherwise, it is symbolically expressed in terms of the size of the input positions in the head.

If the language includes the predicate cut ('!'), special treatment is needed. Operationally, cut removes all the backtracking points created in the current clause. To reflect this effect, therefore, the number-of-solutions expression for cut is set to 1, rather than taking a product. For example, consider the predicate p/2 given in the previous example with the addition of a cut between the literals q/2 and r/2:

p(X, Y) :- q(X, Z), !, r(Z, Y).
Because of the cut, the literal $r/2$ is executed once rather than twice. In this case, $p/2$ can generate at most two solutions instead of four.

**Proposition 6.2.1** Let $C$ be a clause $L_0 : -L_1, \ldots, L_m$, and $c_1, \ldots, c_n$ be the input positions in $L_0$. Suppose the size expression $sz(a)$ for each input position $a$ in $L_i$, $1 \leq i \leq m$, is nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$, and $sz(a)(\theta) \geq |\sigma(a)|_{m_a}$ for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, i - 1)$. If $\text{Sol}_{L_i}$ is nondecreasing, for $1 \leq i \leq m$, and $\text{sol}_i$ computed by Algorithm 6.1 is defined, for $0 \leq i \leq m$, then $\text{sol}_i$ is nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$, and $\text{sol}_i(\theta) \geq ||\theta(L_1, \ldots, L_i)||$, for every $\theta \in \Theta_C$.

**Proof** We prove by induction on the left-to-right order of the literals. For the base case, it is straightforward that $\text{sol}_0 = 1$ is nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$, and $\text{sol}_0(\theta) = 1$, for every $\theta \in \Theta_C$.

Now, suppose $\text{sol}_{i-1}$ is nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$, and

$$\text{sol}_{i-1}(\theta) \geq ||\theta(L_1, \ldots, L_{i-1})||,$$

for every $\theta \in \Theta_C$. Let $a_1, \ldots, a_n$ be the input positions of $L_i$. Since $\text{Sol}_{L_i}$ is nondecreasing and $sz(a_1), \ldots, sz(a_n)$ are nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$,

$$\text{Sol}_{L_i}(sz(a_1), \ldots, sz(a_n))$$

is nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$. From the hypothesis,

$$\text{sol}_i = \text{sol}_{i-1} \times \text{Sol}_{L_i}(sz(a_1), \ldots, sz(a_n))$$

is also nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$.

Since $\text{Sol}_{L_i}$ is nondecreasing, and because for every input position $a$ in $L_i$, $sz(a)(\theta) \geq |\sigma(a)|_{m_a}$, for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, i - 1)$, we have

$$\text{Sol}_{L_i}(sz(a_1)(\theta), \ldots, sz(a_n)(\theta))$$

$$\geq \max\{\text{Sol}_{L_i}(|\sigma(a_1)|_{m_a}, \ldots, |\sigma(a_n)|_{m_a}) | \sigma \in \Theta_C(\theta, i - 1)\}$$

$$\geq \max\{|\sigma(L_i)| \mid \sigma \in \Theta_C(\theta, i - 1)\},$$

for every $\theta \in \Theta_C$. Thus, we have

$$\text{sol}_i(\theta) = \text{sol}_{i-1}(\theta) \times \text{Sol}_{L_i}(sz(a_1)(\theta), \ldots, sz(a_n)(\theta))$$

$$\geq ||\theta(L_1, \ldots, L_{i-1})|| \times \text{Sol}_{L_i}(sz(a_1)(\theta), \ldots, sz(a_n)(\theta))$$
6.3 Predicate Analysis

This section describes how a number-of-solutions complexity function of a predicate can be derived from upper bounds on the number of solutions generated by its clauses. The basic idea of achieving the worst-case analysis is as follows. Recall that the clauses of a predicate are partitioned into mutually exclusive clusters as discussed in Section 4.5. An upper bound on the number of solutions generated by a predicate can, therefore, be obtained by taking the maximum among the number of solutions generated by each of the mutually exclusive clusters. Further, because the heads of all the clauses in a mutually exclusive cluster may succeed with a call to the predicate, an upper bound on the number of solutions generated by a cluster can be obtained by taking the sum of the number of solutions generated by each of the clauses in the cluster.

Let \( C \) be a clause \( L_0: \neg L_1, \ldots, L_m \) of a predicate \( p, a_1, \ldots, a_n \) be the input positions in \( L_0 \), and \( \text{sol}_m \) be the expression computed for \( L_m \) by Algorithm 6.1, denoting an upper bound on the number of solutions generated by \( C \). Since the number of solutions generated by a clause usually depends on the size of the input to the clause, we can also symbolically represent the number of solutions generated by \( C \) as \( \text{Sol}_p(sz(a_1), \ldots, sz(a_n)) \), namely,

\[
\text{Sol}_p(sz(a_1), \ldots, sz(a_n)) = \text{sol}_m.
\]

This equation explicit represents the number of solutions generated by a clause in terms of the size of the input positions in the head of the clause. If \( \text{sol}_m \) does not contain any symbolic expression except for the size of the input positions in the head, then this equation is a closed form function. On the other hand, if, in addition to the size of the input positions in the head, \( \text{sol}_m \) also contains symbolic expressions representing the number of solutions generated by the recursive literals, then this equation is in the form of difference equation.
An algorithm for deriving the number-of-solutions complexity function of a predicate is given in Figure 6.2 and described as follows. We first describe how an upper bound on the number of solutions generated by a mutually exclusive cluster can be derived from the bounds on the number of solutions generated by the clauses in the cluster. Because the heads of all the clauses in a mutually exclusive cluster may succeed with a call to the predicate, we take the sum of the bounds obtained form the clauses in the cluster as a bound on the number of solutions generated by the cluster.

As an example, consider the following predicate:

```prolog
:- mode(p/2, [+, -]).
p(1, u). p(2, v). p(2, w).
```

The first clause consists of a cluster by itself; therefore, this cluster generates at most one solution for each call to the predicate. On the other hand, the second and the third clauses together constitute a cluster; hence, for each call to the predicate, this cluster may generate as many as two solutions.

Let $M$ be a mutually exclusive cluster of a predicate $p$ with input positions $a_1, \ldots, a_n$. Also, let $e_1, \ldots, e_m$ be the number-of-solutions expressions obtained from the clauses in $M$. Then, an upper bound on the number of solutions generated by $M$ is given as

$$\text{Sol}_p(sz(a_1), \ldots, sz(a_n)) = \sum_{i=1}^{m} e_i.$$  \hspace{1cm} (6.1)

We now describe how a number-of-solutions complexity function of a predicate can be derived from the upper bounds on the number of solutions generated by the mutually exclusive clusters of the predicate. The analysis for nonrecursive predicates is straightforward. Since the complexity equations obtained from the clusters are already in closed form, we can take the maximum among them as a number-of-solutions complexity function for the predicate.

As an example, consider the predicate $p/2$ given in the previous example. Since it contains two mutually exclusive clusters and the maximum among the number of solutions generated by these two clusters is $\max\{1, 2\} = 2$, it can generate at most two solutions for any single call to the predicate, namely, $\text{Sol}_p \equiv \lambda x. 2$.

Let $p$ be a nonrecursive predicate with input positions $a_1, \ldots, a_n$. Also, let $f_1, \ldots, f_k$ be the number-of-solutions expressions obtained from the mutually exclusive clusters of $p$. 
Algorithm 6.2

sol.predicate\( (p, Sz) \): Number-of-solutions complexity analysis of a predicate.

**Input:** A predicate \( p \) and the set \( Sz \) of size expressions for the input positions in each literal of each clause of \( p \)

**Output:** The number-of-solutions complexity function \( \text{Sol}_p \) for \( p \).

**Method:**

begin

for each clause \( C_i \) of \( p \) do

let \( Sz_i \subseteq Sz \) be the set of size expressions for the input positions in each literal of \( C_i \);

\( e_i := \text{solclause}(C_i, Sz_i) \)

\( \text{od} \)

for each mutually exclusive cluster \( M_j \) of \( p \) do

\( f_j := \sum_{C_i \in M_j} e_i \)

\( \text{od} \)

if \( p \) is nonrecursive then

\( \text{Sol}_p := \max_j \{ f_j \} \)

else

\( d := \max \{ f_j \mid f_j \text{ is a difference equation} \} \);

compute the boundary conditions \( B \) due to implicit failure;

\( D := d \cup B \cup \{ f_j \mid f_j \text{ is not a difference equation} \} \);

\( \text{Sol}_p := \text{solve_difference_equation}(D) \)

\( \text{fi} \)

return \( \text{Sol}_p \)

end

Figure 6.2: An algorithm for number-of-solutions complexity analysis of a predicate
Then, a number-of-solutions complexity function for $p$ is given as

$$\text{Sol}_p(sz(A_1), \ldots, sz(A_n)) = \max\{f_1, \ldots, f_k\}. \quad (6.2)$$

The analysis for recursive predicates is more complicated. The complexity equation obtained from each mutually exclusive cluster may be a difference equation, or a closed form function serving as a boundary condition. Since a recursion may go through recursive clauses in several distinct mutually exclusive clusters, to achieve the worst-case analysis, we need to consider the maximum of the complexity functions obtained from distinct clusters that contain recursive clauses.

As an example, consider the following predicate $\text{test-gen}/3$ that generates different sets of random test cases based on if the elements of a list are larger than a given pivot number:

```prolog
:- mode(test_gen/3, [+,-,-]).
:- measure(test_gen/2, [?,length,?]).

test_gen(X, [H|_], R) :- H > X, randon1(R).

test_gen(X, [H|L], R) :- H > X, test_gen(X, L, R).

test_gen(X, [H|_], R) :- H =< X, random2(R).

test_gen(X, [H|L], R) :- H =< X, test_gen(X, L, R).
```

Suppose predicate $\text{random1}/1$ generates 1 random test case for each call, while predicate $\text{random2}/1$ generates 2 random test cases for each call. Then, the analysis will yield the following two equations for the two mutually exclusive clusters of the predicate:

$$\text{Sol}_{\text{test-gen}}(n) = \text{Sol}_{\text{test-gen}}(n - 1) + 1;$$
$$\text{Sol}_{\text{test-gen}}(n) = \text{Sol}_{\text{test-gen}}(n - 1) + 2.$$

Since a given list usually contains both elements that are larger than the pivot number and that are smaller than or equal to the pivot number, to achieve the worst-case analysis, we need to consider the maximum of the two equations, that is,

$$\text{Sol}_{\text{test-gen}}(n) = \max(\text{Sol}_{\text{test-gen}}(n - 1) + 1, \text{Sol}_{\text{test-gen}}(n - 1) + 2)$$
$$= \text{Sol}_{\text{test-gen}}(n - 1) + 2.$$

Let $p$ be a recursive predicate with input positions $a_1, \ldots, a_n$. Also, let $f_1, \ldots, f_k$ be the number-of-solutions expressions obtained from the mutually exclusive clusters that
contain recursive clauses. Then, a number-of-solutions complexity function for \( p \) can be obtained by solving the following difference equation

\[
\text{Sol}_p(sz(a_1), \ldots, sz(a_n)) = \max\{f_1, \ldots, f_k\}. \tag{6.3}
\]

together with the boundary conditions provided by the clusters that contain only nonrecursive clauses.

In addition to terminating through nonrecursive clauses, a recursion may also terminate via *implicit failure*. For example, consider the following predicate `member/2` that checks membership in a list:

\[
\text{mode}(\text{member/2}, [+,*]). \\
\text{measure}(\text{member/2}, [?, \text{length}]). \\
\text{member}(X, [X\_]). \\
\text{member}(X, [\_L]) :- \text{member}(X, L).
\]

Given a nonempty list as its second argument, `member/2` checks if its first argument is equivalent to the head of the list, and then recursively checks it against the rest of the list. The recursion terminates when the second argument of `member/2` becomes the empty list. At that point, none of the heads of the two clauses can unify with the recursive call. Thus, this predicate can be considered to have an implicit clause

\[
\text{member}(X, []) :- \text{fail}.
\]

that accounts for the termination via failure.

In these situations, it often happens that no mutually exclusive cluster has a closed form complexity function; in other words, no boundary condition is available. In the example above, the algorithm for number-of-solutions complexity analysis of a clause would infer the complexity equation

\[
\text{Sol}_{\text{member}}(n) = 1,
\]

for the first clause, and the complexity equation

\[
\text{Sol}_{\text{member}}(n) = \text{Sol}_{\text{member}}(n - 1),
\]

for the second clause. Since the two clauses are not mutually exclusive, and therefore constitute a cluster, the complexity equation for the cluster is

\[
\text{Sol}_{\text{member}}(n) = \text{Sol}_{\text{member}}(n - 1) + 1. \tag{6.4}
\]
Notice that there is no boundary condition from which this equation can be solved.

To deal with this problem, it is necessary to explicitly add boundary conditions. The number of boundary conditions we need to add depends on the type of the difference equation obtained from Equation (6.3). For example, the difference equation obtained for predicate membership/2 is a first order difference equation. Hence, one boundary condition is enough for solving that equation. The index of such a boundary condition can be taken to be the minimum input size $s$ that a recursive literal in a recursive clause may have or that the head unification of a nonrecursive clause may succeed with. If $s$ is such a size, then any call to the predicate with input of size less than $s$ will always fail. The value of a boundary condition with the index $s$ can be computed from the values of the complexity equations for the clauses with respect to the specific index $s$.

In the example of membership/2, because the head of the recursive clause can only unify with calls with input of size larger than 1, the minimum input size the recursive literal can have is 0. Also, the head of the nonrecursive clause can only unify with calls with input of size larger than 1. Thus, the index of the boundary condition is taken to be $\min(0, 1) = 0$. Since both clauses will fail when the input size is 0, or when the input is the empty list, 0 is taken to be the value of the boundary condition. Hence, we can add the following boundary condition:

$$\text{Sol}_{\text{member}}(0) = 0,$$

meaning that when the size of the input is 0, no solution is generated. Now Equations (6.4) and (6.5) can be solved to yield the expected result $\lambda x . x$, namely, every call to membership/2 with input list of length $x$ can succeed at most $x$ times.

**Theorem 6.3.1** Let $p$ be a predicate, $C_i : L_{i,0} : - L_{i,1}, \ldots, L_{i,m_i}$ be the $i$th clause of $p$, and $c_{i,1}, \ldots, c_{i,n}$ be the input positions of $p$. Suppose for each $C_i$, the expression $sz(\sigma a)$, for each input position $a$ in $L_{i,j}$, $1 \leq j \leq m$, is nondecreasing with respect to $sz(\sigma c_{i,1}), \ldots, sz(\sigma c_{i,n})$, and $sz(\sigma a)(\theta) \geq |\sigma(\theta a)|_{m_\sigma}$ for every $\theta \in \Theta_{C_i}$ and every $\sigma \in \Theta_{C_i}(\theta, j - 1)$. If the function $solve\_diffeq$ returns a nondecreasing upper bound solution for any given set of difference equations, and $Sol_p$ computed by Algorithm 6.2 is defined, then $Sol_p$ is a nondecreasing number-of-solutions complexity function for $p$.

**Proof** Let $p$ be a predicate in a program $P$, and $a_1, \ldots, a_n$ be the input positions of $p$. We will prove that by processing the predicates in $P$ in a topological order of the call
graph, every defined complexity function computed by Algorithm 6.2 is a nondecreasing number-of-solutions complexity function. We show by induction on the topological order of the call graph.

We first consider three base cases in which there is no nonrecursive literals in the clauses of a predicate. First, consider a nonrecursive predicate \( p \) consisting of clauses with empty body. For each clause \( C_i \) of \( p \), let \( \text{sol}_i \) denote the number-of-solutions expression for \( L_{i,m_i} \) computed by Algorithm 6.1. From Proposition 6.2.1, \( \text{sol}_i \) is nondecreasing with respect to \( \text{sz}(\theta_{C_{i,1}}), \ldots, \text{sz}(\theta_{C_{i,n}}) \), or equivalently, with respect to \( \text{sz}(\theta_{a_1}), \ldots, \text{sz}(\theta_{a_n}) \). For each mutually exclusive cluster \( M_k \) of \( p \), since Equation (6.1) involves only summation, and summation is nondecreasing, \( \sum_{C_i \in M_k} \text{sol}_i \) is also nondecreasing with respect to \( \text{sz}(\theta_{a_1}), \ldots, \text{sz}(\theta_{a_n}) \). Further, because Equation (6.2) involves only the maximum function, and the maximum function is nondecreasing, \( \text{Sol}_p \) is also nondecreasing with respect to \( \text{sz}(\theta_{a_1}), \ldots, \text{sz}(\theta_{a_n}) \).

From Proposition 6.2.1, \( \text{sol}_i(\theta) \geq ||\theta(L_{i,1}, \ldots, L_{i,m_i})|| \), for every \( \theta \in \Theta_{C_i} \). For each goal \( g \) of \( p \), let \( t_{a_1}, \ldots, t_{a_n} \) be the input arguments of \( p \), and \( \theta_i = \text{mgu}(g, L_{i,0}) \) for all \( i \). Then, we have

\[
\text{Sol}_p(t_{a_1}|m_{a_1}, \ldots, t_{a_n}|m_{a_n}) \geq \max_k \{ \sum_{C_i \in M_k} \text{sol}_i(\theta_i) \} \\
\geq \max_k \{ \sum_{C_i \in M_k} ||\theta_i(L_{i,1}, \ldots, L_{i,m_i})|| \} \\
\geq ||g||.
\]

Therefore, \( \text{Sol}_p \) is a nondecreasing number-of-solutions complexity function for \( p \).

Second, consider a direct recursive predicate \( p \) that involves only itself, namely, all the body literals in its clauses have the predicate symbol \( p \). For each clause \( C_i \) of \( p \), let \( \text{sol}_i \) denote the number-of-solutions expression for \( L_{i,m_i} \) computed by Algorithm 6.1. From Proposition 6.2.1, if the number-of-solutions complexity function for \( p \) is nondecreasing, then \( \text{sol}_i \) is nondecreasing with respect to \( \text{sz}(\theta_{C_{i,1}}), \ldots, \text{sz}(\theta_{C_{i,n}}) \), or equivalently, with respect to \( \text{sz}(\theta_{a_1}), \ldots, \text{sz}(\theta_{a_n}) \). For each mutually exclusive cluster \( M_k \) of \( p \), since Equation (6.1) involves only summation, and summation is nondecreasing, \( \sum_{C_i \in M_k} \text{sol}_i \) is also nondecreasing with respect to \( \text{sz}(\theta_{a_1}), \ldots, \text{sz}(\theta_{a_n}) \). Further, because Equation (6.3) involves only the maximum function, the difference equation obtained is also nondecreasing with respect to \( \text{sz}(\theta_{a_1}), \ldots, \text{sz}(\theta_{a_n}) \). Since the boundary conditions due to implicit failure are constant, they are nondecreasing. Thus, if the function \( \text{solve}_p \text{.difference}_p \)
returns a nondecreasing solution for any given set of difference equations, then $\text{Sol}_p$ is also nondecreasing with respect to $\text{sz}(\otimes\alpha_1), \ldots, \text{sz}(\otimes\alpha_n)$.

From Proposition 6.2.1, if the number-of-solutions complexity function for $p$ is a nondecreasing upper bound function, then $\text{sol}_i(\theta) \geq ||\theta(L_{i,1}, \ldots, L_{i,mi})||$, for every $\theta \in \Theta_{C_i}$. The boundary conditions due to implicit failure are computed from Equations (6.1) and (6.3) with respect to specific values; hence, they are just special cases of $\max_k \{\sum_{C_i \in \mathcal{M}_k} \text{sol}_i\}$. Let $D$ be the set of difference equations derived for $p$. For each goal $g$ of $p$, let $t_{a_1}, \ldots, t_{a_n}$ be the input arguments of $p$, and $\theta_i = \text{mgu}(g, L_{i,0})$ for all $i$. Then, if the function $\text{solve_difference_equation}$ returns an upper bound solution for any given set of difference equations, we have

$$\text{Sol}_p (\mid t_{a_1} \mid_{m_{a_1}}, \ldots, \mid t_{a_n} \mid_{m_{a_n}})$$

$$= \text{difference_equation_solver}(D)(\mid t_{a_1} \mid_{m_{a_1}}, \ldots, \mid t_{a_n} \mid_{m_{a_n}})$$

$$\geq \max_k \{\sum_{C_i \in \mathcal{M}_k} \text{sol}_i(\theta_i)\}$$

$$\geq \max_k \{\sum_{C_i \in \mathcal{M}_k} ||\theta_i(L_{i,1}, \ldots, L_{i,mi})||\}$$

$$\geq ||g||.$$  

Therefore, $\text{Sol}_p$ is a nondecreasing number-of-solutions complexity function for $p$. This also verifies the assumptions for applying Proposition 6.2.1 in the proof.

Third, consider a set of indirect recursive predicates that involve only themselves. Following the analogous argument as for the second case, and using the function $\text{solve_difference_equation}$ to solve a system of simultaneous difference equations, we can derive the same result as the second case.

Since these three cases are the only base cases, after proving these base cases, Proposition 6.2.1 can be applied for the inductive cases as for the base cases. Consequently, the inductive cases can be proved in the analogous way as for the base cases. □

If the set of difference equations of a recursive predicate can be solved to get a closed form expression, we can substitute this closed form expression for symbolic expressions in various number-of-solutions expressions for the body literals of its recursive clauses, and transform them into closed form number-of-solutions expressions. These closed form expressions are important for the inference of the time complexity for the clauses and for the predicate. However, for tail recursive clauses (namely, ones in which the recursive literal is the rightmost literal), there is no need for the substitution of symbolic expressions.
because the number-of-solutions expression for each body literal in such a clause is already in closed form.

**Theorem 6.3.2** Let \( C \) be a clause \( L_0 \leftarrow L_1, \ldots, L_m \), and \( c_1, \ldots, c_n \) be the input positions in \( L_0 \). Suppose the size expression \( sz(a) \) given for each input position \( a \) in each literal of each clause is a nondecreasing upper bound expression. If the function \( \text{solve_difference_equation} \) returns a nondecreasing upper bound solution for any given set of difference equations, and \( \text{sol}_i \), computed by Algorithms 6.1 and 6.2 is defined, for \( 0 \leq i \leq m \), then \( \text{sol}_i \) is nondecreasing with respect to \( sz(c_1), \ldots, sz(c_n) \), and \( \text{sol}_i(\theta) \geq ||\theta(L_1, \ldots, L_i)|| \), for every \( \theta \in \Theta_C \).

**Proof** From Proposition 6.2.1 and Theorem 6.3.1. \( \square \)

### 6.4 An Example

Consider the list permutation program:

```
:- mode(perm/2, [+, -]).
:- measure(perm/2, [length, length]).
perm([], []).
perm([X|Y], [R|Rs]) :- select(R, [X|Y], Z), perm(Z, Rs).
```

```
:- mode(select/3, [->+, -]).
:- measure(select/3, [? , length, length]).
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

Let \( head[i] \) denote the size of the \( i^{th} \) argument position in a clause head, and \( sol_j \) denote the number of solutions generated by the \( j^{th} \) literal of a clause. The argument-size complexity analysis of this program is discussed in Section 5.4.

We start with the analysis for the predicate select/3. Since the two clauses are not mutually exclusive, determinacy analysis infers that select/3 is indeterminate and needs further number-of-solutions complexity analysis. We first consider the nonrecursive clause. The number-of-solutions expression for the clause head is
\( \text{sol}_0 = 1 \).

Thus, the complexity equation for this clause can be represented as

\[
\text{Sol}_{\text{select}}(\text{head}[2]) = \text{sol}_0 = 1. \tag{6.6}
\]

We now consider the recursive clause. The number-of-solutions expressions for the literals of this clause are

\[
\begin{align*}
\text{sol}_0 &= 1; \\
\text{sol}_1 &= \text{sol}_0 \times \text{Sol}_{\text{select}}(\text{head}[2] - 1) = \text{Sol}_{\text{select}}(\text{head}[2] - 1).
\end{align*}
\]

Thus, the complexity equation for this clause can be represented as

\[
\text{Sol}_{\text{select}}(\text{head}[2]) = \text{sol}_1 = \text{Sol}_{\text{select}}(\text{head}[2] - 1). \tag{6.7}
\]

We now consider number-of-solutions complexity analysis for the predicate. Since the two clauses constitute a mutually exclusive cluster, we take the sum of Equations (6.6) and (6.7) to form the complexity equation for the cluster:

\[
\text{Sol}_{\text{select}}(\text{head}[2]) = \text{Sol}_{\text{select}}(\text{head}[2] - 1) + 1. \tag{6.8}
\]

Since there is no other clusters that can provide a closed form complexity function, and because Equation (6.8) is a first order difference equation, we need to add one boundary condition in order to solve Equation (6.8). We can take 0 as the index of this boundary condition because the smallest input size with which the head unification of the clauses may succeed is 1 and the smallest input size the recursive literal may have is 0. Since both clauses fail when the input size is 0, the value of the boundary condition is taken to be 0. Thus, we can add the following boundary condition:

\[
\text{Sol}_{\text{select}}(0) = 0. \tag{6.9}
\]

Now Equations (6.8) and (6.9) can be solved to yield \( \text{Sol}_{\text{select}} = \lambda x. x \), namely, the number of solutions generated by \texttt{select/3} is bounded by its input size. Since the recursive clause is tail recursive, there is no need for the substitution of symbolic expressions.

We now continue with the analysis for the predicate \texttt{perm/2}. Although the two clauses in the predicate belong to distinct mutually exclusive clusters, determinacy analysis still infers that \texttt{perm/2} is indeterminate because the first literal \texttt{select/3} in the recursive
clause is indeterminate. Therefore, we still need further number-of-solutions complexity analysis. We first consider the nonrecursive clause. The number-of-solutions expression for the clause head is

\[ s_{01} = 1. \]

Thus, the complexity equation for this clause can be represented as

\[ \text{Sol}_{\text{perm}}(0) = s_{01} = 1. \]  \hspace{1cm} (6.10)

We now consider the recursive clause. The number-of-solutions expression for the literals in this clause are

\[ s_{01} = 1; \]
\[ s_{02} = s_{01} \times \text{Sol}_{\text{select}}(\text{head}[1]) = \text{head}[1];\]
\[ s_{03} = s_{01} \times \text{Sol}_{\text{perm}}(\text{head}[1] - 1) = \text{head}[1] \times \text{Sol}_{\text{perm}}(\text{head}[1] - 1). \]

Thus, the complexity equation for this clause can be represented as

\[ \text{Sol}_{\text{perm}}(\text{head}[1]) = s_{02} = \text{head}[1] \times \text{Sol}_{\text{perm}}(\text{head}[1] - 1). \]  \hspace{1cm} (6.11)

We now consider number-of-solutions complexity analysis for the predicate. Since each of the clauses constitutes a mutually exclusive cluster, the two complexity equations also correspond to the complexity equations for the two clusters. Equations (6.10) and (6.11) can be solved to yield \( \text{Sol}_{\text{perm}} \equiv \lambda x.x! \), namely, the number of solutions generated by \( \text{perm}/2 \) is bounded by the factorial of its input size. Again, since the recursive clause is tail recursive, there is no need for the substitution of symbolic expressions.

**6.5 Complexity**

This section discusses the complexity of Algorithms 6.1 and 6.2. Let \( p, c, l, a, t, E, L, \) and \( D \) be defined as in Appendix B. Also, let \( A \) be the complexity of adding two symbolic expressions, and \( M \) the complexity of multiplying two symbolic expressions. The parameters \( A \) and \( M \) are also listed in Appendix B for later reference. Like the parameters \( E, L, \) and \( D \), the parameters \( A \) and \( M \) are indeed functions of \( p, c, l, a, \) and \( t \). However, for simplicity, we will only discuss the complexity of Algorithms 6.1 and 6.2 in terms of these parameters, and will not discuss how these parameters depend on \( p, c, l, a, \) and \( t \).
We first discuss the complexity of Algorithm 6.1. For each body literal in a clause, we may need to simplify the number-of-solutions complexity function of the literal. This simplification takes time $O(a \cdot E)$. To compute the number-of-solutions expression for a literal, we also need to take an operation to multiply two symbolic expressions. Thus, handling a literal takes time $O(a \cdot E + M)$. It takes time $O(l \cdot a \cdot E + l \cdot M)$ to consider all the literals in a clause.

We now discuss the complexity of Algorithm 6.2. Analyzing all the clauses in the predicate requires time $O(c \cdot a \cdot E + c \cdot l \cdot M)$. Since there are at most $c$ clauses, we need to add two symbolic expressions at most $c - 1$ times in Equation (6.1). Therefore, the complexity of analyzing mutually exclusive clusters is $O(c \cdot A)$.

We now consider the complexity of analyzing a predicate. Since there are at most $c$ clusters, we need to evaluate the maximum of two symbolic expressions at most $c - 1$ times in Equations (6.2) or (6.2). Hence, these equations can be computed in time $O(c \cdot l \cdot a \cdot E + c \cdot l \cdot M + c \cdot A + c \cdot L)$. Because each set of difference equations can be solved in time $D$, the complexity of computing the number-of-solution complexity function of a predicate is $O(c \cdot l \cdot a \cdot E + c \cdot l \cdot M + c \cdot A + c \cdot L + D)$.

In summary, the complexity of the number-of-solutions analysis is $O(p \cdot c \cdot l \cdot a \cdot E + p \cdot c \cdot l \cdot M + p \cdot c \cdot A + p \cdot c \cdot L + p \cdot D)$.

### 6.6 Summary

This chapter described three of the main functions of the number-of-solutions complexity analyzer. First, for each clause, the number-of-solutions complexity analyzer infers the number of solutions generated by each body literal and represents it as an expression in terms of the size of the input positions in the head. Second, the analyzer infers, for each predicate, the number of solutions generated by a single input to the predicate and represents it as a function in terms of the input size of the predicate. Third, the analyzer uses the number-of-solutions complexity functions of predicates to determine the number of times each body literal in each clause is executed.
CHAPTER 7

SOLUTION-SIZE AND RELATION-SIZE COMPLEXITY ANALYSES

This chapter describes two other main functions of the number-of-solutions complexity analyzer. They concern two declarative complexity properties of programs related to the number-of-solutions complexity of predicates. The first one is the relation size of a predicate, namely, the number of ground atoms of the predicate that are in the success set of the program under consideration. The relation size of a predicate can also be considered as the number of distinct ground tuples satisfying the predicate. The second one is the solution size of a predicate, namely, for each ground input tuple of the predicate, the number of ground atoms of the predicate that are in the success set of the program under consideration. The solution size of a predicate can also be considered as the number of distinct solutions generated by a single input to the predicate.

These properties are interesting in their own right. In addition, they also provide useful information for time complexity analysis. As an example, there are predicates that explicitly compute distinct solutions, such as the predicate setof/3 in Prolog. For these predicates, the solution-size complexity is more desirable than the number-of-solutions complexity for inferring the time complexity of predicates that depend on the output of setof/3. As another example, if a predicate is provably shown to be free of duplicates, then the solution-size complexity analysis described in this chapter usually gives a more precise estimate on the number of solutions generated by the predicate than the number-of-solutions complexity analysis described in the previous chapter. Some sufficient conditions for predicates to be duplicate-free are discussed by Maher and Ramakrishnan [MR89]. The relation-size complexity of predicates is useful because it allows us to compute a more precise estimate for the solution-size complexity of predicates.

This chapter describes methods for the inference of a worst-case upper bound on the solution-size complexity and of an upper bound on the relation-size complexity of general predicates. The next two chapters will describe two methods for the inference of an upper
bound on the relation-size complexity of two special classes of predicates involving linear
constraints over finite domains. In this chapter we only describe the analysis for a clause;
the analysis for a predicate is the same as that of the number-of-solutions complexity
analysis described in Section 6.3. We will first present a basic algorithm for each of the
complexity analyses, then discuss several optimization schemes.

7.1 A Basic Algorithm

This section presents an algorithm for the inference of a worst-case upper bound on the
solution-size complexity and of an upper bound on the relation-size complexity of a clause.
In the discussion that follows, we assume that the size for each input position in each body
literal of a clause has already been computed by the argument-size complexity analyzer
discussed in Chapter 5.

The basic idea behind the algorithm is to infer the maximum number of (distinct)
bindings possible for each variable in a clause. This inference is based on two useful
properties of unification. Given the number of bindings possible for each variable occurring
in a term, one of the properties allows us to infer the number of (distinct) instances possible
for the term. On the other hand, the other property allows us to infer the number of
bindings possible for a variable from the number of instances possible for the terms in
which the variable occurs.

We use \( B_t \) to denote the maximum number of instances possible for a (tuple of)
term(s) \( t \) in a clause. In the discussion of solution-size and relation-size complexity anal­
yses, we will overload this notation. For each term \( t \) in a clause, in the solution-size
complexity analysis, \( B_t \) denotes the number of instances for \( t \) that are generated by a
single input to the clause. On the other hand, in the relation-size complexity analysis,
it denotes the number of instances for \( t \) that are generated by all possible inputs to the
clause. Because the algorithms for the two analyses are very similar, and the quantities
denoted by the above notation share the same properties in both analyses, the overloading
of the notation makes the discussion much more concise.

We now describe the two useful properties of unification. First, consider a clause

\[ p(Y) :- q(Y), r(Y). \]

Assume that \( q/1 \) can bind \( Y \) to two values \( a \) and \( b \), while \( r/1 \) can bind it to \( a, b \) and
c. Since a successful call to a predicate ensures that different occurrences of a variable in a clause must bind to the same term, the number of bindings possible for Y in this clause should be 2, that is, the bindings to a and b. Thus, the number of bindings for a variable in a clause is bounded above by the minimum of the number of possible bindings for different occurrences of that variable:

**Proposition 7.1.1** If a variable v has n occurrences in the body of a clause and the numbers of possible bindings for these different occurrences are separately computed as $k_1, \ldots, k_n$, then $B(v) \leq \min\{k_1, \ldots, k_n\}$. □

Recall our assumption that the input arguments to any predicate are bound to ground terms when that predicate is called, and its output arguments are also bound to ground terms if the call succeeds. Proposition 7.1.1 may not hold, in general, if nonground bindings are considered [Dev90]. For example, consider the following program:

```prolog
p :- q(X), r(X).
q([Y]). q([Y,Z]).
r([a|T]). r([b|T]).
```

Predicate q/1 can bind X to two terms [Y] and [Y,Z], and predicate r/1 can also bind X to two terms [a|T] and [b|T]. However, predicate p/0 can succeed 4 times, instead of $\min\{2, 2\} = 2$ times.

Next, consider a term $f(X,Y,X)$ that contains more than one variable. Suppose X can take on two bindings, a and b, while Y can take on two bindings c and d. Then the total number of instances possible for the term $f(X,Y,X)$ is 4: $f(a,c,a), f(b,c,b), f(a,d,a), f(b,d,b)$. In general, given the number of bindings possible for the variables occurring in a term, an upper bound on the number of instances for that term is given by the product of the number of bindings possible for each of its variables:

**Proposition 7.1.2** Let $t$ be a (tuple of) term(s) in a clause. If $t$ contains a set of variables $S = \{v_1, \ldots, v_m\}$, such that $B(v_i) \leq N(v_i)$, for $1 \leq i \leq m$, then $B(t) = B_S = \prod_{i=1}^{m} N(v_i)$, with $B_S = 1$. □

Thus, if the number of possible bindings for each variable occurring in a tuple of terms $t$ has been determined, then we can define a function to compute the number of instances
for that tuple of terms. Let $t$ contain a set of variables $S = \{v_1, \ldots, v_n\}$, and $N_{\{v_i\}}$ be the determined number of bindings possible for $v_i$. We define a function, called *instance function* and denoted by $\text{instance}(t)$, as follows:

$$\text{instance}_1(t) = \prod_{i=1}^n N_{\{v_i\}}.$$ 

We associate each predicate $p$ in the program with a pair $\langle \text{Rel}_p, \text{DSol}_p \rangle$, called the *binding pattern* of $p$, where $\text{Rel}_p$ denotes an upper bound on the relation size of $p$, and $\text{DSol}_p$ denotes an upper bound on the solution size of $p$. In general, for a predicate $p$ with $n$ input positions, $\text{Rel}_p$ is in $N_{\bot,\infty}$, while $\text{DSol}_p : N_{\bot,\infty} \rightarrow N_{\bot,\infty}$ is a function in terms of the size of the input positions in $p$.

Based on Propositions 7.1.1 and 7.1.2, we can devise a simple algorithm to compute the binding pattern for each predicate. An algorithm for relation-size complexity analysis of a clause is given in Figure 7.1, and described as follows. Consider a clause

$$p(\tilde{x}_0, \tilde{y}_0) :- q_1(\tilde{x}_1, \tilde{y}_1), \ldots, q_n(\tilde{x}_n, \tilde{y}_n),$$

where $\tilde{x}_i$ are input arguments and $\tilde{y}_i$ are output arguments. Let $\text{vars}(\tilde{t})$ be the set of variables in tuple $\tilde{t}$, and $\text{lits}(v)$ be the set of literals in which variable $v$ appears. Since the predicates in a program are processed in a topological order of the call graph as discussed in Section 4.4, we can assume that the binding patterns of its nonrecursive body literals (namely, literals whose predicate symbols do not appear in a cycle involving the predicate symbol of the clause head) have already been computed. On the other hand, the binding patterns of its recursive literals are represented in symbolic form, which are the functions we want to infer in the analyses.

If the predicate $p$ is recursive, then $\text{Rel}_p$ is set to $\infty$. This is because recursive clauses usually apply recursion on (terms derived from) subterms of the input arguments, and they may successfully resolve with infinitely many calls of distinct input sizes. As a consequence, the relations they define are, in general, infinite. It is possible for a recursive predicate to define a finite relation [Big92], but $\bot$ is a safe approximation to this.

If $p$ is nonrecursive, we proceed as follows. The number of bindings, $B_{\{v\}}$, for each variable $v$ in the input arguments $\tilde{x}_0$ is estimated using Proposition 7.1.1:

$$B_{\{v\}} \leq N_{\{v\}} = \min\{\text{Rel}_j \mid j \in \text{lits}(v)\}.$$ 

\[\text{Since we will successively improve the realization of the instance function, we use subscripts to distinguish the different versions of this function.}\]
Algorithm 7.1

rel.clause(C, Sz): Relation-size complexity analysis of a clause.

Input: A clause C : p(x₀, y₀) :- q₁(x₁, y₁), ..., qₙ(xₙ, yₙ), where xᵢ are input arguments and yᵢ are output arguments, and the set Sz of size expressions for the input positions in each literal of C.

Output: The relation-size expression Rel_p for C.

Method: Let vars(ᵢ) be the set of variables in tuple i, and lits(v) be the set of literals in which variable v appears. The algorithm proceeds as follows:

begin
  if p is recursive then Rel_p := ∞
  else do
    for each variable v ∈ vars(x₀) do
      N_v := \min\{Rel_j | j ∈ lits(v)\}
    od
    for i := 1 to n do
      I_i := \min\{instance(xᵢ), Rel_q_i\};
      let a₁, ..., a_n be the input positions of qᵢ;
      O_i := \min\{I_i × DSol_{q_i}(sz(a_1), ..., sz(a_n)), Rel_q_i\};
      for each variable v ∈ vars(ᵢ) do
        N_v := \min\{O_i, Rel_j | j ∈ lits(v)\}
      od
    od
    Rel_p := instance((x₀, y₀))
  od fi
end

Figure 7.1: An algorithm for relation-size complexity analysis of a clause
Using the binding patterns of the body literals, we can then estimate the number of instances for input and output arguments in the body literals. For each literal \( q_i(\bar{x}_i, \bar{y}_i) \), the number of instances, \( B_{\{\bar{x}_i\}} \), for the input arguments \( \bar{x}_i \) is bounded by instance function applied on \( \bar{x}_i \) by Proposition 7.1.2. In addition, it should also be bounded by the size of the relation defined by \( q_i \), namely, \( \text{Rel}_{q_i} \). Thus, the smaller of these two quantities is taken to be the estimated value of \( B_{\{\bar{x}_i\}} \):

\[
B_{\{\bar{x}_i\}} \leq I_i = \min\{\text{instance}(\bar{x}_i), \text{Rel}_{q_i}\}.
\]

Let \( a_1, \ldots, a_n \) be the input positions of literal \( q_i \). Recall that \( \text{sz}(\overline{a}) \) denotes an upper bound on the size of the terms appearing at an argument position \( a \). The number of instances, \( B_{\{\bar{y}_i\}} \), for the output arguments \( \bar{y}_i \) of \( q_i \) is bounded by the product of \( B_{\{\bar{x}_i\}} \) and \( \text{DSol}_{q_i}(\text{sz}(\overline{a_1}), \ldots, \text{sz}(\overline{a_n})) \). In addition, it should also be bounded by \( \text{Rel}_{q_i} \). Their minimum is taken to be the estimated value of \( B_{\{\bar{y}_i\}} \):

\[
B_{\{\bar{y}_i\}} \leq O_i = \min\{I_i \times \text{DSol}_{q_i}(\text{sz}(\overline{a_1}), \ldots, \text{sz}(\overline{a_n})), \text{Rel}_{q_i}\}.
\]

Having the binding information about output arguments \( \bar{y}_i \), we can continue to estimate the number of bindings for each variable \( v \) becoming bound in \( \bar{y}_i \) by taking the minimum of the numbers of bindings for different occurrences of \( v \) using Proposition 7.1.1:

\[
B_{\{v\}} \leq N_{\{v\}} = \min\{O_i, \text{Rel}_j \mid j \in \text{lits}(v)\}.
\]

Since all the clauses are well-moded, once all the body literals are processed, the number of bindings possible for all the variables in the clause have been estimated. Finally, using Proposition 7.1.2 again, we can estimate the number of instances, \( B_{\{\bar{x}_0, \bar{y}_0\}} \), for the arguments in the head:

\[
B_{\{\bar{x}_0, \bar{y}_0\}} \leq \text{Rel}_h = \text{instance}((\bar{x}_0, \bar{y}_0)).
\]

As an example, consider the following program:

\[
\begin{align*}
:- \text{mode}(p/3, [+,-,-]). \\
:- \text{mode}(q/2, [+,-]). \\
:- \text{mode}(r/2, [+,-]). \\
p(X, Y, Z) :- q(X, Y), r(Y, Z). \\
q(a1, b1). \\
q(a2, b1). \\
r(b1, c1). \\
r(b2, c2). \\
r(b3, c3).
\end{align*}
\]
Since there are two unit clauses defining \( q/2 \), \( \text{Rel}_q = 2 \), and since for any input, \( q/2 \) generates at most one output, \( \text{DSol}_q \equiv \lambda x. 1 \). So the binding pattern for \( q/2 \) is \( (2, \lambda x. 1) \). Similarly, the binding pattern for \( r/2 \) is \( (3, \lambda x. 1) \). In this example, because all the predicates are nonrecursive, the corresponding solution sizes do not depend on the input size. Hence, we will ignore the size of arguments in the discussion. To compute \( \text{Rel}_p \), we first compute

\[
\text{N}_{\{x\}} = \text{Rel}_q = 2
\]

using Proposition 7.1.1. Using Propositions 7.1.1 and 7.1.2,

\[
\begin{align*}
I_1 &= \min\{\text{instance}(x), \text{Rel}_q\} = \min\{2, 2\} = 2, \\
O_1 &= \min\{I_1 \times \text{DSol}_q, \text{Rel}_q\} = \min\{2 \times 1, 2\} = 2, \\
N_{\{y\}} &= \min\{O_1, \text{Rel}_r\} = \min\{2, 3\} = 2;
\end{align*}
\]

\[
\begin{align*}
I_2 &= \min\{\text{instance}(y), \text{Rel}_r\} = \min\{2, 3\} = 2, \\
O_2 &= \min\{I_2 \times \text{DSol}_r, \text{Rel}_r\} = \min\{2 \times 1, 3\} = 2, \\
N_{\{z\}} &= O_2 = 2.
\end{align*}
\]

The relation size of \( p/3 \) is bounded by

\[
\text{instance}(x, y, z) = N_{\{x\}} \times N_{\{y\}} \times N_{\{z\}} = 8,
\]

by Proposition 7.1.2. Thus, we have \( \text{Rel}_p = 8 \), namely, there are at most 8 ground tuples that can satisfy the predicate \( p/3 \).

An algorithm for solution-size complexity analysis of a clause is given in Figure 7.2, and described as follows. The computation of \( \text{DSol}_p \), the solution size of \( p \), can be carried out in the same way as the computation of \( \text{Rel}_p \). The only differences are that at the beginning the number of bindings for each variable in the input arguments \( \bar{x}_p \) in the head is not estimated using Proposition 7.1.1, instead it is assigned to be 1. Also, at the end we only estimate the number of possible output arguments for the head, rather than both input and output arguments.

Continuing the previous example, to compute \( \text{DSol}_p \), we set \( N_{\{x\}} = 1 \) because variable \( X \) is the input. Then we follow the same procedure as the computation of \( \text{Rel}_p \). Using Propositions 7.1.1 and 7.1.2,
Algorithm 7.2

dsol.clause(C, Sz): Solution-size complexity analysis of a clause.

Input: A clause \( C : p(\bar{x}_0, \bar{y}_0) :- q_1(\bar{x}_1, \bar{y}_1), \ldots, q_n(\bar{x}_n, \bar{y}_n) \) where \( \bar{x}_i \) are input arguments and \( \bar{y}_i \) are output arguments, and the set \( Sz \) of size expressions for the input positions in each literal of \( C \).

Output: The solution-size expression \( DSol_p \) for \( C \).

Method: Let \( vars(t) \) be the set of variables in tuple \( t \), and \( lits(v) \) be the set of literals in which variable \( v \) appears. The algorithm proceeds as follows:

\[
\begin{align*}
\text{begin} \\
\quad \text{for each variable } v \in vars(\bar{x}_0) \text{ do} \\
\quad \quad N_{\{v\}} := 1 \\
\quad \text{od} \\
\quad \text{for } i := 1 \text{ to } n \text{ do} \\
\quad \quad I_i := \min\{instance(\bar{x}_i), Rel_n\}; \\
\quad \quad \text{let } a_1, \ldots, a_n \text{ be the input positions of } q_i; \\
\quad \quad O_i := \min\{I_i \times DSol_n(sz(a_1), \ldots, sz(a_n)), Rel_n\}; \\
\quad \quad \text{for each variable } v \in vars(\bar{y}_i) \text{ do} \\
\quad \quad \quad N_{\{v\}} := \min\{O_i, Rel_j \mid j \in lits(v)\} \\
\quad \quad \text{od} \\
\quad \text{od} \\
\quad DSol_p := instance(\bar{y}_0) \\
\text{end}
\]

Figure 7.2: An algorithm for solution-size complexity analysis of a clause
\[ I_1 = \min \{ \text{instance}(X), \text{Rel}_q \} = \min \{1, 2\} = 1, \]
\[ O_1 = \min \{ I_1 \times DSol_q, \text{Rel}_q \} = \min \{1 \times 2\} = 1, \]
\[ N_{\{1\}} = \min \{O_1, \text{Rel}_r\} = \min \{1, 3\} = 1; \]
\[ I_2 = \min \{ \text{instance}(Y), \text{Rel}_r \} = \min \{1, 3\} = 1, \]
\[ O_2 = \min \{ I_2 \times DSol_r, \text{Rel}_r \} = \min \{1 \times 3\} = 1, \]
\[ N_{\{2\}} = O_2 = 1. \]

Using Proposition 7.1.2 again, the number of outputs generated by a single input to \( p/3 \)
is bounded by
\[ \text{instance}(Y, Z) = N_{\{1\}} \times N_{\{2\}} = 1 \times 1 = 1. \]

So we have \( DSol_p \equiv \lambda x.1 \), namely, \( p/3 \) generates at most one distinct output for each input.

A class of predicates that are particularly useful for nondeterministic programs and need special handling are all-solutions predicates like \( \text{findall/3}, \text{bagof/3}, \text{setof/3}, \) etc. An all-solutions predicate takes a goal as the input and collects all the solutions satisfying that goal as the output (difference between the various all-solutions predicates arise from their treatment of local variables, duplicate solutions, etc). Therefore, the output argument size of an all-solutions predicate corresponds to the number of solutions (or distinct solutions) of its input goal. For each particular all-solutions predicate, the relation size and solution size can also be determined using the relation size and solution size of the input goal (see Example 9 in Appendix C).

### 7.2 Optimization Schemes

This section presents several optimization schemes for improving the basic algorithms. Proposition 7.1.2 is based on the tacit assumption that bindings for distinct variables are generated independently of each other. For example, if two variables \( X \) and \( Y \) can each get two bindings in a clause, then Proposition 7.1.2 assumes that, in the worst case, \( 2 \times 2 = 4 \) instances can be generated for a term \( f(X, Y) \). This may be overly conservative if the bindings for \( X \) and \( Y \) are not generated independently. This is the case for distinct variables that are bound by the same literal, for example, in the program
:- mode(p/2, [+,-]).
:- mode(q/3, [+,-,-]).
p(X, f(Y, Z)) :- q(X, Y, Z).
q(a, b, c). q(a, d, e).

Although q/3 generates 2 bindings for each of the variables Y and Z, only 2 solutions are possible for q/3 and thus for p/2, rather than $2 \times 2 = 4$ solutions. The following proposition gives a rule for improving the estimation in such cases. Intuitively, it says the following: if every variable in a set $S$ occurs as an output of the same literal, then the number of bindings for $S$ is bounded by the number of possible outputs generated by that literal.

**Proposition 7.2.1** Let $\bar{y}$ be the output arguments of a literal, and $O$ be the computed number of instances for $\bar{y}$. If $S \subseteq \text{vars}(\bar{y})$ is a set of variables, then $B_S \leq \min(\prod_{v \in S} N_v, O)$.

**Proof** By Proposition 7.1.2, $B_S \leq \prod_{v \in S} N_v$. Since $B_S \leq B_{\text{vars}(\bar{y})}$ and $B_{\text{vars}(\bar{y})} \leq O$, we have $B_S \leq O$. $\square$

Using Proposition 7.2.1, we can improve the instance function as follows: let $t$ be a tuple of terms in a clause with $n$ literals, the variables in $t$ be divided into $V_1, \ldots, V_n$ sets of variables such that the variables in $V_i$ become bound in literal $q_i$, and $O_i$ be the computed number of output instances for literal $q_i$. Then a new realization of the instance function can be defined as:

$$\text{instance}_2(t) = \prod_{i=1}^{n} \min\{\prod_{v \in V_i} N_v, O_i\}.$$  

As an example, for the program above, since $N_{\{\bar{y}\}} = 2$, $N_{\{\bar{z}\}} = 2$, and $O_1 = 2$, we have $\text{instance}_1(f(Y,Z)) = N_{\{\bar{y}\}} \times N_{\{\bar{z}\}} = 4$, while $\text{instance}_2(f(Y,Z)) = \min\{N_{\{\bar{y}\}} \times N_{\{\bar{z}\}}, O_1\} = 2$.

Another case in which the variable bindings are not independent concerns variables in the input and output arguments of the same literal. Consider the program

:- mode(p/2, [+,-]).
:- mode(q/2, [+,-]).
:- mode(r/2, [+,-]).
p(X, f(Y, Z)) :- q(X, Y), r(Y, Z).
q(a, b). q(a, c).
r(b, d). r(c, e).
Each of the variables Y and Z in the clause defining p/2 can get 2 bindings, but only 2 solutions are possible for r/2 and thus for p/2, instead of 2 \times 2 = 4 solutions. The dependence between these variable bindings comes from the fact that the bindings for the output variables are instantiated according to the bindings of the input variables. The following proposition gives a rule for improving the estimation in such cases. Intuitively, it states that if every variable in a set S occurs as either an input or an output of the same literal, then the number of bindings for S is bounded by the number of possible outputs generated by that literal.

Proposition 7.2.2 Let \( \bar{x} \) and \( \bar{y} \) be the input and output arguments of a literal \( q \), and \( O \) be the computed number of instances for \( \bar{y} \). If \( S \subseteq \text{vars}(\bar{x}, \bar{y}) \), then \( B_S \leq \min(\prod_{v \in S} N_v, O) \).

Proof By Proposition 7.1.2, \( B_S \leq \prod_{v \in S} N_v \). Let \( \bar{n} \) denote the input size to literal \( q \). Since \( B_{\text{vars}(\bar{x}, \bar{y})} \leq B_\bar{x} \times \text{DSol}_q(\bar{n}) \), and \( B_{\text{vars}(\bar{x}, \bar{y})} \leq \text{Rel}_q, B_{\text{vars}(\bar{x}, \bar{y})} \leq O \). Because \( B_S \leq B_{\text{vars}(\bar{x}, \bar{y})} \), we have \( B_S \leq O \). \( \square \)

Using Proposition 7.2.2, we can further improve the instance function as follows: let \( t \) be a tuple of terms in a clause with \( n \) literals, the variables in \( t \) be divided into \( V_1, \ldots, V_n \) sets of variables such that the variables in \( V_i \) become bound in literal \( q_i \), and \( O_i \) be the computed number of output instances for literal \( q_i \). The improvement can be achieved by merging these variable sets using Proposition 7.2.2 such that the number of nonempty variable sets is fewer. This merging process can proceed by considering the literals in the right-to-left order. Let \( q_i \) be the literal under consideration. If the variable set \( V_i \) corresponding to \( q_i \) is nonempty, then we can move all the variables that occur in the input arguments of \( q_i \) and that are in the variable set corresponding to a literal to the left of \( q_i \) into \( V_i \). Let \( U_1, \ldots, U_n \) be the resulting sets of variables from the merging process. Then we can define a new realization of the instance function as:

\[
\text{instance}_3(t) = \prod_{i=1}^n \min\{\prod_{v \in U_i} N_v, O_i\}.
\]

As an example, for the program above, we have \( N_{\{1\}} = 2 \), \( N_{\{2\}} = 2 \), \( O_1 = 2 \), and \( O_2 = 2 \). Since \( Y \) becomes bound in \( q/2 \) and \( Z \) becomes bound in \( r/2 \), we have \( V_1 = \{Y\} \) and \( V_2 = \{Z\} \). Thus, \( \text{instance}_2(\bar{f}(Y, Z)) = \min\{N_{\{1\}}, O_1\} \times \min\{N_{\{2\}}, O_2\} = 4 \). However, since both \( Y \) and \( Z \) appear in \( r/2 \), we have \( U_1 = \emptyset \) and \( U_2 = \{Y, Z\} \). Thus, \( \text{instance}_3(\bar{f}(Y, Z)) = \min\{N_{\{1\}} \times N_{\{2\}}, O_2\} = 2 \).
Propositions 7.2.1 and 7.2.2 give rules for improving estimation for variable bindings within a single literal. We now consider dependent variable bindings which may involve variables beyond a single literal. Consider the program:

```prolog
:- mode(p/2, [+,-]).
:- mode(q/3, [+,-,-]).
:- mode(r/2, [+,-]).
p(X, f(Y, Z)) :- q(X, Y, W), r(W, Z).
q(a, b1, c1). q(a, b2, c2).
r(c1, d1). r(c1, d2). r(c2, d3). r(c2, d4).
```

Each of the variables \( Y \) and \( W \) in the clause defining \( p/2 \) can get 2 bindings: \( b1 \) and \( b2 \), \( c1 \) and \( c2 \). Since each of the bindings for \( W \) can generate 2 bindings for variable \( Z \), \( Z \) will get a total of 4 bindings: \( d1, d2, d3, d4 \). The number of instances for \( f(Y, Z) \), therefore, should be 4: \( f(b1,d1), f(b1,d2), f(b2,d3), f(b2,d4) \), instead of \( 2 \times 4 = 8 \). The dependency between the variable bindings for \( Y \) and \( W \) is due to the fact that the variable bindings for \( Y \) and \( W \) are generated dependently by literal \( q/3 \) and the bindings for \( Z \) are instantiated according to the bindings of \( W \). In other words, because of \( W \), the variable bindings for \( Y \) and \( Z \) are generated in an indirectly dependent way.

Let \( q_i \) and \( q_j \) be two literals, \( \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \) be the corresponding input and output arguments, \( \bar{n}_j \) be the input size to \( q_j \), and \( O_i \) and \( O_j \) be the corresponding computed number of output instances. We say literal \( q_j \) dominates literal \( q_i \) if \( q_i \) and \( q_j \) satisfy the following four conditions:

1. \( \text{vars}(\bar{x}_j) \subseteq \text{vars}(\bar{x}_i, \bar{y}_i) \),

2. \( \text{instance}(\bar{x}_j) = O_i \),

3. \( \text{instance}(\bar{x}_j) \leq \text{Rel}_{q_i} \),

4. \( \text{instance}(\bar{x}_j) \times \text{DSol}_{q_j}(\bar{n}_j) \leq \text{Rel}_{q_i} \).

The following proposition provides a rule for improving the estimation in the cases where dominance relations occur between literals:

**Proposition 7.2.3** Let \( q_i \) and \( q_j \) be two literals, \( \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \) be the corresponding input and output arguments, and \( O_i \) and \( O_j \) be the corresponding computed number of output
instances. Let $S \subseteq \text{vars}(\langle \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \rangle)$ be a set of variables. If $q_j$ dominates $q_i$, then $B_S \leq \min(\prod_{v \in S} N_v, O_j)$.

**Proof** By Proposition 7.1.2, $B_S \leq \prod_{v \in S} N_v$. Also,

$$B_{\text{vars}(\langle \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \rangle)} \leq B_{\text{vars}(\langle \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \rangle)} \times D\text{Sol}_{q_i}(\bar{n}_j)$$

$$= B_{\text{vars}(\langle \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \rangle)} \times D\text{Sol}_{q_i}(\bar{n}_j)$$

(from condition 1)

$$\leq O_i \times D\text{Sol}_{q_i}(\bar{n}_j)$$

$$= \text{instance}(\bar{x}_j) \times D\text{Sol}_{q_i}(\bar{n}_j)$$

(from condition 2)

$$= I_j \times D\text{Sol}_{q_i}(\bar{n}_j)$$

(from condition 3)

$$= O_j.$$  

(from condition 4)

Because $B_S \leq B_{\text{vars}(\langle \bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j \rangle)}$, we have $B_S \leq O_j$.  

Using a proof similar to Proposition 7.2.3, it is easy to verify by induction that the result of Proposition 7.2.3 can be generalized to any number of literals:

**Proposition 7.2.4** Let $q_1, \ldots, q_n$ be literals in the body of a clause, $\bar{x}_i, \bar{y}_i, \ldots, \bar{x}_n, \bar{y}_n$ be the corresponding input and output arguments, and $O_n$ be the computed number of output instances for literal $q_n$. Let $S \subseteq \text{vars}(\langle \bar{x}_i, \bar{y}_i, \ldots, \bar{x}_n, \bar{y}_n \rangle)$ be a set of variables. If $q_{i+1}$ dominates $q_i$ for $1 \leq i < n$, then $B_S \leq \min(\prod_{v \in S} N_v, O_n)$.

Using Proposition 7.2.4, we can improve the $\text{instance}$ function once again. We apply a merging process similar to that of $\text{instance}_3$. For each literal under consideration, if the corresponding variable set is nonempty, we first move all the variables which satisfy Proposition 7.2.4 into the variable set, then we move all the variables which satisfy Proposition 7.2.2 into the variable set. Let $W_1, \ldots, W_n$ be the resulting variable sets from the merging process, and $O_i$ be the computed number of output instances for literal $q_i$. Then the new $\text{instance}$ function is defined as:

$$\text{instance}_i(t) = \prod_{i=1}^{n} \min\{\prod_{v \in W_i} N_v, O_i\}.$$ 

As an example, for the program above, we have $N_{\{v\}} = 2$, $N_{\{z\}} = 4$, $O_1 = 2$, and $O_2 = 4$. Since $Y$ becomes bound in $q/2$ and $Z$ becomes bound in $x/2$, we have $V_1 = \{Y\}$ and $V_2 = \{Z\}$. Thus, $\text{instance}_2(\langle y, z \rangle) = \min\{N_{\{v\}}, O_1\} \times \min\{N_{\{z\}}, O_2\} = 2 \times 4 = 8$. Since $Y$ and $Z$ do not appear together in any literal, we have $\text{instance}_0(\langle y, z \rangle) = \text{instance}_2(\langle y, z \rangle) = 8$. 


However, since literal $r/2$ dominates literal $q/3$, we have $W_1 = \emptyset$ and $W_2 = \{Y, Z\}$. Thus, $instance_4(f(Y, Z)) = \min\{N_{\{Y\}} \times N_{\{Z\}}, 2\} = \min\{8, 4\} = 4$.

A final improvement on the $instance$ function makes use of the relation size of test literals (namely, literals that have no output positions). Since all the arguments in a test literal are input arguments, in previous versions of the $instance$ function, the relation size of test literals can be used to restrict only the number of bindings for each single variable, but not the number of instances for a term containing more than one variable. However, test literals usually function as filters. If all the variables occurring in a term also appear in a test literal, then the number of instances for the term should also be bounded by the relation size of the test literal. Let $q_1, \ldots, q_m$ be the test literals in which all the variables of a term $t$ appear. Then the new $instance$ function is defined as:

$$instance_5(t) = \min\{instance_4(t), Rel_{q_i} \mid 1 \leq i \leq m\}.$$ 

This improvement is particularly useful in generate-and-test programs (see Example 9 in Appendix C).

Note that in Algorithms 7.1 and 7.2, we can also keep track of the parameters that maintain the upper bounds of the number of input and output instances for each predicate, or even the size of the domain for each argument position in a predicate. This may improve the accuracy of estimation at each step of the algorithm, and we may also derive optimization schemes similar to those specified in Propositions 7.2.1 - 7.2.4. In general, however, the more information is used in the algorithm, the less effective are the derived optimization schemes. Because of this, it is difficult to predict how beneficial such additional information is with regard to the precision of the final result.

### 7.3 An Example

Consider the list permutation program:

```prolog
:- mode(perm/2, [+, -]).
:- measure(perm/2, [length, length]).
perm([], []).
perm([X|Y], [R|R1|Rs]) :- select(R, [X|Y], Z), perm(Z, Rs).
```

Let $head[i]$ denote the size of the $i^{th}$ argument position in the head. We begin with the analysis for the predicate `select/3`. Since predicate `select/3` is recursive, we obtain $Rel_{select} = \infty$. We now illustrate the computation of $DSol_{select}$. We first consider the nonrecursive clause. We first set $N_{\{x\}} = N_{\{xs\}} = 1$. Using Proposition 7.2.2, the number of possible outputs for the head is

$$\text{instance}((X,Xs)) = N_{\{x\}} \times N_{\{xs\}} = 1.$$ 

Thus, we have the equation

$$DSol_{select}(head[2]) = 1.$$ 

We next consider the recursive clause. We first set $N_{\{ys\}} = N_{\{zs\}} = 1$. Using Proposition 7.1.1, we obtain

$$O_1 = N_{\{ys\}} \times DSol_{select}(\text{body}_1[2]) = DSol_{select}(head[2] - 1),$$

$$N_{\{x\}} = N_{\{zs\}} = \min\{N_{\{x\}} \times N_{\{ys\}} \times B_{\{zs\}}, O_1\} = DSol_{select}(head[2] - 1).$$

Using Proposition 7.2.2, the number of possible outputs for the head is

$$\text{instance}((X, [Y|Zs])) = \min\{N_{\{x\}} \times N_{\{ys\}} \times B_{\{zs\}}, O_1\} = DSol_{select}(head[2] - 1).$$

Thus, we have the equation

$$DSol_{select}(head[2]) = DSol_{select}(head[2] - 1).$$

The analysis for the predicate is the same as Section 6.4. The solution size for `select/3` is obtained as $DSol_{select} \equiv \lambda x. x$, namely, the predicate `select/3` will generate at most $x$ distinct solutions for an input of size $x$.

We now continue with the analysis for the predicate `perm/2`. Since predicate `perm/2` is recursive, we also obtain $Rel_{perm} = \infty$. We now illustrate the computation of $DSol_{perm}$. We first consider the nonrecursive clause. Since $[]$ is the only possible input and output, we have the equation
We now consider the recursive clause. We first set \( N_x = 1 \). Using Proposition 7.1.1, we obtain

\[
\begin{align*}
O_1 &= N_x \times DSol_{select} (body_1[2]) = head[1], \\
N_{\{x\}} &= \min\{O_1, Rel_{select}\} = head[1], \\
N_{\{y\}} &= \min\{O_1, Rel_{select}, Rel_{perm}\} = head[1], \\
O_2 &= N_{\{y\}} \times DSol_{perm}(body_2[1]) = head[1] \times DSol_{perm}(head[1] - 1), \\
N_{\{a\}} &= \min\{O_2, Rel_{perm}\} = head[1] \times DSol_{perm}(head[1] - 1),
\end{align*}
\]

Using Proposition 7.2.3, since

\[
\begin{align*}
\{y\} &\subseteq \{x, y, z\}, \\
\text{instance} (y) &= head[1] = O_1, \\
\text{instance} (y) &\leq Rel_{perm}, \\
\text{instance} (y) \times DSol_{perm}(head[1] - 1) &= head[1] \times DSol_{perm}(head[1] - 1) \\
&\leq Rel_{perm},
\end{align*}
\]

we have

\[
\text{instance} ([R|R_{\text{select}}]) = \min\{N_{\{x\}} \times N_{\{a\}}, O_2\} = head[1] \times DSol_{perm}(head[1] - 1).
\]

Notice that variables \( R \) and \( R_{\text{select}} \) are from distinct literals. Thus, we have the equation

\[
DSol_{perm}(head[1]) = head[1] \times DSol_{perm}(head[1] - 1).
\]

The analysis for the predicate is the same as Section 6.4. The solution size for \( \text{perm}/2 \) is obtained as \( DSol_{perm} \equiv \lambda x. x! \), namely, the predicate \( \text{perm}/2 \) will generate at most \( x! \) distinct solutions for an input of size \( x \).

### 7.4 Complexity

This section discusses the complexity of Algorithms 7.1 and 7.2. Let \( p, c, l, a, t, E, \) and \( M \) be defined as in Appendix B. Also, let \( S \) be the complexity of evaluating the minimum of two symbolic expressions. The parameter \( S \) is also listed in Appendix B for later reference.
Like the parameters $E$ and $M$, $S$ is indeed a function of $p$, $c$, $l$, $a$, and $t$. However, for simplicity, we will only discuss the complexity of Algorithms 7.1 and 7.2 in terms of this parameter, and will not discuss how this parameter depends on $p$, $c$, $l$, $a$, and $t$.

We first consider the complexity of evaluating the \textit{instance} function. We will discuss the complexity of evaluating \textit{instance}_5, the most expensive version. The complexity of dividing the variables into variable sets is $O(a \cdot t)$ because the number of variables in a tuple of arguments appearing in a literal is bounded by $at$ and the division can be performed in linear time. The complexity of merging variable sets can be divided into two parts. First, the complexity of merging variable sets based on Proposition 7.2.4. The property of dominance between two literals can be determined by one set inclusion operation and three minimum operations. This yields a complexity of $O(a^2 \cdot t^2 + S)$. Second, the complexity of merging variable sets based on Proposition 7.2.2. Proposition 7.2.2 involves only one set inclusion operation. Thus, the complexity of merging variable sets based on Proposition 7.2.2 is $O(l^2 \cdot a^2 \cdot t^2)$. The number of multiplication or minimum operations needed for evaluating \textit{instance}_4 is bounded by $at$. Thus, the complexity of evaluating these two operations in evaluating \textit{instance}_4 is $O(a \cdot t \cdot M + a \cdot t \cdot S)$. Determining if all the variables in a term are contained in a test literal requires one set inclusion operation of size at most $at$. Thus, the complexity of handling test literals is $O(l \cdot a^2 \cdot t^2)$. After \textit{instance}_4 is computed, the number of minimum operations needed for evaluating \textit{instance}_5 is at most $l$. This yields a complexity of $O(l \cdot S)$. In summary, the complexity of evaluating \textit{instance}_6 is $O(l^2 \cdot a^2 \cdot t^2 + l^2 \cdot S + a \cdot t \cdot M + a \cdot t \cdot S)$.

We now consider the complexity of Algorithm 7.1. The complexity is dominated by the complexity of the loop that handles the body literals. In each iteration, the computation is divided into three main steps. First, computing the number of instances for the input arguments consists of evaluating each of the \textit{instance} function and the minimum operation once. Thus, its complexity is $O(l + S)$, where $l$ is an abbreviation for the complexity of evaluating the \textit{instance} function. Second, the complexity of computing the number of instances for the output arguments is $O(a \cdot E + M + S)$, namely, a symbolic simplifications for obtaining the solution size of the literal, one multiplication, and one minimum operation. Third, the complexity of determining the number of bindings for the variables becoming bound at the output is $O(a \cdot t \cdot l \cdot S)$ since there are at most $a \cdot t$ var-
ables and \( l \) literals. In summary, the complexity of the relation-size complexity analysis is

\[ O(a \cdot t \cdot l^2 \cdot S + l \cdot a \cdot E + l^3 \cdot a^2 \cdot t^2 + l^3 \cdot S + l \cdot a \cdot t \cdot M + l \cdot a \cdot t \cdot S). \]

Algorithm 7.2 has the same loop as Algorithm 7.1, and the complexity of the loop dominates the complexity of the analysis. Thus, the solution-size complexity analysis has the same complexity as the relation-size complexity analysis.

### 7.5 Summary

This chapter described two other functions of the number-of-solutions complexity analyzer: the inference of a worst-case upper bound on the solution size and on the relation size of predicates. These two complexity properties are useful for the inference of time complexity for predicates that explicitly compute distinct solutions or that are duplicate-free.
CHAPTER 8
RELATION-SIZE COMPLEXITY ANALYSIS FOR LINEAR ARITHMETIC CONSTRAINTS

This chapter describes a method for the inference of the relation-size complexity for a class of predicates that can be “unfolded” into a set of linear arithmetic constraints connected by conjunctions and disjunctions. Each linear arithmetic constraint involves one of the following comparison operators: equality (=), disequality (≠), or inequalities (>, ≥, <, or ≤). Each variable in a constraint has a natural number interval as its domain. The method will compute an upper bound on the relation-size complexity for predicates in this class. For convenience we restrict ourselves to arithmetic constraints, but in fact the method is more general—it can be applied to any set on which notions of equality and disequality are defined.

Using the relation-size complexity analysis described in the previous chapter, we usually get trivial solutions, namely, infinity, as the relation size complexity for these operators. This is because there are, in general, infinitely many tuples of natural numbers that can satisfy predicates involving these operators. For example, the number of ground tuples that can satisfy the predicate ‘X ≥ Y + 1’ is infinity. However, using the method described in this chapter, we can often obtain more useful estimates. The method is graph-theoretic and based on the notion of the n-cliques in graphs. We first show how, using domain information, the problem of computing the relation size of a set of linear arithmetic constraints can be reduced to a problem of computing the number of n-cliques in a graph. We then describe a family of algorithms for estimating the number of n-cliques in a graph. Finally, we present a set of performance measurements for this family of algorithms.

8.1 Consistency Graph

This section shows how, using information about the underlying domain, the problem of computing the relation size of a set of linear arithmetic constraints can be reduced to
the problem of computing the number of cliques in a certain family of graphs. We will mainly consider the relation-size of a set of unary or binary constraints, namely, constraints involving at most two variables. The manipulation of general \(n\)-ary constraints, that is, constraints involving \(n\) variables, is approximated by "projecting" each \(n\)-ary constraint into a set of binary constraints. Given the natural number intervals associated with variables, the projection can be carried out via interval arithmetic. For example, consider the constraint \('X+Y+Z >= 11'\) with variables \(X, Y\) and \(Z\) ranging over the domain \([0..4]\). Since \(Z\) ranges over \([0..4]\), we can remove \(Z\) from the constraint as follows:

\[
X+Y+Z >= 11
\]
\[
\Rightarrow X+Y+[0..4] >= 11
\]
\[
\Rightarrow X+Y >= [7..11].
\]

Similarly, we can also remove, respectively, \(X\) and \(Y\) from the constraint, and obtain the following set of binary constraints:

\[
X+Y >= [7..11],
\]
\[
X+Z >= [7..11],
\]
\[
Y+Z >= [7..11].
\]

It is clear that each solution to an \(n\)-ary constraint is also a solution to the corresponding set of projected binary constraints. Therefore, the relation size of the set of binary constraints is an upper bound on that of the original \(n\)-ary constraint.

From now on, we will focus on the inference of the relation size of a set of linear binary constraints. We assume that the set of constraints involves \(n\) variables \(x_1, \ldots, x_n\) and each variable ranges over the same natural number domain \(\{d_1, \ldots, d_m\}\) of size \(m\). The set of binary constraints can be represented as a graph \(G = (V, E)\), called a consistency graph. Each vertex \(v_{ij}\) in \(V\) denotes the assignment of a domain value \(d_j\) to a variable \(x_i\), written \(x_i \leftarrow d_j\), for \(1 \leq i \leq n\) and \(1 \leq j \leq m\). There is an edge \((v_{pq}, v_{qh})\) between two vertices \(v_{pq}\) and \(v_{qh}\) if the two assignments \(x_p \leftarrow d_g\) and \(x_q \leftarrow d_h\) satisfy all the constraints involving variables \(x_p\) and \(x_q\). The two assignments are then said to be consistent. The order of a consistency graph \(G\) is \((n, m)\) if \(G\) corresponds to a set of constraints involving \(n\) variables and \(m\) domain values.

\(^1\)The variables actually need not range over the same domain: we impose this restriction for the brevity of the presentation.
The set of vertices $V_i = \{v_{i1}, \ldots, v_{im}\}$ corresponding to a variable $x_i$ is called the assignment set of $x_i$. Because two distinct domain values cannot be assigned to the same variable simultaneously, no two vertices in an assignment set are adjacent. Therefore, the consistency graph corresponding to a set of constraints involving $n$ variables is an $n$-partite graph. As an example, the consistency graph corresponding to the problem with variables $\{x_1, x_2\}$, domain $\{1, 2, 3\}$, and constraint $x_1 \geq x_2 + 1$ is shown in Figure 8.1. It is clearly a bipartite graph.

Because a solution $s$ to a set of constraints $c$ involving $n$ variables is an $n$-tuple of assignments of domain values to variables such that all the constraints in $c$ are satisfied, every pair of assignments in $s$ is consistent. Thus, if the constraints involve only conjunctions, then $s$ corresponds to an $n$-clique of the consistency graph of $c$, and the relation size of $c$ is equal to the number of $n$-cliques in the consistency graph of $c$. If the constraints involve also disjunctions, the number of $n$-cliques in the consistency graph of $c$ is an upper bound on the relation size of $c$. For instance, because there are 3 edges (2-cliques) in the consistency graph in Figure 8.1, there exist 3 solutions for the corresponding problem.

Consequently, the problem of computing an upper bound on the relation size of a set of binary constraints can be reduced to the problem of computing an upper bound on the number of $n$-cliques in the corresponding consistency graph. However, the problem of computing the number of $n$-cliques in a consistency graph is NP-hard.

**Theorem 8.1.1** The problem of computing the number of $n$-cliques in a consistency graph is NP-hard.

**Proof** We prove this theorem through the enumeration version of the graph coloring problem. The enumeration version of the graph coloring problem counts the number of different ways of coloring a graph using a given number of colors. Since this problem
involves only conjunctive binary constraints, it can be reduced to the problem of computing the number of n-cliques in a consistency graph. Because the enumeration version of the graph coloring problem is NP-hard (see Section 9.1, Theorem 9.1.1), the problem of computing the number of n-cliques in a consistency graph is also NP-hard.

8.2 An Approximation Algorithm

This section describes an approximation algorithm for estimating the number of n-cliques in a consistency graph. We will use $K(G, n)$ to denote the number of n-cliques in a consistency graph $G$. The following proposition shows that the number of n-cliques in a consistency graph can be represented in terms of the number of $(n - 1)$-cliques in the adjacency graphs corresponding to the vertices in an assignment set.

**Proposition 8.2.1** Let $G$ be a consistency graph of order $(n, m)$. Then for each assignment set $V = \{v_1, \ldots, v_m\}$,

$$K(G, n) = \sum_{i=1}^{m} K(\text{Adj}_G(v_i), n - 1).$$

**(8.1)**

**Proof** Let $G_i$ be the subgraph of $G$ induced by $N_G(v_i) \cup \{v_i\}$. Since no pair of vertices in $V$ are adjacent, $K(G, n) = \sum_{i=1}^{m} K(G_i, n)$. Because $v_i$ is adjacent to every vertex in $N_G(v_i)$, $K(G_i, n) = K(\text{Adj}_G(v_i), n - 1)$. □

Proposition 8.2.1 says that the problem of computing the number of n-cliques in a consistency graph of order $(n, m)$ can be transformed into $m$ subproblems of computing the number of $(n - 1)$-cliques in a consistency graph of order $(n - 1, m)$. However, when $m$ is larger than 1, the computation will require exponential time $O(m^n)$. This should not be surprising because the problem is NP-hard. To make it practical, therefore, we have to find an efficient means to combine the set of subgraphs $\text{Adj}_G(v_1), \ldots, \text{Adj}_G(v_m)$ in Formula (8.1) into a graph $H$ so that $K(H, n - 1)$ is an upper bound on $K(G, n)$.

To this end, we first extend the representation of a consistency graph into a weighted consistency graph. A weighted consistency graph $G = (V, E, W)$ is a consistency graph with each edge $e \in E$ associated with a weight, $W(e)$, where $W : V \times V \rightarrow \mathcal{N}$ is a function that assigns a positive integer to an edge $(u, v)$ if $(u, v) \in E$, and assigns 0 to $(u, v)$ if
The aim is to use the weights to accumulate information about the number of n-cliques.

We then define a binary operator $\oplus$, called graph addition, to combine two weighted consistency graphs that have the same set of vertices. Let $G_1 = (V, E_1, W_1)$ and $G_2 = (V, E_2, W_2)$ be two weighted consistency graphs. Then $G_1 \oplus G_2 = (V, E_{1\oplus 2}, W_{1\oplus 2})$, where $E_{1\oplus 2} = E_1 \cup E_2$, and $W_{1\oplus 2}(e) = W_1(e) + W_2(e)$, for all $e \in E_{1\oplus 2}$. Graphs $G_1$ and $G_2$ are said to be the component graphs of $G_1 \oplus G_2$. Graph addition provides a means to combine the number of n-cliques information accumulated in the component graphs. An example of graph addition is shown in Figure 8.2. We use the same notation as the one in Figure 8.1. Suppose every edge in the graphs $G_1$ and $G_2$ has weight 1. We can easily verify that every edge in the graph $G_1 \oplus G_2$ has weight 1 except for edges $(12, 21), (12, 33)$ and $(21, 33)$, which have weight 2, where $(ij, gh)$ denotes the edge joining the vertices that represent $x_i \leftarrow d_j$ and $x_g \leftarrow d_h$. 

Figure 8.2: An example of graph addition
The number of $n$-cliques, $K(G, n)$, in a weighted consistency graph $G$ of order $(n, m)$ is defined as follows. Let $S$ be the set of $n$-cliques of $G$ and $H = (V_H, E_H, W_H) \in S$ be an $n$-clique. We define $K(H, n) = \min \{W_H(e) \mid e \in E_H\}$, and $K(G, n) = \sum_{H \in S} K(H, n)$. The intuition behind this formulation is that for each $n$-clique $H$ of $G$, $K(H, n) = k$ implies that $H$ appears in at most $k$ component graphs of $G$. For instance, for the consistency graph $G_1 \oplus G_2$ in Figure 8.2, $G_1 \oplus G_2$ has three 3-cliques $(11, 23, 32), (12, 21, 33)$ and $(13, 22, 31)$, and 3-clique $(12, 21, 33)$ appears in both $G_1$ and $G_2$. Thus $K(G_1 \oplus G_2, 3) = 1 + 2 + 1 = 4$.

Let $G' = (V, E, W)$ be the weighted consistency graph corresponding to a consistency graph $G = (V, E)$. To allow us to have $K(G, n) = K(G', n)$, we define $W(e) = 1$, for all $e \in E$. With this definition, we can focus on the inference of the number of $n$-cliques in weighted consistency graphs from now on. The following proposition shows the effect of graph addition on the number of $n$-cliques in the graphs.

**Proposition 8.2.2** Let $G_1 = (V, E_1, W_1)$ and $G_2 = (V, E_2, W_2)$ be two weighted consistency graphs of order $(n, m)$. Then

$$K(G_1 \oplus G_2, n) \geq K(G_1, n) + K(G_2, n). \tag{8.2}$$

**Proof** Let $S, S_1$ and $S_2$ be the sets of $n$-cliques of $G_1 \oplus G_2, G_1$ and $G_2$ respectively. Then $S = S_1 \cup S_2$. Let $H = (V_H, E_H, W_H) \in S$ be an $n$-clique. Then $W_H(e) = W_1(e) + W_2(e)$, for all $e \in E_H$. If $H$ is in both $G_1$ and $G_2$, then $\min \{W_H(e) \mid e \in E_H\} \geq \min \{W_1(e) \mid e \in E_H\} + \min \{W_2(e) \mid e \in E_H\}$. If $H$ is in either $G_1$ or $G_2$, but not both, then $\min \{W_H(e) \mid e \in E_H\} \geq \min \{W_1(e) \mid e \in E_H\}$ and $\min \{W_2(e) \mid e \in E_H\} = 0$, or $\min \{W_H(e) \mid e \in E_H\} \geq \min \{W_2(e) \mid e \in E_H\}$ and $\min \{W_1(e) \mid e \in E_H\} = 0$. If $H$ is in neither $G_1$ nor $G_2$, then $\min \{W_H(e) \mid e \in E_H\} > 0$, and $\min \{W_1(e) \mid e \in E_H\} = \min \{W_2(e) \mid e \in E_H\} = 0$. □

Proposition 8.2.2 shows that graph addition is a means to combine the set of adjacency graphs of a consistency graph corresponding to an assignment set so that the number of $n$-cliques in the added graph is an upper bound on the sum of the number of $n$-cliques in each individual adjacency graph.

An adjacency graph of a weighted consistency graph is called a *weighted adjacency graph*. We now define the weight for each edge in a weighted adjacency graph as follows. Let $\text{Adj}_G(v) = (V_A, E_A, W_A)$ be the weighted adjacency graph of $v$ with respect
to a weighted consistency graph \( G = (V, E, W) \). For every edge \( \langle u, w \rangle \in E_A \), we define \( W_A(\langle u, w \rangle) = \min(W(\langle v, u \rangle), W(\langle v, w \rangle), W(\langle u, w \rangle)) \). This definition leads to a proposition similar to Proposition 8.2.1.

**Proposition 8.2.3** Let \( G \) be a weighted consistency graph of order \((n, m)\). Then for each assignment set \( V = \{v_1, \ldots, v_m\} \),

\[
K(G, n) = \sum_{i=1}^{m} K(\text{Adj}_G(v_i), n - 1).
\]  

**Proof** Let \( G_i = (V_{G_i}, E_{G_i}, W_{G_i}) \) be the subgraph of \( G \) induced by \( N_G(v_i) \cup \{v_i\} \). Since no two vertices in \( V \) are adjacent, \( K(G, n) = \sum_{i=1}^{m} K(G_i, n) \). Let \( \text{Adj}_G(v_i) = (V_{A_i}, E_{A_i}, W_{A_i}) \). Because \( v_i \) is adjacent to every vertex in \( N_G(v_i) \), \( \min\{W_{G_i}(e) \mid e \in E_{G_i}\} = \min\{W_{A_i}(e) \mid e \in E_{A_i}\} \). Therefore, \( K(G_i, n) = K(Adj_G(v_i), n - 1) \). \( \square \)

Combining Propositions 8.2.2 and 8.2.3, the derivation of an upper bound on the number of \( n \)-cliques in a weighted consistency graph is concluded by the following theorem.

**Theorem 8.2.4** Let \( G \) be a weighted consistency graph of order \((n, m)\). Then for each assignment set \( V = \{v_1, \ldots, v_m\} \),

\[
K(G, n) \leq K(\bigoplus_{i=1}^{m} \text{Adj}_G(v_i), n - 1).
\]  

**Proof** By Theorems 8.2.2 and 8.2.3. \( \square \)

We are now ready to present an algorithm for computing an upper bound on \( K(G, n) \) for a weighted consistency graph \( G \) of order \((n, m)\). The basic idea is to apply Theorem 8.2.4 repeatedly to a series of successively smaller graphs. By starting with the graph \( G \), at each iteration, one assignment set is removed from the graph, and a smaller graph is constructed by performing graph addition on the set of adjacency graphs corresponding to the vertices in the removed assignment set. This assignment set elimination process continues until there are only two assignment sets left. The resultant graph is now a bipartite graph. By definition, the number of 2-cliques in a weighted bipartite consistency graph is the sum of the weights of the edges (2-cliques) in the graph. The algorithm is shown in Figure 8.3.
Algorithm 8.1

\( n\text{-cliques}(G, n, m) \): Estimating the number of \( n \)-cliques in a weighted consistency graph.

**Input:** A weighted consistency graph \( G \) with order \((n, m)\).

**Output:** An upper bound on the number of \( n \)-cliques in \( G \).

**Method:**

\[
\begin{align*}
G_1 &:= G; \\
\text{for } i := 1 \text{ to } n - 2 \text{ do} &\quad G_{i+1} := \bigoplus_{j=1}^{m} \text{Adj}_G(v_i, j); \\
\text{od} &\quad \text{let } G_{n-1} = (V, E_{n-1}, W_{n-1}); \\
\text{return } &\quad \sum_{e \in E_{n-1}} W_{n-1}(e); \\
\end{align*}
\]

Figure 8.3: An algorithm for estimating the number of \( n \)-cliques in a weighted consistency graph

### 8.3 An Example

Consider the following eight-queens program. The eight-queens problem requires the placement of 8 queens on an 8 × 8 chessboard such that each queen is placed in a different row and no pair of queens are in the same column or diagonals. In the predicate \( \text{queens}/8 \), each variable \( X_i \) denotes the column in which the queen in the \( i \)-th row is placed.

\[
\begin{align*}
\text{:- mode}(\text{queens}/8, [+,-,+,-,+,-,+,-]). \\
\text{:- measure}(\text{queens}/8, [\text{nat},\text{nat},\text{nat},\text{nat},\text{nat},\text{nat},\text{nat},\text{nat}]). \\
\text{:- domain}(\text{queens}/8, [1-8,1-8,1-8,1-8,1-8,1-8,1-8,1-8]). \\
\text{queens}(X_1,X_2,X_3,X_4,X_5,X_6,X_7,X_8) :\text{ safe}([X_1,X_2,X_3,X_4,X_5,X_6,X_7,X_8]).
\end{align*}
\]
The predicate queens/8 can be unfolded into a conjunction of binary constraints: \(X_j \neq X_i\) and \(X_j \neq X_i \pm (j - i)\), for \(1 \leq i < j \leq 8\). Using the domain information and the n-cliques algorithm, we can infer that the relation size of this set of constraints, and thus the relation size of queens/8, is bounded above by 46,312 (the exact relation size is 92). However, using the relation-size complexity analysis described in Chapter 7, we can only infer that the relation size of queens/8 is infinity. This example shows that the method described in this chapter can significantly enhance the method described in the previous chapter.

### 8.4 Complexity

Let us consider the complexity of the n-cliques algorithm. Let \(v = n \cdot m\) be the number of vertices in the graph and \(e\) the number of edges in the graph. Each adjacency graph can be constructed in time \(O(v + e)\), and each graph addition can be performed in time \(O(v + e)\). Therefore, the complexity of constructing the graph \(G_{i+1}\) at each iteration is \(O(m \cdot (v+e))\). Taking the number of iterations into account, the complexity of the loop is \(O(n \cdot m \cdot (v + e)) = O(v \cdot (v + e))\). The summation of weights for a bipartite graph in the return statement can be performed in time \(O(m^2)\). Thus, the complexity for the entire n-cliques algorithm is \(O(v \cdot (v + e))\).
8.5 Optimization Schemes

In this section we investigate three optimization schemes for improving the precision of Algorithm 8.1 discussed in Section 8.2: (1) network consistency; (2) exact expansion; and (3) assignment memorization. For each scheme, applying it to the fullest extent will lead to exact solution and require exponential time. Therefore, we will examine the tradeoff between the efficiency and precision of these schemes.

8.5.1 Network Consistency

Because the basic backtracking algorithm may incur significant inefficiency, a class of network consistency algorithms has been proposed to improve the efficiency of backtracking algorithms [Mon74, Wal75, Mac77, Fre78, Gas79, HE80, MH86]. The basic idea behind network consistency algorithms is as follows. Each individual constraint in a set of constraints only makes the local consistencies (consistent assignments between two variables) explicit. By exploiting some global consistencies (namely, consistent assignments among more than two variables), we might remove beforehand some of the domain values from consideration at each stage of a backtracking algorithm.

We can use the same idea to reduce the consistency graph by removing the edges (local consistencies) that are unable to satisfy global consistencies. For example, a vertex \( v \) in an assignment set must be adjacent to a vertex in every other assignment set so that the assignment corresponding to \( v \) may be in a potential solution. Otherwise, we can remove all the edges incident on \( v \). A graph is said to be 2-consistent if all its vertices satisfy the above condition. More generally, a consistency graph \( G \) is said to be \( k \)-consistent if for every \((k-1)\)-clique \( H = (V, E, W) \) in \( G \), there exists at least one vertex \( v \) in every assignment set, apart from those assignment sets containing the vertices in \( V \), such that the subgraph induced by \( V \cup \{v\} \) is a \( k \)-clique. To incorporate an algorithm \( k \)-consistency, which achieves \( k \)-consistency of a graph, into Algorithm 8.1, we can just replace the formula \( \bigoplus_{j=1}^{m} Adj_G(v_{i,j}) \) in Algorithm 8.1 by formula \( \bigoplus_{j=1}^{m} k\text{-consistency}(Adj_G(v_{i,j})) \). The complexity for the network consistency algorithms that achieve 2-consistency and 3-consistency are \( O(n^2m^3) \) and \( O(n^3m^5) \) respectively [MF85]. Using algorithms that achieve 2-consistency and 3-consistency, the estimates for the 8-queens problem of Section 8.3 are 46,312 and 44,702, respectively.
8.5.2 Exact Expansion

The exact expansion scheme tries to balance the precision of Formula (8.3) and the efficiency of Formula (8.4). The intent is to first expand Formula (8.3) some number of times to exactly generate some subproblems, then use Formula (8.4) to approximately solve the expanded subproblems. Each time Formula (8.3) is used, the complexity of the entire algorithm will increase by a factor of \(O(m)\). Using algorithms that expand Formula (8.3) once and twice, the estimates for the 8-queens problem of Section 8.3 are 4,736 and 310, respectively.

8.5.3 Assignment Memorization

In Algorithm 8.1, the weight of an edge \(e\) at the end of the \(i^{th}\) iteration denotes the number of component graph sequences \(A_1, \ldots, A_i\) in which the edge \(e\) occurs, where \(A_j\) is the component graph in which the edge \(e\) occurs at the \(j^{th}\) iteration. Equivalently, it denotes the number of partial assignments to variables \(x_1, \ldots, x_i\) with which the two assignments corresponding to the edge \(e\) are consistent. Due to the lack of partial assignment information, graph addition is performed without knowing which partial assignments the weight contributes to. The assignment memorization scheme tries to memorize the weight as well as some partial assignment information so that we can take advantage of this information and perform graph addition in a more precise manner.

Operationally, to memorize the most recent variable assignment, each edge of the weighted consistency graph needs to maintain an array of \(m\) weights instead of a single weight. At the end of the \(i^{th}\) iteration of the \(n\)-cliques algorithm 8.1, the \(j^{th}\) element of the weight array of an edge \(e\), \(W(e)[j]\), corresponds to the number of partial assignments to variables \(x_1, \ldots, x_i\) to which \(e\) contributes with \(x_i \leftarrow d_j\). Therefore, let \(G_{i+1} = (V, E, W)\) be the graph at the end of the \(i^{th}\) iteration, and \(\text{Adj}_{G_i}(u, v) = (V_j, E_j, W_j)\) be the \(j^{th}\) adjacency graph at the \(i^{th}\) iteration. For each edge \(e \in E\), we have \(W(e)[j] = \sum_{k=1}^{m} W_j(e)[k]\).

The memorization of \(k\) variable assignments will cost the entire algorithm a factor of \(O(m^k)\) in both time and space. Using algorithms that memorize the most recent one and two variable assignments, the estimates for the 8-queens problem of Section 8.3 are 4,060 and 526, respectively.
8.6 Performance Measurements

This section gives the performance measurements of the algorithms described in the previous sections. Experiments are performed on a set of randomly generated consistency graphs. The edges in the graphs are chosen independently and with the probability 0.75. The probability 0.75 is chosen so that the graphs have reasonably high density, and thus a reasonable number of cliques.

Apart from Algorithm 8.1, we also incorporate network consistency, exact expansion and assignment memorization algorithms into Algorithm 8.1. The 2-consistency and 3-consistency algorithms achieve, respectively, 2-consistency and 3-consistency for each constructed adjacency graph. The 1-expansion and 2-expansion algorithms expand Formula (8.3) once and twice, respectively. The 1-memorization and 2-memorization algorithms memorize, respectively, the most recent one and two variable assignments. We apply each algorithm to 16 random graphs for each of the orders (4,4), (5,5), ..., (8,8).

The performance results are shown in Figure 8.4. Figures 8.4(a), (c) – (h) display the relative error of the algorithms with respect to the order of the consistency graphs. The relative error is defined as $(K - K^*)/K^*$, where $K$ and $K^*$ denote, respectively, the estimated and the exact number of $n$-cliques in the graph. The results show that the relative error of the algorithms increases as the order of the graph grows. Because the number of graph additions performed increases as the order of the graph grows, the estimation error accumulated via graph addition also increases. Furthermore, since the number of $n$-cliques in a graph usually grows exponentially with respect to the order of the graph, the growth rate of the relative error is also high. However, the relation between two exact solutions usually also reflects upon the corresponding estimated solutions. This result is shown in Figure 8.4(b). Here we compare the relation between two exact solutions with the relation between the two corresponding estimated solutions for all pairs of the 16 random graphs. The relativity ratio is the ratio between the number of pairs that preserve the relation and the total number of pairs.

The average behavior of the performance measurements is given in Figure 8.5. The results show that the low-order network consistency algorithms do not work well for dense graphs. In general, as the density of graphs becomes higher, the graphs are more likely to be low-order consistent. However, for applications where network consistency algorithms
Figure 8.4: Experimental measurements for the n-cliques algorithms
Figure 8.5: Average behavior of experimental measurements for the $n$-cliques algorithms
work well for finding solutions, the scheme of using network consistency algorithms for estimating the number of solutions will also work well. On the other hand, the exact expansion algorithms and the assignment memorization algorithms can still significantly improve the estimation precision for dense graphs.

Because these algorithms can achieve any degree of precision by paying the price on efficiency, it is also possible to construct adaptive algorithms that will adaptively decide the amount of efforts to invest based on the order and density of the graphs and the precision requirements.

8.7 Summary

This chapter described a method for the inference of an upper bound on the relation-size complexity for a class of predicates that can be unfolded into a set of linear arithmetic constraints connected by conjunctions and disjunctions. Each linear arithmetic constraint involves one of the following comparison operators: $=$, $\neq$, $>$, $\geq$, $<$, or $\leq$. Each variable in a constraint has a natural number interval as its domain.

Using the relation-size complexity analysis described in the previous chapter, we usually get trivial estimates as the relation size for these operators. However, using the method described in this chapter, we can often obtain more useful relation size estimates. The method is graph-theoretic and based on the notion of the $n$-cliques in graphs. We first showed how, using domain information, the problem of computing the relation size of a set of linear arithmetic constraints can be reduced to the problem of computing the number of $n$-cliques in a graph. We then described a family of algorithms for estimating the number of $n$-cliques in a graph. Finally, we presented a set of performance measurements for this family of algorithms.
CHAPTER 9

RELATION-SIZE COMPLEXITY ANALYSIS FOR BINARY EQUALITY AND DISEQUALITY CONSTRAINTS

This chapter describes a method for the inference of the relation-size complexity for a class of predicates that can be "unfolded" into a conjunction of binary equality and disequality constraints of the forms \( x = y \) and \( x \neq y \). The variables in the constraints are assumed to range over the same finite domain. Since this class of constraints is somewhat restricted, it is possible to compute both upper and lower bounds on the relation-size complexity.

The method is graph-theoretic and based on the notion of the chromatic polynomial of a graph, which is a polynomial representing the number of different ways in which the graph can be colored by a given number of colors. We first show how the problem of computing the relation size of a conjunction of binary equality and disequality constraints can be reduced to the problem of computing the chromatic polynomial of a graph. We then describe an algorithm for estimating upper and lower bounds on the chromatic polynomial of graphs. Finally, we give a performance analysis and a set of performance measurements for this algorithm.

9.1 Constraint Graph

This section shows how the problem of computing the relation size of a conjunction of binary equality and disequality constraints can be reduced to the problem of computing the chromatic polynomial of a graph. A conjunction of binary equality and disequality constraints can be represented as a graph, called a constraint graph. Each vertex in the graph corresponds to a variable in the constraints. There is an edge between two vertices if there is a disequality constraint involving the two corresponding variables. Two vertices can be merged into a single vertex if there is an equality constraint involving the two
corresponding variables; the set of edges incident to the merged vertex is the union of the sets of edges incident to the two original vertices.

The chromatic number of a graph $G$, written $\chi(G)$, is the minimum number of colors necessary for coloring $G$ such that no adjacent vertices have the same color. The chromatic polynomial of a graph $G$, denoted by $C(G, x)$, is a polynomial in $x$ representing the number of different ways in which $G$ can be colored by using no more than $x$ colors.

If we view each value in the domain as a distinct color, then each solution to the conjunction of binary equality and disequality constraints corresponds to a coloring of the corresponding constraint graph. Accordingly, the relation size of the set of constraints is exactly the number of different ways in which the constraint graph can be colored using no more than the number of colors equal to the cardinality of the domain. In other words, if the chromatic polynomial $C(G, x)$ of a constraint graph $G$ is available and the cardinality of the domain is $k$, then the relation size of the corresponding set of constraints is the value of $C(G, x)$ at $k$. Therefore, the problem of computing the relation size of a conjunction of binary equality and disequality constraints is reduced to the problem of computing the chromatic polynomial of the corresponding constraint graph. Unfortunately, the problem of computing the chromatic polynomial of a graph is NP-hard.

**Theorem 9.1.1** The problem of computing the chromatic polynomial of a graph is NP-hard.

**Proof** We prove this theorem through the graph $k$-colorability problem. The graph $k$-colorability problem decides whether a graph $G$ can be colored using $k$ colors, that is, whether the value of $C(G, x)$ at $k$ is larger than 0. Since the graph $k$-colorability problem is NP-complete [Kar72], the problem of determining the value of $C(G, x)$ at $k$ is NP-hard. Thus, the problem of computing the chromatic polynomial of a graph is also NP-hard.

### 9.2 An Approximation Algorithm

This section describes an approximation algorithm for estimating the chromatic polynomial of graphs. We will derive both upper and lower bounds on the chromatic polynomial of a graph. Because the value of the chromatic polynomial $C(G, x)$ is always nonnegative for any given graph $G$ and natural number $x$, we will assume that any polynomial $P(x)$ really means the function $\max(P(x), 0)$. 
Figure 9.1: An example graph

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**accumulating subgraphs**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**interfacing subgraphs**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.2: The accumulating and interfacing subgraphs of the graph in Figure 9.1

Let $G = (V, E)$ be a graph of order $n$ and $\omega = v_1, \ldots, v_n$ be an ordering on $V$. We define two sequences of subgraphs of $G$ according to $\omega$. The first is a sequence of subgraphs $G_1(\omega), \ldots, G_n(\omega)$, called **accumulating subgraphs**, where $G_i(\omega)$ is the subgraph induced by $V_i = \{v_1, \ldots, v_i\}$, for $1 \leq i \leq n$. The second is a sequence of subgraphs $G'_2(\omega), \ldots, G'_n(\omega)$, called **interfacing subgraphs**, where $G'_i(\omega)$ is the adjacency graph $\text{Adj}_{G_{i-1}(\omega)}(v_i)$, for $1 < i \leq n$. For example, consider the graph shown in Figure 9.1. The imposed ordering is denoted by the labels of vertices. The corresponding accumulating subgraphs and interfacing subgraphs are shown in Figure 9.2.

The following proposition gives an upper bound and a lower bound on the chromatic polynomial of a graph in terms of, respectively, the chromatic number and the order of
Proposition 9.2.1 Let $G = (V, E)$ be a graph of order $n$ and $\Omega$ be the set of all possible orderings on $V$. Suppose the interfacing subgraphs of $G$ corresponding to an ordering $\omega \in \Omega$ are $G_2(\omega), \ldots, G_n(\omega)$. Then

$$\max_{\omega \in \Omega} \left\{ x \prod_{i=2}^{n} (x - |G_i(\omega)|) \right\} \leq C(G, x) \leq \min_{\omega \in \Omega} \left\{ x \prod_{i=2}^{n} (x - \chi(G_i')) \right\}. \quad (9.1)$$

Proof Suppose $G_1(\omega), \ldots, G_n(\omega)$ are the accumulating subgraphs of $G$ corresponding to an ordering $\omega$. The proof is by induction on $G_j(\omega)$, for $1 \leq j \leq n$. In base case, $G_1(\omega)$ is a graph consisting of one vertex $v_1$, so $C(G_1(\omega), x) = x$. Suppose $x \prod_{i=2}^{j} (x - |G_i(\omega)|) \leq C(G_j, x) \leq x \prod_{i=2}^{j} (x - \chi(G_i'))$ for some $j, 1 \leq j < n$. Then consider adding the vertex $v_{j+1}$ and associated edges into $G_j(\omega)$ to form $G_{j+1}(\omega)$. For the lower bound, since $|G'_{j+1}(\omega)|$ is the degree of $v_{j+1}$ in $G_{j+1}(\omega)$, we have $(x - |G'_{j+1}(\omega)|)C(G_j(\omega), x) \leq C(G_{j+1}(\omega), x)$.

From the hypothesis, we obtain

$$x \prod_{i=2}^{j+1} (x - |G_i(\omega)|) \leq (x - |G'_{j+1}(\omega)|)C(G_j(\omega), x) \leq C(G_{j+1}(\omega), x).$$

For the upper bound, since $\chi(G'_{j+1}(\omega))$ is the minimum number of colors necessary for coloring $G'_{j+1}(\omega)$, we have $C(G_{j+1}(\omega), x) \leq (x - \chi(G'_{j+1}(\omega)))C(G_j(\omega), x)$. From the hypothesis, we obtain

$$C(G_{j+1}(\omega), x) \leq (x - \chi(G'_{j+1}(\omega)))C(G_j(\omega), x) \leq x \prod_{i=2}^{j+1} (x - \chi(G_i')).$$

Since $G_n(\omega) = G$, we have the following formula

$$x \prod_{i=2}^{n} (x - |G_i(\omega)|) \leq C(G, x) \leq x \prod_{i=2}^{n} (x - \chi(G_i')). \quad (9.2)$$

Finally, because Formula (9.2) holds for any ordering $\omega$, Formula (9.1) follows. \Box

The upper bound in Formula (9.1) is in terms of the chromatic number of interfacing subgraphs. Unfortunately, the computation of the chromatic number of a graph is NP-complete [Kar72].Nevertheless, it turns out that if each of the chromatic numbers in Formula (9.1) is replaced by any of its lower bounds, the resultant expression is still an upper bound on the chromatic polynomial. The following theorem by Bondy [Bon69] gives a lower bound on the chromatic number of a graph that can be computed efficiently.
Theorem 9.2.2 Let $G$ be a graph of order $n$, $d(1) \geq \cdots \geq d(n)$ be the degrees of nodes in $G$ and $\sigma_j$ be defined recursively by

$$\sigma_j = n - d(\sum_{i=1}^{j-1} \sigma_i + 1).$$

Suppose $k \leq n$ is some integer satisfying

$$\sum_{j=1}^{k-1} \sigma_j < n.$$  \hfill (9.3)

Then $\chi(G) \geq k$. $\Box$

We define $\rho(G)$, called the Bondy number of a graph $G$, to be the largest integer $k \leq n$ satisfying Formula (9.3). Then an upper bound on the chromatic polynomial of a graph can be expressed in terms of the Bondy number of interfacing subgraphs.

Proposition 9.2.3 Let $G = (V, E)$ be a graph of order $n$ and $\Omega$ be the set of all possible orderings on $V$. Suppose the interfacing subgraphs of $G$ corresponding to an ordering $\omega \in \Omega$ are $G'_1(\omega), \ldots, G'_n(\omega)$. Then

$$\max_{\omega \in \Omega} \{\sum_{i=2}^{n} (x - |G'_i(\omega)|)\} \leq C(G, x) \leq \min_{\omega \in \Omega} \{\prod_{i=2}^{n} (x - \rho(G'_i(\omega)))\}. \hfill (9.4)$$

Proof By Theorem 9.2.2 and an argument parallel to the upper bound argument of Proposition 9.2.1. $\Box$

It is clear that carrying out the computation of the maximum and minimum among all possible orderings as represented in Formula (9.4) is impractical. As a result, we will employ a representative ordering to compute upper and lower bounds on the chromatic polynomial. An ordering on the vertices of a graph is said to be a perfect elimination ordering if all the corresponding interfacing subgraphs are cliques [RTL76]. The following proposition states an appealing property of perfect elimination ordering.

Proposition 9.2.4 Let $G = (V, E)$ be a graph of order $n$ and $\omega$ be an ordering on $V$. If $\omega$ is a perfect elimination ordering, then

$$\prod_{i=2}^{n} (x - |G'_i(\omega)|) = C(G, x) = \prod_{i=2}^{n} (x - \rho(G'_i(\omega))).$$
Proof By the fact, easily proved, that \( \rho(K) = |K| \) for any clique \( K \). □

One implication of Proposition 9.2.4 is that if there exists a perfect elimination ordering for a graph and it can be generated efficiently, then the chromatic polynomial of that graph can be computed efficiently. Unfortunately, not every graph has a perfect elimination ordering. A graph is called triangulated if every cycle of length greater than 3 has an edge joining two nonconsecutive vertices of the cycle. Dirac [Dir61] and Rose [Ros70] have shown that a graph is triangulated if and only if it has a perfect elimination ordering. As an example, the graph in Figure 9.1 is a triangulated graph, and the labels of vertices show a perfect elimination ordering. On the other hand, since no complete bipartite graph \( K(n,m) \) with \( n \) and \( m \) greater than 1 is triangulated, they have no perfect elimination ordering.

We now describe the ordering that will be employed to compute bounds on the chromatic polynomials. For obvious reasons, we will try to generate a perfect elimination ordering whenever it is possible. The ordering generation is an iterative graph reduction process, and the ordering is generated in reverse order.

At each iteration, we search for a vertex such that its adjacency graph is a clique. If such a vertex \( v \) exists, it is chosen as the vertex to generate. It is clear that if the graph resulted from removing \( v \) has a perfect elimination ordering, then the original graph containing \( v \) also has a perfect elimination ordering. The latter ordering can be constructed by simply adding \( v \) at the rear of the former ordering. The process continues by removing \( v \) from the graph and proceeding to the next iteration.

On the other hand, if such a vertex \( v \) does not exist, then we choose a vertex \( w \) that has the smallest degree to generate. The basic idea behind this heuristic is that, to yield nontrivial lower bounds for larger number of values, we demand the maximum order of the interfacing subgraphs in Formula 9.2 to be as small as possible. Since the ordering generation is a graph reduction process, the degree of vertices or the order of the adjacency graph of vertices will become smaller when the process goes on. Therefore, when the generation of a perfect elimination ordering cannot continue, we greedily choose the vertex that has the smallest degree to generate. The process continues by removing the chosen vertex \( w \) from the graph and proceeding to the next iteration. The whole process terminates when all the vertices are generated.

Notice also that the ordering among the vertices whose adjacency graph is a clique is
not crucial. In the process, once a vertex has a clique as its adjacency graph, its later adjacency graphs will also be cliques. This is because the removal of vertices from a clique still results in a clique.

We define an ordering \( \omega_0 \), called a perfect-smallest-last ordering, as follows. Let \( G_1, \ldots, G_n \) be a sequence of subgraphs of a graph \( G \) of order \( n \), with

1. \( G_1 = (V_1, E_1) = G \) and \( G_{i+1} = (V_{i+1}, E_{i+1}) = G_i - \{v_i\} \),
2. \( v_i = \begin{cases} v & \text{if there is a vertex } v \in V_i \text{ such that } Adj_{G_i}(v) \text{ is a clique} \\ \min_{v \in V_i} \{d_{G_i}(v)\} & \text{otherwise,} \end{cases} \)

for \( 1 \leq i \leq n \), then \( \omega_0 = v_n, \ldots, v_1 \). One important property of perfect-smallest-last ordering is stated in the following proposition:

**Proposition 9.2.5** If a graph \( G \) is triangulated, then a perfect-smallest-last ordering on the vertices of \( G \) is a perfect elimination ordering.

**Proof** Let \( G \) be a graph of order \( n \) and \( G_1, \ldots, G_n \) be the sequence of subgraphs of \( G \) corresponding to a perfect-smallest-last ordering \( \omega_0 \). Assume that \( \omega_0 \) is not a perfect elimination ordering, and that \( j, 1 \leq j \leq n \), is the maximum integer such that \( \{v \in V_j : Adj_{G_j}(v) \text{ is a clique}\} = \emptyset \). Note that the subgraphs \( Adj_{G_j}(v) \) for vertices \( v \) in \( G_j \) are not cliques for \( 1 \leq i \leq j \), and this fact is independent of the ordering on the vertices \( v_1, \ldots, v_{j-1} \). So it is impossible that \( G \) is triangulated. \( \square \)

We are now ready to present an algorithm for computing an upper bound and a lower bound on the chromatic polynomial of a graph based on Formula (9.4) and the perfect-smallest-last ordering. The algorithm is shown in Figure 9.3. Apart from the bounds, this algorithm also computes a mean of the bounds. Let \( L(G, x) \) denote the lower bound \( x \prod_{i=2}^{n} (x - |G_i(\omega_0)|) \) and \( U(G, x) \) denote the upper bound \( x \prod_{i=2}^{n} (x - \rho(G_i(\omega_0))) \) in Formula (9.4), respectively, with \( \omega_0 \) being a perfect-smallest-last ordering on the vertices of \( G \). We estimate the chromatic polynomial of a graph \( G \) to be

\[
\hat{C}(G, x) = \frac{2 \times U(G, x) \times L(G, x)}{U(G, x) + L(G, x)},
\]

(9.5)

the harmonic mean of \( U(G, x) \) and \( L(G, x) \). Notice that although \( \hat{C}(G, x) \) is an estimate of \( C(G, x) \), \( \hat{C}(G, x) \) itself may be a rational function, but not a polynomial.
Algorithm 9.1

c._polynomial(G): Estimating the chromatic polynomial of a graph.

Input: A graph G.

Output: An estimate of the chromatic polynomial of G.

Method:

begin
    \( G_1 = (V_1, E_1) := G; \)
    \( U(G, x) := L(G, x) := 1; \)
    for \( i := 1 \) to \( n \) do
        if there is a vertex \( v \in V_i \) such that \( \text{Adj}_{G_i}(v) \) is a clique then
            \( v_i := v; \)
        else
            \( v_i := \min_{v \in V_i} \{d_{G_i}(v)\}; \)
        fi
        \( U(G, x) := U(G, x) \times (x - \rho(G_i(v_i))); \)
        \( L(G, x) := L(G, x) \times (x - |G_i(v_i)|); \)
        \( G_{i+1} = (V_{i+1}, E_{i+1}) := G_i - \{v_i\}; \)
    od
    \( \hat{C}(G, x) := \text{the harmonic mean of } U(G, x) \text{ and } L(G, x); \)
end

Figure 9.3: An algorithm for estimating the chromatic polynomial of a graph
9.3 An Example

Consider the following scheduling program. The predicate time_slot/1 is a time-slot generator. The predicate constraints/5 specifies a set of binary disequality constraints on its arguments. The predicate scheduling/5 generates all the schedules satisfying the constraints specified in constraints/5.

```prolog
:- mode(scheduling/5,[-,-,-,-,-]).
:- measure(scheduling/5,[size,size,size,size,size]).
scheduling(A,B,C,D,E) :-
    time_slot(A), time_slot(B), time_slot(C), time_slot(D), time_slot(E),
    constraints(A,B,C,D,E).

:- mode(constraints/5,[+,+,+,+,+]).
:- measure(constraints/5,[size,size,size,size,size]).
:- domain(constraints/5,[[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]]).
constraints(A,B,C,D,E) :-
    A \= B, A \= C, A \= D, A \= E, B \= C, C \= D, C \= E.

:- mode(time_slot/1,[-]).
:- measure(time_slot/1,[size]).
time_slot(1). time_slot(2). time_slot(3). time_slot(4). time_slot(5).
```

The constraint graph corresponding to the predicate constraints/5 is that shown in Figure 9.1. The estimated chromatic polynomial for that graph is \(x(x - 1)(x - 2)^3\). In this case, since the graph is triangulated, it is also the exact chromatic polynomial of the graph. In fact, both the computed upper and lower bounds are also \(x(x - 1)(x - 2)^3\), as implied by Propositions 9.2.4 and 9.2.5.

Since there are five time slots, the relation size of constraints/5 is the value of \(x(x - 1)(x - 2)^3\) at \(x = 5\), namely, 540. Using the \textit{instance5} function, we can infer that both the solution size and relation size of scheduling/5 are \(\min\{5^5, 540\} = 540\). On the other hand, using the relation-size complexity analysis described in Chapter 7, we can only conservatively infer that the relation size of constraints/5 is infinite, which in turn
leads us to infer that both the solution size and relation size of scheduling/5 are 5°.

9.4 Complexity

We now consider the complexity of Algorithm 9.1. Let \( n \) and \( m \) be, respectively, the number of vertices and edges in the graph. We first consider the test in the if statement. Detecting if a graph of order \( k \) is a clique can be performed in time \( O(k^2) \). At the worst case, when no adjacency graph is a clique and the detection has to be performed for every vertex, the time required is

\[
O(n + \sum_{j=1}^{n} d_{G_i}(v_j)) = O(n + (\sum_{j=1}^{n} d_{G_i}(v_j))^2) = O(n + m^2).
\]

The minimum operation in the else branch can be performed in time \( O(1) \). Thus, altogether, the complexity of the if statement is \( O(n + m^2) \).

The symbolic multiplications of polynomials for \( U(G, x) \) and \( L(G, x) \) can be performed in time \( O(n) \) since the order of \( U(G, x) \) and \( L(G, x) \) is at most \( n \). The computation of \( \rho(G_i(v_i)) \) demands a sorting step, so it requires time \( O(n \cdot \log n) \). The updating from \( G_i \) to \( G_{i+1} \) can be performed in time \( O(n + m) \). Put together, the complexity for each iteration of the for statement is \( O(n \cdot \log n + n \cdot m^2) \). Taking the number of iterations into account, the time demanded for the for statement is \( O(n^2 \cdot \log n + n \cdot m^2) \).

The computation of the harmonic mean needs a symbolic multiplication and a symbolic addition. It can be performed in time \( O(n^2) \). Therefore, the complexity of the entire algorithm is \( O(n^2 \cdot \log n + n \cdot m^2) \). This complexity analysis leads to the following theorem:

**Theorem 9.4.1** The problem of computing the chromatic polynomial of a triangulated graph is solvable in polynomial time.

**Proof** By Propositions 9.2.4, 9.2.5 and the complexity analysis of Algorithm 9.1. \( \square \)

9.5 Performance Analysis and Measurements

This section investigates the performance behavior of the Algorithm 9.1. Since the value of \( C(G, x) \) are usually very large, we will consider the relative error

\[
\Delta = \frac{|\hat{C}(G, x) - C(G, x)|}{C(G, x)}
\]

in performance analysis.
The maximum of the relative error occurs when \( C(G, x) = U(G, x) \) or \( C(G, x) = L(G, x) \), namely, at the two extremes. This maximum is minimized when the relative errors for \( C(G, x) = U(G, x) \) and \( C(G, x) = L(G, x) \) are in fact equal. The harmonic mean just provides such a property. Using the harmonic mean, we have

\[
\Delta \leq \frac{U(G, x) - \hat{C}(G, x)}{U(G, x)} = \frac{\hat{C}(G, x) - L(G, x)}{L(G, x)} = \frac{U(G, x) - L(G, x)}{U(G, x) + L(G, x)}.
\]

Since both \( U(G, x) \) and \( L(G, x) \) are nonnegative, we have

\[
\Delta \leq \frac{U(G, x) - L(G, x)}{U(G, x) + L(G, x)} \leq 1,
\]

for any natural number \( x \). This also implies that we always have \( 0 \leq \hat{C}(G, x) \leq 2C(G, x) \). This statement is always true if we can compute both a lower bound and an upper bound on the measure we are interested in. On the other hand, we should also realize that because the computation of the chromatic polynomial of a graph is NP-hard, there is no polynomial time algorithm that has a relative error less than 1 unless \( P = NP \) [Sto85]. Therefore, 1 is the best upper bound we can expect. We now give an example that has a relative error of 1. Consider the complete bipartite graph \( K_{m,2} \) shown in Figure 9.4.

According to Formula (9.7), we have the relative error

\[
\Delta \leq \frac{x^2(x-1)^{m-1} - x^2(x-2)^{m-1}}{x^2(x-1)^{m-1} + x^2(x-2)^{m-1}},
\]

which gives 1 when \( x = 2 \). In general, \( \hat{C}(K_{m,n}, x) = 0 \) for \( x \leq \min(m, n) \), while \( C(K_{m,n}, x) > 0 \) for \( x \geq 2 \).

Other reasons for choosing the harmonic mean are as follows. We generally have \( \hat{C}(G, x) > C(G, x) \) except for some small natural numbers \( x \), and the harmonic mean is always less than or equal to the arithmetic mean [All90]. Therefore, the harmonic mean
usually gives a better upper bound than the arithmetic mean. Moreover, for each graph $G$ of order $n$, both $U(G, x)$ and $L(G, x)$ are polynomial of degree $n$ with the coefficient of $x^n$ being 1. Hence, the degree of the numerator is smaller than the degree of the denominator in Formula (9.7). That implies $\lim_{x \to \infty} \Delta = 0$.

To consider the average performance behavior, we have run some experiments on randomly generated graphs. The edges in the graph are chosen independently and with probability 0.5. Figures 9.5(a) and (b) display, respectively, the distributions of the relative errors with respect to the number of colors over 5 graphs of order 8 and over 5 graphs of order 16. Figure 9.5(c) displays the average relative errors with respect to the number of colors over these graphs (0 for order 8 graphs and $X$ for order 16 graphs). The results show that the relative error $\Delta$ decreases as the number of colors $k$ increases except for transient fluctuation for small values of $k$, and the relative error increases as the order of graph increases. These phenomena coincide with Formula (9.7) obtained from performance analysis. The smallest-degree heuristic employed in the perfect-smallest-last ordering aims to restrict the transient fluctuation to very small values. This allows us to obtain a good approximation of the chromatic polynomial of a graph in most cases.

9.6 Summary

This chapter described a method for the inference of the relation-size complexity for a class of predicates that can be unfolded into a conjunction of binary equality and disequality constraints of the forms $x = y$ and $x \neq y$. The variables in the constraints are assumed to range over the same finite domain. Since the class of predicates is somewhat restricted, the method can compute both upper and lower bounds on the relation-size complexity.

The problem of computing the relation size of a conjunction of binary equality and disequality constraints was shown to be able to reduce to the problem of computing the chromatic polynomial of a graph. However, the problem of computing the chromatic polynomial of a graph is NP-hard. This chapter presented an approximation algorithm for estimating upper and lower bounds on the chromatic polynomial of a graph. Finally, this chapter also gave a performance analysis and a set of performance measurements for this algorithm.
Figure 9.5: Experimental measurements for Algorithm 9.1
CHAPTER 10

THE TIME COMPLEXITY ANALYZER

The main function of the time complexity analyzer is the inference of the time complexit
functions of predicates. A time complexity function $T_p : \mathcal{N}_1^\infty \rightarrow \mathcal{N}_1^\infty$ of a predicate $p$ with $n$ input positions is a function representing a worst-case upper bound on the time
taken to execute $p$ for a single input in terms of the size of the input positions of $p$.

We will first describe a method for the inference of a worst-case upper bound on the
time taken to execute a clause. We then discuss how a worst-case upper bound on the
time taken to execute a predicate can be derived from upper bounds on the time taken to
eexecute its clauses. Finally, we give an example and discuss the complexity of the time
complexity analyzer.

10.1 Clause Analysis

This section describes a method for the inference of a worst-case upper bound on the time
complexity of a clause. We assume that upper bounds on the input size of each body literal
and on the number of times each body literal is executed have already been computed by
the argument-size and number-of-solutions complexity analyzers as discussed in Chapters
5 and 6, respectively.

Let $C$ be a clause $L_0 : -L_1, \ldots, L_m$. We will use $t_i$ to denote an expression representing
an upper bound on the time taken to execute the literals $L_0, L_1, \ldots, L_i$ (with respect to
a single input to $C$) in terms of the size of the input positions in the head $L_0$. The time
$t_m$ taken to execute the entire clause with respect to a single input is also denoted by
t_C$. Recall that we also use $sz(\alpha a)$ to denote an upper bound on the size of the term
at an argument position $\alpha$, and $sol_i$ to denote an upper bound on the number of times
the literal $L_{i+1}$ is executed (with respect to a single input to $C$). The expression $sz(\alpha a)$
for each input position $\alpha$ in $L_0$ is indeed a function with domain $\Theta_C$ (namely, the set of
substitutions $\theta$ such that the input arguments of $\theta(L_0)$ are ground). Hence, the expressions
t_i and t_C are also considered to be a function with domain $\Theta_C$. 
An algorithm for the inference of an upper bound on the time taken to execute a clause is given in 10.1 and described as follows. Let $C$ be a clause $L_0:-L_1,\ldots,L_m$. The time taken to execute this clause is the time taken to unify the actual parameters with the formal parameters in the head $L_0$ together with the time taken to execute each of its body literals.

Let $\tau$ denote an expression representing an upper bound on the time taken to unify the formal parameters in the head with the actual parameters in the call being executed in terms of the size of the input positions in the head. Then the time expression $t_0$ is given as

$$t_0 = \tau.$$ 

Because literals can produce multiple solutions via backtracking, the time taken to execute a body literal $L_i$ is given by the product of $\text{sol}_{i-1}$ (namely, the number of times $L_i$ is executed) and the time complexity function of $L_i$ (namely, the time taken to execute $L_i$ once). Let $a_1,\ldots,a_n$ be the input positions of $L_i$. Then, we have

$$t_i = t_{i-1} + \text{sol}_{i-1} \times T_{L_i}(\text{sz}(a_1),\ldots,\text{sz}(a_n)),$$

for $1 \leq i \leq m$. Since the predicates of a program are processed in a topological order of the call graph as discussed in Section 4.4, we can assume that the time complexity functions for nonrecursive literals (namely, ones whose predicate symbols are not involved in a cycle in the call graph involving the predicate symbol of the clause head) have already been computed. On the other hand, we represent the time complexity functions for recursive literals in symbolic form, which are the functions we want to infer in the analysis.

There are a number of different metrics that can be used as the unit of time complexity, for example, the number of resolutions, the number of unifications, or the number of instructions executed. If the time complexity metric is the number of resolutions, then $\tau$ is 1; if it is the number of unifications, then $\tau$ is the arity of the clause head; if it is the number of instructions executed, then $\tau$ depends on the mode of the predicate, and the type and size of the arguments in the head [War83].

For simplicity, in what follows, we will assume that the time complexity metric used is the number of resolutions. The analyses for other time complexity metrics can be handled analogously. Therefore, the time complexity inferred for a goal is an upper bound on the
Algorithm 10.1

\( t_{\text{clause}}(C, Sz, Sol, N) \): Time complexity analysis of a clause.

**Input:** A clause \( C \): \( L_0: \overline{L}_1, \ldots, L_m \), the set \( Sz \) of size expressions for the input positions in each literal of \( C \), the set \( Sol \) of number-of-solutions expressions for the literals of \( C \), and a number \( N \) specifying the rightmost literal \( L_N \) to which the analysis will proceed.

**Output:** The time expression \( t_C \) for \( C \).

**Method:**

begin
  if \( N < 0 \) then return 0
  else
    let \( \tau \) be an upper bound on the time taken to unify \( L_0 \) with a call to \( C \);
    \( t_C := t_0 := \tau \);
    if \( N > m \) then \( F := m \) else \( F := N \) fi
    for \( i := 1 \) to \( F \) do
      let \( a_1, \ldots, a_n \) be the input positions of \( L_i \);
      if \( L_i \) is nonrecursive then
        \( t_i := sol_{i-1} \times T_{L_i}(sz(@a_1), \ldots, sz(@a_n)) \)
      else
        use \( T_{L_i}(sz(@a_1), \ldots, sz(@a_n)) \) as a symbolic expression;
        \( t_i := sol_{i-1} \times T_{L_i}(sz(@a_1), \ldots, sz(@a_n)) \)
      fi
      \( t_C := t_C + t_i \)
    od
    return \( t_C \)
  fi
end

Figure 10.1: An algorithm for time complexity analysis of a clause
number of vertices in the search tree corresponding to this goal. Let $G$ denote the set of goals in the language and $P$ a program. A time measure is a function $\| \cdot \|_P : G \to N$. For each goal $g$, $\|g\|_P$ gives the number of vertices in the search tree for $g$ with respect to $P$. Thus, for the following program $P$:

\[
p(1, u), \quad p(2, v), \quad p(2, w).
\]

$\|p(0, X)\|_P = 0$, $\|p(1, X)\|_P = 1$, and $\|p(2, X)\|_P = 2$. Hereafter, when the program under consideration is clear from the context, we will omit the subscript $P$ in $\| \cdot \|_P$.

Algorithm 10.1 is given in a form so that it can also be used to infer an upper bound on the time taken to execute part of a clause. The fourth input argument $N$ in the algorithm specifies the rightmost literal $L_N$ to which the inference will proceed. If $N < 0$, no item is executed and 0 is returned as the time complexity. If $N \geq m$, all the items in the clauses are executed and the time taken to execute the entire clause is returned.

**Proposition 10.1.1** Let $C$ be a clause $L_0 \leftarrow L_1, \ldots, L_m$, and $c_1, \ldots, c_n$ be the input positions in $L_0$. Suppose the size expression $sz(\sigma a)$ for each input position $a$ in $L_i$, $1 \leq i \leq m$, is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$, and $sz(\sigma a)(\theta) \geq |\sigma(\theta)|_m$, for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, i-1)$, and suppose the number-of-solutions expression $sol_i$ for each $L_i$, $1 \leq i < m$, is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$, and $sol_i(\theta) \geq |\theta(L_1, \ldots, L_i)|$ for every $\theta \in \Theta_C$. If $T_{L_j}$ is nondecreasing, for $1 \leq j \leq m$, and $t_i$ computed by Algorithm 10.1 is defined, for $0 \leq i \leq m$, then $t_i$ is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$, and $t_0(\theta) = 1 + |\theta(L_1, \ldots, L_i)|$, for every $\theta \in \Theta_C$.

**Proof** We prove by induction on the left-to-right order of the literals. For the base case, it is straightforward that $t_0 = 1$ is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$, and $t_0(\theta) = 1$, for every $\theta \in \Theta_C$.

Now, suppose $t_{i-1}$ is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$, and

\[
t_{i-1}(\theta) \geq 1 + |\theta(L_1, \ldots, L_{i-1})|,
\]

for every $\theta \in \Theta_C$. Let $a_1, \ldots, an$ be the input positions of $L_i$. Since $T_{L_i}$ is nondecreasing and $sz(\sigma a_1), \ldots, sz(\sigma an)$ are nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$,

\[
T_{L_i}(sz(\sigma a_1), \ldots, sz(\sigma an))
\]

is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$. Because $sol_{i-1}$ is nondecreasing with respect to $sz(\sigma c_1), \ldots, sz(\sigma c_n)$,
is nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$. From the hypothesis,

$$t_i = t_{i-1} + sol_{i-1} \times T_L_i(sz(a_1), \ldots, sz(a_n)),$$

is also nondecreasing with respect to $sz(c_1), \ldots, sz(c_n)$.

Since $T_L_i$ is nondecreasing, and because for every input position $a$ in $L_i$, $sz(a)(\theta) \geq |\sigma(a)|_{m_\theta}$, for every $\theta \in \Theta_C$ and every $\sigma \in \Theta_C(\theta, i-1)$, we have

$$T_L_i(sz(a_1)(\theta), \ldots, sz(a_n)(\theta)) \geq \max\{T_L_i(sz(c_1)(\theta), \ldots, sz(c_n)(\theta)) \mid \sigma \in \Theta_C(\theta, i-1)\}$$

for every $\theta \in \Theta_C$. Thus, we have

$$t_i(\theta) = t_{i-1}(\theta) + sol_{i-1}(\theta) \times T_L_i(sz(a_1)(\theta), \ldots, sz(a_n)(\theta)) \geq 1 + \|\theta(L_1, \ldots, L_{i-1})\| + sol_{i-1}(\theta) \times T_L_i(sz(a_1)(\theta), \ldots, sz(a_n)(\theta)) \geq 1 + \|\theta(L_1, \ldots, L_{i-1})\| + \|\sigma(L_i)\| \mid \sigma \in \Theta_C(\theta, i-1)\} \times \max\{\|\sigma(L_i)\| \mid \sigma \in \Theta_C(\theta, i-1)\} \geq 1 + \|\theta(L_1, \ldots, L_{i-1})\| + \sum_{\sigma \in \Theta_C(\theta, i-1)} \|\sigma(L_i)\| = 1 + \|\theta(L_1, \ldots, L_i)\|$$

for every $\theta \in \Theta_C$. □

### 10.2 Predicate Analysis

This section describes how a worst-case upper bound on the time taken to execute a predicate can be derived from upper bounds on the time taken to execute its clauses. In the argument-size and number-of-solutions complexity analyses, we are mainly concerned with successful computations, that is, the output size of successful computations or the number of successful computations. In the case of time complexity analysis, however, we also need to consider the time taken by failed computations.

An algorithm for time complexity analysis of a predicate is given in Figure 10.2 and described as follows. Recall that the clauses in a predicate are partitioned into mutually
Algorithm 10.2

t\_predicate(p, Sz, Sol): Time complexity analysis of a predicate.

*Input:* A predicate p, the set Sz of size expressions for the input positions in each literal of each clause of p, and the set Sol of number-of-solutions expressions for each literal of each clause of p.

*Output:* A time complexity function T_p for p.

*Method:*

```
begin
  for each cluster M_i E S do
    e_i := 0;
    for each clause C in p do
      let C be L_0 :- L_1, ..., L_m;
      let sz E Sz be the set of size expressions for positions in C;
      let sol E Sol be the set of number-of-solutions expressions for literals in C;
      if C E M_i then e_i := e_i + t\_clause(C, sz, sol, m)
      else
        let L_j be the leftmost literal such that L_0 :- L_1, ..., L_j
        and every clause in M_i are mutually exclusive;
        e_i := e_i + t\_clause(C, sz, sol, j - 1)
      fi
    od
  if p is nonrecursive then T_p := max_i\{e_i\}
  else
    d := max_i\{e_i | e_i is a difference equation\};
    compute the boundary conditions B due to implicit failure;
    D := d \cup B \cup \{e_i | e_i is not a difference equation\};
    T_p := solve_difference_equation(D) fi
  return T_p
end
```

Figure 10.2: An algorithm for time complexity analysis of a predicate
exclusive clusters as discussed in Section 4.5. For any call to the predicate, two clauses in distinct clusters cannot succeed at the same time. We first describe how the time complexity for a mutually exclusive cluster can be derived from the time complexity for each of the clauses in the predicate. The time complexity for each mutually exclusive cluster of clauses is the time taken to successfully execute the clauses in the cluster together with the time taken to unsuccessfully execute the clauses outside of the cluster. An upper bound on the time taken to successfully execute the clauses in the cluster can be obtained by summing the time complexity for each of its clauses. An upper bound on the time taken to unsuccessfully execute a clause outside of the cluster can be estimated by examining the sources that lead to mutual exclusion between that clause and the clauses in the cluster. The sources that lead to mutual exclusion between clauses can be detected by the analysis described in Section 4.5. Note that the time taken to execute a clause unsuccessfully is not a fixed quantity, but depends on the clauses that are executed successfully.

As an example, let us consider the following predicate calibrate/2, which calibrates a given score into one of four categories.

```prolog
:- mode(calibrate/2, [+,-]).
:- measure(calibrate/2, [nat, size]).
calibrate(Score, bad) :- Score < 70.
calibrate(Score, medium) :- 70 =< Score, Score < 80.
calibrate(Score, good) :- 80 =< Score, Score < 90.
calibrate(Score, excellent) :- Score >= 90.
```

It is clear that the four clauses are pairwise mutually exclusive because of mutual exclusion between the inequalities in their clause bodies. As a result, each of them constitutes a cluster by itself. However, for any call to the predicate, head resolution with the call can in fact succeed for all four clauses because the term Score appearing at the input position of the heads is a variable and it can unify with any input to the predicate. Therefore, if the given score in a call to the predicate is less than 70, say 65, then apart from the time taken to successfully execute the first clause, we also need to count the time taken to unsuccessfully execute the other clauses. In this case, the time taken for one head resolution and one inequality test is attributed to both the second and the third clauses since mutual exclusion between them and the first clause is caused by the second
inequality in their bodies. Similarly, the time taken for one head resolution is attributed
to the last clause because mutual exclusion between it and the first clause is caused by the
only inequality in its clause body. Consequently, if $B$ is the complexity of executing an
inequality test, then the time complexity for the cluster containing the first clause should
be counted as $4 + 3B$, instead of just $1 + B$. On the other hand, when considering the
time complexity for the cluster containing the last clause, only the time taken for one head
resolution is attributed to both the second and the third clauses since mutual exclusion
between them and the last clause is caused by the first inequality in their bodies.

Let $M$ be one of the mutually exclusive clusters of clauses of a predicate $p$. For each
clause $C$ in $p$ and $C \in M$, let $\tau_C$ be the time expression for the entire $C$ computed by
Algorithm 10.1. Then, the time taken to successfully execute the clauses in $M$ is bounded by

$$\sum_{C \in M} \tau_C.$$  

For each clause $C$ in $p$ and $C \not\in M$, let $C$ be defined as $L_0 \vdash L_1, \ldots, L_m$, and $a_{i,1}, \ldots, a_{i,n_i}$
be the input positions of $L_i$. Also, let $L_j$ be the leftmost literal such that $L_0 \vdash L_1, \ldots, L_j$
and every clause in $M$ are mutually exclusive. Then, the time $f_{C,M}$ taken to unsuccessfully
execute $C$ with respect to $M$ is given as

$$f_{C,M} = \begin{cases} 
0 & \text{if } j = 0 \\
\tau + \sum_{i=1}^{j-1} (\text{sol}_{i-1} \times T_{L_i}(\text{sz}(a_{i,1}), \ldots, \text{sz}(a_{i,n_i}))) & \text{if } j > 0.
\end{cases}$$  \hspace{1cm} (10.1)

The time taken to unsuccessfully execute the clauses outside of $M$ is thus bounded by

$$\sum_{C \not\in M} f_{C,M}.$$  

Put together, an upper bound expression $t_M$ on the time complexity of $M$ is given as

$$t_M = \sum_{C \in M} \tau_C + \sum_{C \not\in M} f_{C,M}. \hspace{1cm} (10.2)$$

Since the time complexity of a cluster $M$ of a predicate $p$ with input positions $a_1, \ldots, a_n$
usually depends on the input size of the calls to $p$, we can also symbolically represent the
time complexity of $M$ as $T_p(\text{sz}(a_1), \ldots, \text{sz}(a_n))$, namely,

$$T_p(\text{sz}(a_1), \ldots, \text{sz}(a_n)) = t_M.$$
This equation explicit represents the time complexity of a mutually exclusive cluster in terms of the size of the input positions in the predicate. If \( t_M \) does not contain any symbolic expression except for the size of the input positions in the predicate, then this equation is a closed form function. On the other hand, if, in addition to the size of the input positions in the predicate, \( t_M \) also contains symbolic expressions representing the time complexity for the recursive literals in the recursive clauses in \( M \), then this equation is in the form of difference equation.

We now describe how a time complexity function of a predicate can be derived from the upper bounds on the time complexity for the mutually exclusive clusters of the predicate. The analysis for nonrecursive predicates is straightforward. Since the complexity equations obtained from the clusters are already in closed form, we can take the maximum among them as a time complexity function for the predicate.

Let \( p \) be a nonrecursive predicate with input positions \( a_1, \ldots, a_n \). Also, let \( e_1, \ldots, e_k \) be the time expressions obtained from the mutually exclusive clusters of \( p \). Then, a time complexity function for \( p \) is given as

\[
T_p(sz(a_1), \ldots, sz(a_n)) = \max\{e_1, \ldots, e_k\}. \tag{10.3}
\]

The analysis for recursive predicates is more complicated. The complexity equation obtained from each mutually exclusive cluster may be a difference equation, or a closed form function serving as a boundary condition. Since a recursion may go through recursive clauses in several distinct mutually exclusive clusters, to achieve the worst-case analysis, we need to consider the maximum of the complexity functions obtained from distinct clusters that contain recursive clauses.

As an example, consider the following predicate \( \text{part/4} \) that partitions a list of numbers into two sublists based on the pivot number given at the first argument position:

\[
\begin{align*}
\text{mode(+,+,,-,-).} \\
\text{measure(part/4, [?,length,length,length]).} \\
\text{part(F, [], []).} \\
\text{part(F, [X|Y], [X|Y], Y2) :- X =< F, cost1, part(F, Y, Y1, Y2).} \\
\text{part(F, [X|Y], Y1, [X|Y2]) :- X > F, cost2, part(F, Y, Y1, Y2).}
\end{align*}
\]

where \( \text{cost1/0} \) is a predicate with time complexity 1 and \( \text{cost2/0} \) is a predicate with time complexity 2. The three clauses are pairwise mutually exclusive, and thus each of
them constitutes a cluster by itself. The time complexity equations computed for them are respectively

\[ T_{\text{part}}(0) = 1; \]
\[ T_{\text{part}}(n) = T_{\text{part}}(n - 1) + 3; \]
\[ T_{\text{part}}(n) = T_{\text{part}}(n - 1) + 4. \]

The time complexity equation for each cluster containing a recursive clause only bounds the time complexity of computations that recurse through that recursive clause. However, given a list of numbers, if some of them are larger than the pivot number and some of them are smaller than the pivot number, then the corresponding computation will recurse through both recursive clauses. Therefore, we need to take the maximum of the complexities for the two clusters containing recursive clauses and use this resulted difference equation to compute the complexity of the predicate. In this example, we should solve the following two equations:

\[ T_{\text{part}}(0) = 1, \]
\[ T_{\text{part}}(n) = \max\{T_{\text{part}}(n - 1) + 3, T_{\text{part}}(n - 1) + 4\} = T_{\text{part}}(n - 1) + 4. \]

and yield the result \( T_{\text{part}} = \lambda x. 4x + 1. \)

Let \( p \) be a recursive predicate with input positions \( a_1, \ldots, a_n \). Also, let \( e_1, \ldots, e_k \) be the time complexity expressions obtained from the mutually exclusive clusters that contain recursive clauses. Then, a time complexity function for \( p \) can be obtained by solving the following difference equation

\[ T_p(sz(a_1), \ldots, sz(a_n)) = \max\{e_1, \ldots, e_k\}. \quad (10.4) \]

together with the boundary conditions provided by the clusters that contain only nonrecursive clauses.

In addition to terminating through nonrecursive clauses, a recursion may also terminate via implicit failure as discussed in Section 6.3. For example, consider again the predicate \texttt{member/2} introduced in Section 6.3:

\begin{verbatim}
:- mode(member/2, [+]).
:- measure(member/2, [?length]).
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
\end{verbatim}
The time expressions for the two clauses would be inferred to be

\[ T_{\text{member}}(n - 1) + 1. \]

Since the two clauses are not mutually exclusive, they together constitute a single cluster. The complexity equation for the cluster would thus be inferred to be

\[ T_{\text{member}}(n) = T_{\text{member}}(n - 1) + 2. \]  \hspace{1cm} (10.5)

Notice that there is no base case from which this equation can be solved. As a result, it is necessary to explicitly add boundary conditions to account for this fact. Following the same reasoning as in Section 6.3, we can add the following boundary condition:

\[ T_{\text{member}}(0) = 0. \]  \hspace{1cm} (10.6)

Now Equations (10.5) and (10.6) can be solved to give the expected result \( \lambda x \cdot 2x \), namely, the number of resolution steps taken for a call to \text{member}/2 with an input of length \( x \) is at most \( 2x \).

**Theorem 10.2.1** Let \( p \) be a predicate, \( C_i : L_{i,0} : - L_{i,1}, \ldots, L_{i,m_i} \) be the \( i \)th clause of \( p \), and \( c_{i,1}, \ldots, c_{i,n} \) be the input positions in \( L_{i,0} \). For each \( C_i \), suppose the expression \( \text{sz}(\text{@}a) \), for each input position \( a \) in \( L_{i,j} \), \( 1 \leq j \leq m_i \), is nondecreasing with respect to \( \text{sz}(\text{@}c_{i,1}), \ldots, \text{sz}(\text{@}c_{i,n}) \), and \( \text{sz}(\text{@}a)(\theta) \geq |\sigma(\text{@}a)|_{m_a} \) for every \( \theta \in \Theta_{C_i} \) and every \( \sigma \in \Theta_{C_i}(\theta, j - 1) \), and suppose the expression \( \text{sol}_j \), for each \( L_{i,j} \), \( 1 \leq j < m_i \), is nondecreasing with respect to \( \text{sz}(\text{@}c_{i,1}), \ldots, \text{sz}(\text{@}c_{i,n}) \), and \( \text{sol}_j(\theta) \geq \|\theta(L_{i,1}, \ldots, L_{i,j})\| \) for every \( \theta \in \Theta_{C_i} \). If the function \( \text{solve\_difference\_equation} \) returns a nondecreasing upper bound solution for any given set of difference equations, and \( T_p \) computed by Algorithm 10.2 is defined, then \( T_p \) is a nondecreasing time complexity function for \( p \).

**Proof** Let \( p \) be a predicate in a program \( P \), and \( a_1, \ldots, a_n \) be the input positions of \( p \). We will prove that by processing the predicates in \( P \) in a topological order of the call graph, every defined complexity function computed by Algorithm 10.2 is a nondecreasing time complexity function. We show by induction on the topological order of the call graph.

We first consider three base cases in which there is no nonrecursive literals in the clauses of a predicate. First, consider a nonrecursive predicate \( p \) consisting of clauses with
empty body. For each mutually exclusive cluster $M_k$ of $p$, let $t_{Ci}$ denote the time expression for successfully executing a clause $Ci \in M_k$, and $f_{Ci,M_k}$ denote the time expression for unsuccessfully executing a clause $Ci \notin M_k$ with respect to $M_k$. From Proposition 10.1.1, both $t_{Ci}$ and $f_{Ci,M_k}$ are nondecreasing with respect to $sz(@c_1), \ldots, sz(@c_n)$, or equivalently, with respect to $sz(@a_1), \ldots, sz(@a_n)$. Since Equation (10.1) involves only summation, and summation is nondecreasing,

$$\sum_{Ci \in M_k} t_{Ci} + \sum_{Ci \notin M_k} f_{Ci,M_k}$$

is also nondecreasing with respect to $sz(@a_1), \ldots, sz(@a_n)$. Further, because Equation (10.3) involves only the maximum function, and the maximum function is nondecreasing, $T_p$ is also nondecreasing with respect to $sz(@a_1), \ldots, sz(@a_n)$.

Let $L_{i,j_i}$ be the leftmost literal in $Ci \notin M_k$ such that $L_{i,0} : - L_{i,1}, \ldots, L_{i,j_i}$ and every clause in $M_k$ are mutually exclusive. Also, let $step$ be the step function such that $step(x) = 0$, if $x \leq 0$, and $step(x) = 1$, otherwise. From Proposition 10.1.1,

$$t_{Ci}(\theta) \geq 1 + \|\theta(L_{i,1}, \ldots, L_{i,m_i})\|,$$

for every $Ci \in M_k$ and every $\theta \in \Theta_{Ci}$, and

$$f_{Ci,M_k}(\theta) \geq step(j_i) \times (1 + \|\theta(L_{i,1}, \ldots, L_{i,j_i-1})\|),$$

for every $Ci \notin M_k$ and every $\theta \in \Theta_{Ci}$. For each goal $g$ of $p$, let $t_{a_1}, \ldots, t_{a_n}$ be the input arguments of $p$, and $\theta_i = mgv(g,L_{i,0})$ for all $i$. Then, we have

$$T_p(t_{a_1|m_1}, \ldots, t_{a_n|m_n})$$

$$\geq \max_k \{ \sum_{Ci \in M_k} t_{Ci}(\theta_i) + \sum_{Ci \notin M_k} f_{Ci,M_k}(\theta_i) \}$$

$$\geq \max_k \{ \sum_{Ci \in M_k} (1 + \|\theta_i(L_{i,1}, \ldots, L_{i,m_i})\|) + \sum_{Ci \notin M_k} (step(j_i) \times (1 + \|\theta_i(L_{i,1}, \ldots, L_{i,j_i-1})\|)) \}$$

$$\geq \|g\|.$$

Therefore, $T_p$ is a nondecreasing time complexity function for $p$.

Second, consider a direct recursive predicate $p$ that involves only itself, namely, all the body literals in its clauses have the predicate symbol $p$. For each mutually exclusive cluster $M_k$ of $p$, let $t_{Ci}$ denote the time expression for successfully executing a clause $Ci \in M_k$, and $f_{Ci,M_k}$ denote the time expression for unsuccessfully executing a clause $Ci \notin M_k$ with respect to $M_k$. From Proposition 10.1.1,
If \( Ci \notin Mk \) with respect to \( Mk \). From Proposition 10.1.1, if the time complexity function for \( p \) is nondecreasing, then both \( t_{Ci} \) and \( f_{Ci,Mk} \) are nondecreasing with respect to \( sz(\$Ci,1), \ldots, sz(\$Ci,n) \), or equivalently, with respect to \( sz(\$a1), \ldots, sz(\$an) \). Since Equation (10.1) involves only summation, and summation is nondecreasing,

\[
\sum_{Ci \in Mk} t_{Ci} + \sum_{Ci \in Mk} f_{Ci,Mk}
\]

is also nondecreasing with respect to \( sz(\$a1), \ldots, sz(\$an) \). Further, because Equation (10.4) involves only the maximum function, the difference equation obtained is also nondecreasing with respect to \( sz(\$a1), \ldots, sz(\$an) \). Since the boundary conditions due to implicit failure are constant, they are nondecreasing. Thus, if the function \( solve\_difference\_equation \) returns a nondecreasing solution for any given set of difference equations, then \( Tp \) is also nondecreasing with respect to \( sz(\$a1), \ldots, sz(\$an) \).

Let \( Li,ji \) be the leftmost literal in \( Ci \notin Mk \) such that \( Li,0 := L_{i,1}, \ldots, L_{i,ji} \) and every clause in \( Mk \) are mutually exclusive. From Proposition 10.1.1, if the time complexity function for \( p \) is a nondecreasing upper bound function, then

\[
t_{Ci}(\theta) \geq 1 + \|\theta(L_{i,1}, \ldots, L_{i,mi})\|
\]

for every \( Ci \in Mk \) and every \( \theta \in \Theta_{Ci} \), and

\[
f_{Ci,Mk}(\theta) \geq \text{step}(ji) \times (1 + \|\theta(L_{i,1}, \ldots, L_{i,ji-1})\|)
\]

for every \( Ci \notin Mk \) and every \( \theta \in \Theta_{Ci} \). The boundary conditions due to implicit failure are computed from Equations (10.3) and (10.4) with respect to specific values; hence, they are just special cases of

\[
\max_k \{\sum_{Ci \in Mk} t_{Ci}(\theta_i) + \sum_{Ci \in Mk} f_{Ci,Mk}(\theta_i)\}
\]

Let \( D \) be the set of difference equations derived for \( p \). For each goal \( g \) of \( p \), let \( t_{a1}, \ldots, t_{an} \) be the input arguments of \( p \), and \( \theta_i = mgu(g, L_{i,0}) \) for all \( i \). Then, if the function \( solve\_difference\_equation \) returns an upper bound solution for any given set of difference equations, we have

\[
Tp([t_{a1}|m_{a1}, \ldots, |t_{an}|m_{an}) = difference\_equation\_solver(D)([t_{a1}|m_{a1}, \ldots, |t_{an}|m_{an})
\geq \max_k \{\sum_{Ci \in Mk} t_{Ci}(\theta_i) + \sum_{Ci \in Mk} f_{Ci,Mk}(\theta_i)\}
\]
\[
\geq \max_k \{ \sum_{c \in \mathcal{E}_M} (1 + \| \theta_i (L_{i,1}, \ldots, L_{i,m_i}) \|) + \sum_{c \in \mathcal{E}_M} (\text{step}(ji) \times (1 + \| \theta_i (L_{i,1}, \ldots, L_{i,j_i-1}) \|)) \}
\]

Therefore, \( T_p \) is a nondecreasing time complexity function for \( p \). This also verifies the assumptions for applying Proposition 10.1.1 in the proof.

Third, consider a set of indirect recursive predicates that involve only themselves. Following the analogous argument as for the second case, and using the function \textit{solve_difference_equation} to solve a system of simultaneous difference equations, we can derive the same result as the second case.

Since these three cases are the only base cases, after proving these base cases, Proposition 10.1.1 can be applied for the inductive cases as for the base cases. Consequently, the inductive cases can be proved in the analogous way as for the base cases. \( \square \)

### 10.3 An Example

Consider the list permutation program:

\[
:- \text{mode}(\text{perm}/2, [+,-]).
\]

\[
:- \text{measure}(\text{perm}/2, [\text{length}, \text{length}]).
\]

\[
\text{perm}([], []).
\]

\[
\text{perm}([X|Y], [R|Rs]) :- \text{select}(R, [X|Y], Z), \text{perm}(Z, Rs).
\]

\[
:- \text{mode}(\text{select}/3, [-,+,-]).
\]

\[
:- \text{measure}(\text{select}/3, [?, \text{length}, \text{length}]).
\]

\[
\text{select}(X, [X|Xs], Xs).
\]

\[
\text{select}(X, [Y|Ys], [Y|Zs]) :- \text{select}(X, Ys, Zs).
\]

Let \( \text{head}[] \) denote the size of the \( i \)th argument position in the head of a clause, \( \text{sol}_{i-1} \) denote the number of times the \( i \)th body literal in a clause is executed, and \( t_i \) denote the time taken to execute a clause up to and including the \( i \)th literal. The argument-size complexity analysis and the number-of-solutions complexity analysis of this programs are discussed in Sections 5.4 and 6.4.
We start with the analysis for the predicate `select/3`. We first consider the nonrecursive clause. The time complexity expression for the clause head is

\[ t_0 = 1. \]

We next consider the recursive clause. The time complexity expressions for the literals in this clause are

\[ t_0 = 1; \]
\[ t_1 = t_0 + sol_0 \times T_{select}(head[2] - 1) = T_{select}(head[2] - 1) + 1. \]

Since the two clauses constitute a mutually exclusive cluster, we take the sum of the two time complexity expressions to form the time complexity equation for the cluster:

\[ T_{select}(head[2]) = T_{select}(head[2] - 1) + 2. \]  \hspace{1cm} (10.7)

Because the difference equation is first order, the minimum input size the recursive literal may have is 0, and head resolution fails for both clauses when the input size is 0, we explicitly add the following boundary condition:

\[ T_{select}(0) = 0. \]  \hspace{1cm} (10.8)

Equations (10.7) and (10.8) can be solved to yield \( T_{select} \equiv \lambda x. 2x \), namely, the number of resolution steps taken by a call to `select/3` with an input of length \( x \) is at most \( 2x \).

We continue with the analysis for the predicate `perm/2`. We first consider the nonrecursive clause. The time complexity expression for the clause head is

\[ t_0 = 1. \]

We next consider the recursive clause. The time complexity expressions for the literals in this clause are

\[ t_0 = 1; \]
\[ t_1 = t_0 + sol_0 \times T_{select}(head[1]) = 2 \times head[1] + 1; \]
\[ t_2 = t_1 + sol_1 \times T_{perm}(head[1] - 1) \]
\[ = head[1] \times T_{perm}(head[1] - 1) + 2 \times head[1] + 1. \]

Since the two clauses are mutually exclusive, each of them constitutes a cluster by itself. For the cluster consisting of the first clause, since it and the head of the second clause
are mutually exclusive, the time taken to unsuccessfully execute the second clause is 0. Similarly, for the cluster consisting of the second clause, since it and the head of the first clause are mutually exclusive, the time taken to unsuccessfully execute the first clause is also 0. Therefore, we have the following two equations for the two clusters:

\[ T_{\text{perm}}(0) = 1, \]  
\[ (10.9) \]

and

\[ T_{\text{perm}}(\text{head}[1]) = \text{head}[1] \times T_{\text{perm}}(\text{head}[1] - 1) + 2 \times \text{head}[1] + 1. \]  
\[ (10.10) \]

Equations (10.9) and (10.10) can be solved to yield

\[ T_{\text{perm}} = \lambda x. \sum_{i=1}^{x} (3x!/i!) + 3x! - 2. \]  
\[ (10.11) \]

10.4 Complexity

This section discusses the complexity of Algorithms 10.1 and 10.2. We will use the parameters defined in Appendix B.

We first consider the complexity of Algorithm 10.1. The complexity of simplifying the time complexity function of a body literal is \( O(a \cdot E) \). To infer the time complexity expression \( t_c \), for a clause \( C \) with \( l \) body literals, we need \( l \) multiplications and additions. Altogether, the complexity of Algorithm 10.1 is \( O(l \cdot a \cdot E + l \cdot M + l \cdot A) \).

We now consider the complexity of Algorithm 10.2. Since the complexity for inferring the time taken to unsuccessfully execute a clause is bounded by the complexity for inferring the time taken to successfully execute a clause, it is also \( O(l \cdot a \cdot E + l \cdot M + l \cdot A) \). For each mutually exclusive cluster, we need to consider each clause in the predicate once and add the resulted expressions accordingly. Thus, the complexity for analyzing each mutually exclusive cluster is \( O(c \cdot l \cdot a \cdot E + c \cdot l \cdot M + c \cdot l \cdot A). \)

We now consider the complexity of analyzing a predicate. For a predicate with \( c \) clauses, there are at most \( c \) mutually exclusive clusters. Therefore, the complexity for analyzing all the clusters in the predicate is \( O(c^2 \cdot l \cdot a \cdot E + c^2 \cdot l \cdot M + c^2 \cdot l \cdot A). \) Equations (10.3) and (10.4) evaluate the maximum of two expressions at most \( c - 1 \) times. Thus, they can be computed in time \( O(c^2 \cdot l \cdot a \cdot E + c^2 \cdot l \cdot M + c^2 \cdot l \cdot A + c \cdot L). \) Because each set of difference equations can be solved in time \( D \), the complexity for the inference of a time complexity function of a predicate is \( O(c^2 \cdot l \cdot a \cdot E + c^2 \cdot l \cdot M + c^2 \cdot l \cdot A + c \cdot L + D). \)
In summary, the complexity of the time complexity analyzer is $O(p \cdot c^2 \cdot l \cdot a \cdot E + p \cdot c^3 \cdot l \cdot M + p \cdot c^2 \cdot l \cdot A + p \cdot c \cdot L + p \cdot D)$.

10.5 Summary

The main function of the time complexity analyzer is to use the argument-size complexity and the number-of-solutions complexity to infer a worst-case upper bound on the time complexity of predicates. We first described a method for the inference of a worst-case upper bound on the time complexity of a clause. We then discussed how a worst-case upper bound on the time complexity of a predicate can be derived from the worst-case upper bounds on the time complexity of its clauses.
CHAPTER 11
THE DIFFERENCE EQUATION SOLVER

The main function of the difference equation solver is to manipulate symbolic expressions and to solve difference equations. This chapter discusses the scope of symbolic manipulations and difference equation solving in the difference equation solver.

11.1 Manipulating Symbolic Expressions

Complexity functions of a predicate are generally functions in terms of the size of its input arguments. The size of each input argument is usually represented as a symbolic variable, and thus complexity functions are usually represented as closed form functions in terms of these symbolic variables. Moreover, in various complexity analyses for recursive clauses, the complexity functions for recursive literals are initially represented in symbolic form before they are solved. Therefore, to automate complexity analysis, we need the capability of manipulating a great deal of symbolic expressions. In particular, we need a systematic way to simplify symbolic expressions so that they can be stored, manipulated and displayed in a concise form.

Each complexity expression in CASLOG is represented in a normal form. Each normal form expression is represented as the sum of two subexpressions: a recurrent expression that involves symbolic complexity functions representing the complexity of user-defined predicates; and a nonrecurrent expression that involves only system-defined operators, such as addition, subtraction, multiplication, division, logarithm, exponentiation, factorial, and so on.

Each nonrecurrent expression is represented as a summation of a list of nonrecurrent terms. Each nonrecurrent term is represented as a product of a list of nonrecurrent factors. Each nonrecurrent factor can be a number, a symbolic variable (denoting an input size), the logarithm of, the exponentiation of, or the factorial of nonrecurrent expressions.

Each recurrent expression is represented in an analogous manner. It is represented as a summation of a list of recurrent terms. Each recurrent term is represented as a product...
of a list of recurrent factors. Each recurrent factor can be a nonrecurrent expression (representing the coefficient of the recurrent term), a symbolic complexity function (denoting the complexity of a user-defined predicate), the logarithm of, the exponentiation of, or the factorial of recurrent expressions.

As an example, the following expression

\[ x \times T_{perm}(x - 1) + 2 \times x + 1 \]

is represented in CASLOG as

\[
[[[T_{perm}([[]], [[x], [-1]])), [[x]]), [[x, 2], [1]]].
\]

CASLOG currently supports the following operations on normal form expressions: addition, subtraction, multiplication, logarithm, exponentiation, and factorial. For more thorough account on simplification of symbolic expressions, the reader is referred to the work by Caviness [Cav70] and Moses [Mos71].

11.2 Solving Difference Equations

Complexity functions for recursive predicates are in the form of difference equations. To automate complexity analysis, it is necessary to solve these difference equations into closed-form functions. The automatic solution of arbitrary difference equations is a difficult problem, but reasonable results can be achieved if difference equations are restricted to some common forms [CK77, Ivi78, Pet91]. In our case, however, due to the applications in our minds, complexity analysis is expected to be part of a compiler; therefore, there is an additional requirement that the solution of such equations be efficient. This forces us to sacrifice precision in some cases. Although automatic algebra systems can solve a wide class of difference equations, a table-driven method is chosen for the sake of efficiency and flexibility.

Since we are computing upper bounds on complexity, it suffices to compute an upper bound on the solution of a set of difference equations, rather than an exact solution. This can be done by simplifying the equations using transformations such that a solution to the transformed equations is guaranteed to be an upper bound on the solution to the original equations. In addition, as discussed in the previous chapters, we also want the upper bound solution to be a nondecreasing function. A set \( D \) of difference equations is
said to be nondecreasing-composed if every nonrecurrent term in $D$ is nondecreasing and the coefficient of every recurrent term in $D$ is nondecreasing. As shown in the proofs of Theorems 5.3.1, 6.3.1 and 10.3.1, the difference equations derived in the various complexity analyses are nondecreasing-composed. A class of difference equations is said to be relatively nondecreasing if the solution to every nondecreasing-composed instance of this class is a nondecreasing function. Fortunately, most difference equations encountered in practice are relatively nondecreasing, for example, the class of $n$-order linear difference equations, and the class of divide-and-conquer difference equations.

Our approach to achieving this is as follows. We provide a "library" of relatively nondecreasing difference equation schemas that have known solutions, a solution function $\text{soln}(s, d)$ that returns the known solution to an instance $d$ of a library schema, and a family of transformation rules that transform, if possible, a nondecreasing-composed set of difference equations into an instance of a library schema such that the new set of equations is still nondecreasing-composed and a solution to it is an upper bound on the solution to the original equations. A transformation rule is said to be sound if it satisfies the above nondecreasing upper bound preserving property. For each set of difference equations, we first try to match it with the difference equation schemas in the library. If there is a library schema that matches the equations, then the known solution to the schema is returned as the solution to the equations. If the equations cannot match any of the library schemas, then we try to transform the equations using the transformation rules. If such a transformation is possible, then the known solution to the transformed library schema is returned as the solution to the equations. Otherwise, the solution $\lambda x. \infty$ is returned as a conservative solution to the equations. An algorithm for computing a nondecreasing upper bound solution to a set of difference equations is given in Figure 11.1. This approach provides the flexibility of adding new library schemas and transformation rules to the system in the future.

The following example may make the idea clear. Consider the predicate $\text{fib}/2$, which computes the Fibonacci sequence, defined as
Algorithm 11.1

solve_difference_equation(D): Solving a set of difference equations.

Input: A set $D$ of difference equations.

Output: A nondecreasing upper bound solution to $D$.

Method: Suppose the system contains a library $S$ of relatively nondecreasing difference equation schemas, a solution function $\text{soln}$ over the schemas, and a family $T$ of sound transformation rules.

begin
    if $D$ matches some difference equation schema $s_1 \in S$ then
        return $\text{soln}(s_1, D)$
    else if $D$ can be transformed into a set of difference equation $E$ via a sequence of rules $t \in T$, and $E$ matches some difference equation schema $s_2$ then
        return $\text{soln}(s_2, E)$
    else
        return $\lambda x. \infty$
    fi
end

Figure 11.1: An algorithm for computing a nondecreasing upper bound solution to a set of difference equations

:- mode(fib/2, [+,-]).
:- measure(fib/2, [nat,nat]).

fib(0, 0).

fib(1, 1).

fib(M, N) :- M > 1,
              M1 is M-1, fib(M1, N1),
              M2 is M-2, fib(M2, N2),
              N is N1 + N2.

Using the number of resolutions as the time complexity metric, the difference equations
obtained from time complexity analysis of this predicate are

\[ T_{\text{fib}}(0) = 2, \]
\[ T_{\text{fib}}(1) = 2, \]
\[ T_{\text{fib}}(n) = T_{\text{fib}}(n - 1) + T_{\text{fib}}(n - 2) + 5, \quad n > 1. \]

Suppose that the library contains the schema \( s_1 \) (CASLOG indeed contains this schema):

\[
\begin{align*}
    f(I) &= D, \\
    f(I + 1) &= E, \\
    f(n) &= Af(n - 1) + Bf(n - 2) + C, \quad n > I + 1.
\end{align*}
\]

The general closed form solution to \( s_1 \) can be represented as

\[
f(n) = a\alpha^n + b\beta^n + c,
\]

where

\[
\alpha = (A + \sqrt{A^2 + 4B})/2,
\]
\[
\beta = (A - \sqrt{A^2 + 4B})/2.
\]

If neither \( \alpha \) nor \( \beta \) are equal to 1, then

\[
c = C/(1 - A - B).
\]

Further, if \( \alpha \) is not equal to \( \beta \), then

\[
\begin{align*}
    a &= ((D - c)\beta^{l+1} - (E - c)\beta^l)/(\alpha^l\beta^{l+1} - \alpha^{l+1}\beta^l), \\
    b &= (D - c - a\alpha^l)/\beta^l.
\end{align*}
\]

Given this general solution, we can substitute for the constants \( A, B, C, D, \) and \( E \) in the difference equations to yield

\[
\lambda x. 5.065 \times 1.618^x + 1.934 \times (-0.618)^x - 5
\]

as the closed form solution to the difference equations.

On the other hand, suppose that the library does not contain the schema \( s_1 \), but contains the schema \( s_2 \) (CASLOG indeed contains this schema):

\[
\begin{align*}
    f(I) &= C, \\
    f(n) &= Af(n - 1) + B, \quad n > I.
\end{align*}
\]
The general closed form solution to \( s_2 \) can be represented as

\[
f(n) = CA^n + \sum_{i=0}^{n-2} BA^i.
\]

Suppose there is a sequence of computation rules in the system that can transform the set of difference equations that matches the schema \( s_1 \) into a set of difference equations that matches the schema \( s_2 \). In this example, since the function \( T_{\text{fib}} \) is nondecreasing, the solution to the original equations is bounded above by the solution to the following simplified equations

\[
\begin{align*}
T_{\text{fib}}(0) &= 2, \\
T_{\text{fib}}(n) &= 2T_{\text{fib}}(n-1) + 5, \quad n > 0.
\end{align*}
\]

Using the general solution to \( s_2 \), we yield

\[
\lambda x \cdot 7 \times 2^x - 5
\]

as the closed form solution to the original difference equations. Finally, suppose both \( s_1 \) and \( s_2 \) are not included in the library, and the set of difference equations cannot be transformed to match any of the library schemas. In this case, the system will return \( \lambda x \cdot \infty \) as a conservative solution.

We now discuss the kinds of difference equations that have known solutions [CK77, Ivi78, Pet91]. In general, linear difference equations with constant coefficients can be solved using the methods of characteristic equations or ordinary generating functions. Using exponential generating functions, the solution of linear difference equations with polynomial coefficients can be reduced to that of ordinary differential equations. Moreover, for first order linear difference equations, there is a simple explicit closed form solution that depends on closed form solutions of sums and products. As a consequence, there is a large class of linear difference equations that have known solutions. For linear difference equations, our difference equation solver currently can solve first order difference equations, second order difference equations with constant coefficients, and divide-and-conquer difference equations.

Nonlinear difference equations may arise in the analysis of argument-size complexity, where the output of a recursive literal may be the input of another literal whose argument-size complexity function is nonlinear on its input. They may also arise in the analysis of
number-of-solutions complexity, where multiplication is applied to form the number-of-solutions complexity function of a clause and there may be two or more recursive literals in the clause body. The solution of nonlinear difference equations is generally much more difficult than the solution of linear difference equations. There is, however, a large class of nonlinear difference equations that can be transformed into linear equations by transformation of variables. For example, by taking the logarithm of both sides of an equation, products can be transformed into sums. Consider predicate perm/2.

\[
\text{:- mode}(\text{perm}/2, [+,-]).
\]
\[
\text{:- measure}(\text{perm}/2, [\text{length}, \text{length}]).
\]
\[
\text{perm}([], []).
\]
\[
\text{perm}([X|Y], [R|Rs]) :- \text{select}(R, [X|Y], Z), \text{perm}(Z, Rs).
\]

In Section 6.4, we obtain the equations for the number of solutions generated by its two clauses as follows:

\[
\text{Fr}_{\text{perm}}(n) = n \times \text{Fr}_{\text{perm}}(n - 1),
\]
\[
\text{Fr}_{\text{perm}}(0) = 1.
\]

By taking logarithms of both sides of the equations, we get

\[
\log(\text{Fr}_{\text{perm}}(n)) = \log(n) + \log(\text{Fr}_{\text{perm}}(n - 1)),
\]
\[
\log(\text{Fr}_{\text{perm}}(0)) = 0.
\]

Writing \( F(n) \) for \( \log(\text{Fr}_{\text{perm}}(n)) \), these equations can be written as

\[
F(n) = \log(n) + F(n - 1),
\]
\[
F(0) = 0.
\]

These equations can be solved to get the closed form solution:

\[
F(n) = \sum_{i=1}^{n} \log(i).
\]

To get a solution to the original equations, we exponentiate both sides to obtain

\[
\text{Fr}_{\text{perm}}(n) = \prod_{i=1}^{n} i = n!
\]
or $\lambda x. x!$, which is the desired solution for predicate `perm/2`.

Difference equations involving `max` and `min`—which occur frequently when analyzing logic programs—are also considered to be nonlinear, and there is no general method for solving them. However, since we are interested in computing upper bounds, such equations can be simplified to eliminate occurrences of `max` and `min` such that solutions to the resulting equation will be an upper bound on solutions to the original equation. The essential idea here is the following: in an expression $\text{max}(c_1, c_2)$, if one of the (non-negative) expressions is provably an upper bound on the other for all assignments of values to the variables occurring in them, then this expression is clearly the maximum; otherwise, the maximum is bounded above by the sum $c_1 + c_2$. The situation is somewhat simpler for `min`: if neither expression is a provable lower bound on the other, then either of the two expressions can be chosen as a conservative upper bound on the minimum. There are many possible ways to generalize this basic approach to more than two expressions: the main concern is the tradeoff between the precision and efficiency of the computation.

In addition to the nonlinear difference equations discussed above, there are several special form nonlinear difference equations that have known solutions [GK90].

Indirect recursive predicates result in a system of simultaneous difference equations of more than one function. It is possible in principle to reduce a system of difference equations of more than one function to a single difference equation of one function [LL59]. Consider the system of difference equations of two functions $f$ and $g$:

$$
3nf(n) = g(n) - g(n + 1) + 1,
$$
$$
f^2(n) = g(n) + g(n + 1) - 1.
$$

Substituting $n + 1$ for $n$, we get two more equations

$$
3(n + 1)f(n + 1) = g(n + 1) - g(n + 2) + 1,
$$
$$
f^2(n + 1) = g(n + 1) + g(n + 2) - 1.
$$

Eliminating the term $g(n)$ in the first two equations and $g(n + 2)$ in the last two equations, we get

$$
2g(n + 1) = f^2(n + 1) - 3nf(n) + 2,
$$
$$
2g(n + 1) = f^2(n + 1) + 3(n + 1)f(n + 1).
$$

Finally, eliminating the term $g(n + 1)$ we get
\[ f^2(n+1) + 3(n+1)f(n+1) = f^2(n) - 3nf(n) + 2, \]
a difference equation of a single function \( f \). Therefore, the approach described above is applicable to both direct and indirect recursive clauses.

The following theorem shows the soundness of the difference equation solver.

**Theorem 11.2.1** Given a nondecreasing-composed set \( D \) of difference equations, if all the difference equation schemas are relatively nondecreasing and all the transformation rules are sound, then the solution computed by Algorithm 11.1 is a nondecreasing upper bound on the solution to \( D \).

**Proof** If \( D \) matches a library schema \( s_1 \), then \( \text{soln}(s_1, D) \) is a nondecreasing upper bound solution to \( D \). If \( D \) can be transformed into a set \( E \) of difference equations by a sequence of transformation rules \( r \) such that \( E \) matches a library schema \( s_2 \), then since the computation rules \( r \) are sound, \( \text{soln}(s_2, E) \) is a nondecreasing upper bound solution to \( E \). Further, because the computation rules \( r \) are sound, \( \text{soln}(s_2, E) \) is also a nondecreasing upper bound solution to \( D \). Finally, if there is no transformation rule that can transform \( D \) into a set \( E \) of difference equations such that \( E \) can match some of the library schemas, then \( \lambda z. \infty \) is a nondecreasing upper bound solution to \( D \). \( \square \)

It is, unfortunately, rather difficult to syntactically characterize the classes of programs that can be analyzed by our approach. The reason is that such a characterization basically boils down to characterizing programs that give rise to difference equations of a certain kind, namely, linear difference equations with constant or polynomial coefficients. Now the exact form of the difference equations that are obtained for a predicate depend on the size measures under consideration. Because of this, it is difficult to give an abstract characterization of programs that are analyzable without a careful consideration of the particular size measures involved, and this can become a rather lengthy discussion. To make matters worse, even nonlinear equations can sometimes be transformed into linear equations that can be solved and the solutions transformed back solutions for the original equation. So a discussion of what programs can be analyzed would have to get into these kinds of transformations as well. The details can get quite messy, and are beyond the scope of this dissertation.
11.3 Summary

This chapter discussed the scope of symbolic manipulations and difference equation solving in the difference equation solver. Operations supported for manipulating symbolic expressions include addition, subtraction, multiplication, division, logarithm, exponentiation, and factorial. The classes of difference equations that can be handled by the system include first order linear difference equations, second order linear difference equations with constant coefficients, divide-and-conquer difference equations, nonlinear difference equations that can be transformed into linear equations by transformation of taking logarithm, and nonlinear difference equations involving max and min.
CHAPTER 12
APPLICATION I: CONTROLLING TASK
GRANULARITY IN PARALLEL SYSTEMS

12.1 Introduction

Logic programming languages offer a great deal of scope for parallelism. There are two principal flavors of parallelism in logic programs: AND-parallelism, where subgoals necessary to solve a goal are executed in parallel; and OR-parallelism, where different alternative execution branches are explored concurrently. By employing both kinds of parallelism, it is possible to extract “maximal” parallelism for a program [Kal87].

This is interesting in the abstract. However, just because something can be done in parallel does not necessarily mean, in practice, that it should be done in parallel. This is because the parallel execution of a task incurs various overheads, for example, overheads associated with process creation and scheduling, the possible migration of tasks to remote processors, the associated communication overheads, and so on. Thus, given the clause

\[
\text{part}(F, [X|Y], Y_1, [X|Y_2]) :- X > F, \text{part}(F, Y, Y_1, Y_2).
\]

the test ‘\(X > F\)’ can be executed in parallel with the recursive call. However, this test can typically be carried out in one or two machine instructions, and if the overhead associated with spawning the test as a separate task is more than a few instructions, the parallel execution of this goal may not be cost-effective. In general, a goal should not be a candidate for parallel execution if its granularity, namely, the “work available” underneath it, is less than the work necessary to create a separate task for that goal. This makes it desirable to devise a method whereby the granularity of a goal may be estimated at runtime. In order to be useful, the runtime overhead involved in such a method should be small, that is, as much work should be done at compile time as possible.

This chapter describes a method for estimating the granularity of goals. Most of the work in our approach is done at compile time. However, the work done by a call
to a recursive predicate typically depends on the size of its input, and hence cannot be estimated in any reasonable way at compile time—for such goals, some runtime work is necessary to determine the cost of any particular call to a recursive predicate. However, the cost incurred in such runtime computations is generally quite small.

Since compilers are allowed only to perform optimizations that can be guaranteed to not affect a program's runtime behavior adversely, and because interesting program properties are generally undecidable, compile-time analyses are usually expected to satisfy correctness criteria that state that information inferred during the analysis of a program is a sound, possibly conservative, estimate of the program's runtime behavior. Curiously, correctness criteria are not immediately obvious in the context of granularity analysis, since a mistake in granularity analysis can result in loss of performance but appears unlikely to change the semantics of the program. Despite this, it is desirable to be able to state what kind of invariant (with respect to runtime behavior) is satisfied by a granularity analysis algorithm, in order to allow us to reason formally about the behavior of programs that utilize the information inferred by it. Not surprisingly, the problem of determining precisely how much work will be done by a call is statically undecidable. This means that compile-time granularity analysis will be a conservative estimate of the amount of work performed at runtime. As such, it can give either a lower bound or an upper bound on the amount of runtime computation.

The analysis considered in this chapter gives granularity estimates that are an upper bound on the amount of work that may be done at runtime. There are a number of reasons for this. An important philosophical reason for this choice is the following: if a lower-bound analysis is conservative, it determines there is less work available than there is in practice, resulting in a loss of parallelism; this is conceptually akin to parallelizing sequential language programs, where actions are performed sequentially unless specified otherwise. If an upper-bound analysis is conservative, however, tasks are executed concurrently even though there may not be enough work available to justify this; this corresponds, conceptually, to "sequentializing" a parallel language, where actions are performed in parallel unless specified otherwise. Because the language models we have in mind resemble the latter rather than the former, upper-bound analyses appear to be more appropriate for our purposes. There are also important practical advantages to choosing upper-bound analyses, since it is difficult to give nontrivial lower bounds in most cases (very often,
for example, the case where head unification fails leads to a lower bound estimate of 0, which is not very useful), and also because many important simplifications can be performed if we are required to guarantee only an upper bound. This results in significant simplifications to our algorithms, with concomitant improvements in both compile-time and runtime overhead; in particular, unlike a lower-bound analysis, termination issues do not have to be considered separately.

12.2 Controlling Task Granularity

Granularity analysis for a set of nonrecursive predicates is relatively straightforward. Recursion is somewhat more problematic: the amount of work done by a recursive call depends on the depth of recursion, which in turn depends on the input. Reasonable estimates for the granularity of recursive predicates can thus be made only with some knowledge of the input. Our technique for dealing with this problem is to do as much of the analysis at compile time as possible, but postpone the actual computation of granularity until runtime. A fundamental criterion in our approach is that the runtime overhead incurred in this computation should be small. Given a recursive predicate \( p \), therefore, we compute an expression \( T_p(n) \) that satisfies the following criteria:

1. \( T_p(n) \) is relatively easy to evaluate; and

2. \( \text{Cost}_p(n) \leq T_p(n) \) for all \( n \), where \( \text{Cost}_p(n) \) denotes the cost of computing \( p \) for an input of size \( n \).

The idea is that \( T_p(n) \) is determined at compile time; it is evaluated at runtime, when the size of the input is known, and yields an estimate of the granularity of the predicate.\(^1\) For example, given a predicate defined by

\[
p([ ]).
p([H|L]) :- q(H), p(L).
\]

assume that the literals \( q(H) \) and \( p(L) \) in the body of the second clause can be shown to be independent by examining the literal dependency graph of the clause, so that these

---

\(^1\)In practice, we might prefer to not have to traverse the entire input at runtime to determine its size. This problem, which can be handled by maintaining some additional information, is somewhat orthogonal to the topic of this chapter, and is not pursued further here.
literals are candidates for concurrent execution. Suppose the expression \( T_q(n) \) giving the cost of \( q \) on an input of size \( n \) is \( 3n^2 \), and suppose the cost of creating a concurrent task is 48 units of computation. Then, the code generated for the second clause might be of the form

\[
n := \text{size}(H);
\]

\[
\text{if } 3n^2 < 48 \text{ then execute } q \text{ and } p \text{ sequentially as a single task}
\]

\[
\text{else execute } q \text{ and } p \text{ concurrently as separate tasks}
\]

Of course, this could be simplified further at compile time, so that the code actually executed at runtime might be of the form

\[
\text{if } \text{size}(H) < 4 \text{ then execute } q \text{ and } p \text{ sequentially as a single task}
\]

\[
\text{else execute } q \text{ and } p \text{ concurrently as separate tasks}
\]

Therefore, from a compilation point of view, we may want to obtain a "threshold input size" for predicates instead of the actual solutions to their cost equations. The idea is that if the cost of a predicate for an input of size \( n \) is given by \( f(n) \), and the task management overhead on the system under consideration is \( W \), then we wish to obtain a value \( K \) such that \( n > K \) if and only if \( f(n) > W \). From this, we can generate code that conditionally executes tasks in parallel depending on the size of the input. We realize this idea by extending the difference equation solver of CASLOG as follows. We associate with each solution \( f \) to a difference equation schema a function \( g \) such that \( g(W) = K \), where \( W \) is the task management overhead and \( K \) the threshold input size. The value of \( g(x) \) is defined as the least \( y \) such that \( f(y) > x \): since \( f \) is known ahead of time, \( g \) can also be computed ahead of time. Thus, once the task management overhead \( W \) has been determined for a system, the input threshold for each equation known to the difference equation solver can be determined statically.

If the initial set of difference equations \( d \) cannot match or be transformed to match any of the difference equation schemas, then the solution to \( d \) is returned as \( \lambda x. \infty \), the function that does an infinite amount of work for any size of input (in particular, equations for predicates with nonterminating execution branches do not have solutions). The practical effect of this is that if the system is unable to find a solution to the difference equations for a predicate \( p \), then calls to \( p \) are always executed in parallel. This is consistent
with our philosophy of "sequentializing a parallel language," where tasks are executed in parallel unless it can be proven that it is better to execute them sequentially. Note that since predicates with nonterminating execution branches are always executed in parallel, termination properties of programs are unaffected by such sequentialization.

12.3 Experimental Results

We have run a series of experiments in granularity control using two existing parallel logic programming systems: ROLOG and &-Prolog. ROLOG is a pure logic programming system based on Kale's reduce-or model [Kal85, RK89]; programs are annotated for parallelism by the user. &-Prolog is a parallel Prolog system based on strict- and non-strict independence [HR89] which uses a modified RAP-WAM abstract machine [Her86], and where annotations for parallelism can be automatic or user-provided. In the spirit of the "sequentializing parallel programs" philosophy pointed out in the introduction, in both cases granularity control was added to the parallelized programs and speedup measurements performed while running on a Sequent Symmetry multiprocessor.

The benchmark programs include a constraints consistency problem; the Fibonacci sequence; the Hanoi game; a quick-sort problem; a LR(1) set generation problem; a double sum problem; a simplified FFT problem; a list flattening problem; a matrix multiplication problem; a merge-sort problem; a polygon inclusion problem; and a tree traversal problem.

Figures 12.1 and 12.2 show plots of the average execution time against grain size over 20 runs for three benchmark programs, where the granularities in the x-axis are the threshold input sizes used for controlling task spawning. The larger the threshold input size, the smaller the degree of parallelism. We can draw two broad inferences from these figures. First, the U shape of these curves of execution time vs. grain size does verify that significant speedups can be achieved by proper use of granularity information. Second, based on the fairly wide "trough" in these curves, it is not essential to be absolutely precise in inferring the best grain size for a problem: there is a reasonable amount of leeway in how precise this information has to be; this suggests that granularity inference can usefully be performed automatically by a compiler.

Tables 12.1 and 12.2 show execution times for our benchmark programs compiled with no information of task granularity, compared with the case when they are compiled using grain size information inferred by our algorithm. It can be seen from these tables that
ROLOG on 4 processors (Symmetry)

(a) Benchmark fib(15)

(b) Benchmark hanoi(6)

(c) Benchmark quick-sort(75)

Figure 12.1: Execution time vs. task granularity on ROLOG
&-Prolog on 4 processors (Symmetry)

(a) Benchmark fib(15)

(b) Benchmark hanoi(6)

(c) Benchmark quick-sort(75)

Figure 12.2: Execution time vs. task granularity on &-Prolog
### ROLOG on 4 processors (Symmetry)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>T₀</th>
<th>T₁</th>
<th>T₀/T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>consistency(500)</td>
<td>820</td>
<td>560</td>
<td>1.46</td>
</tr>
<tr>
<td>fib(15)</td>
<td>1170</td>
<td>850</td>
<td>1.38</td>
</tr>
<tr>
<td>hanoi(6)</td>
<td>270</td>
<td>240</td>
<td>1.13</td>
</tr>
<tr>
<td>quick-sort(75)</td>
<td>600</td>
<td>580</td>
<td>1.03</td>
</tr>
<tr>
<td>LR(1)-set(3)</td>
<td>1264</td>
<td>1241</td>
<td>1.02</td>
</tr>
<tr>
<td>double-sum(2048)</td>
<td>2660</td>
<td>2259</td>
<td>1.18</td>
</tr>
<tr>
<td>fft(256)</td>
<td>2760</td>
<td>2636</td>
<td>1.05</td>
</tr>
<tr>
<td>flatten(536)</td>
<td>1161</td>
<td>1387</td>
<td>0.84</td>
</tr>
<tr>
<td>matrix-multi(8)</td>
<td>575</td>
<td>250</td>
<td>2.30</td>
</tr>
<tr>
<td>merge-sort(128)</td>
<td>2226</td>
<td>1912</td>
<td>1.16</td>
</tr>
<tr>
<td>poly-inclusion(30)</td>
<td>8979</td>
<td>5537</td>
<td>1.62</td>
</tr>
<tr>
<td>tree-traversal(8)</td>
<td>1890</td>
<td>1832</td>
<td>1.03</td>
</tr>
</tbody>
</table>

T₀: execution time with no granularity control.
T₁: execution time with granularity control.

Table 12.1: Execution times for benchmarks on ROLOG (times in milliseconds)

### &-Prolog on 4 processors (Symmetry)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>T₀</th>
<th>T₁</th>
<th>T₀/T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>consistency(500)</td>
<td>139</td>
<td>139</td>
<td>1.00</td>
</tr>
<tr>
<td>fib(15)</td>
<td>277</td>
<td>196</td>
<td>1.41</td>
</tr>
<tr>
<td>hanoi(6)</td>
<td>18</td>
<td>21</td>
<td>0.86</td>
</tr>
<tr>
<td>quick-sort(75)</td>
<td>111</td>
<td>93</td>
<td>1.19</td>
</tr>
</tbody>
</table>

T₀: execution time with no granularity control.
T₁: execution time with granularity control.

Table 12.2: Execution times for benchmarks on &-Prolog (times in milliseconds)
the runtime overhead incurred by our approach is small, and that granularity analysis can thus yield good speedups. It works better if the task management overhead is relatively high, as in ROLOG, or in systems that involve non-shared memory architectures. If the task creation and management overhead is sufficiently small, it may better not to use granularity control at all (this happens in some cases for &-Prolog). However, such situations can be determined ahead of time, simply by considering the overhead associated with task creation and management.

The runtime overhead incurred by our approach arises from two factors: the maintenance of size information and the grain size tests. It is observed that many predicates can be classified as either parallel predicates or sequential predicates at compile time, so no grain size control is needed for them, and thus no runtime overhead is associated with them. In this case, programs gain large speedups, for example, in our benchmarks matrix multiplication and polygon inclusion. Furthermore, the runtime overhead for predicates that need grain size control can be significantly reduced by loop unrolling; that is, a grain size test is performed once every $k$ iterations, where $k$ is the number of times the loop has been unrolled, instead of once every iteration.

Sometimes the runtime overhead does swamp the gains from sequentializing small parallel tasks, for example, in our benchmark flatten. At this time our compiler does not take the runtime overhead associated with granularity control into account in deriving the cost functions and inferring the threshold input size, so we get negative result in benchmark flatten. However, we do get positive results in most benchmarks and we shall incorporate the consideration for runtime overhead into our compiler for more precise estimates.

12.4 Related Work

Hudak and Goldberg consider program partitioning with reasonable grain sizes in functional programs [HG85], but do not discuss the compile time analysis necessary to estimate the amount of work that may be done by a call.

The problem of program partitioning has also been considered, in the context of functional programs, by Sarkar, who bases his algorithm on information obtained via runtime profiling rather than compile-time analysis [Sar89].

Rabhi and Manson investigate the use of complexity functions to control parallelism in the parallel evaluation of functional programs [RM90]. They also obtain encourag-
ing experimental results for divide-and-conquer algorithms on a parallel graph reduction machine.

Tick has given a simple algorithm to estimate relative granularities to guide task scheduling decisions for parallel logic programs [Tic90], but his analysis is not expressive enough to enable a compiler to generate code of the form “if the input size exceeds this threshold then execute these goals in parallel, else do them sequentially.”

King and Soper develop a granularity analysis algorithm similar to ours in the context of concurrent logic programs [KS90]. In their method program partitioning is done completely at compile time; calls with linear cost are coalesced and only calls with nonlinear cost are spawned into separate tasks. Recently, King and Soper apply information about data dependencies between literals to partition the literals in a clause into threads so that the execution of the literals in a thread can be totally ordered at compile time [KS92]. This static partitioning of literals may reduce overheads associated with scheduling and synchronizing processes at runtime.

Zhong et al. [ZTD*92] extend Tick’s algorithm [Tic90] to obtain more precise estimates for relative granularities, but the new analysis is still not expressive enough to enable a compiler to generate code of the form “if the input size exceeds this threshold then execute these goals in parallel, else do them sequentially.”

12.5 Summary

Parallel logic programming languages offer a great deal of scope for parallelism. However, because of the overhead associated with task creation and management, the “work available” underneath a goal should be taken into account when deciding whether or not to execute it concurrently as a separate task. This chapter described how programs may be statically analyzed to estimate task granularities. The granularity information inferred can usefully be utilized by compilers to improve the quality of code generated. The runtime overhead associated with our approach is usually quite small.
CHAPTER 13
APPLICATION II: OPTIMIZING PROGRAMS WITH FINITE-DOMAIN CONSTRAINTS

13.1 Introduction

Many problems arising in artificial intelligence and operations research can be formulated as combinatorial problems. A combinatorial problem can be a constraint satisfaction problem or a combinatorial optimization problem. A constraint satisfaction problem involves a set of \( n \) variables, a domain of values, and a set of constraints on the values the variables can assume. A solution to such a constraint satisfaction problem is an \( n \)-tuple of assignments of domain values to variables such that all the constraints in the problem are satisfied. Usually, either a single solution or all solutions are sought. A combinatorial optimization problem can be viewed as a constraint satisfaction problem, which is further subject to an objective function (a maximization or minimization function). Familiar problems that can be formulated as combinatorial problems include boolean satisfiability: deciding whether a boolean formula is satisfiable; subgraph isomorphism: given two graphs, deciding whether one graph is isomorphic to a subgraph of the other one; and graph coloring: using the fewest number of colors to color a graph so that adjacent vertices have different colors.

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This chapter will examine in particular the class of programs whose constraints involve only comparison operators (\( =, \neq, >, \geq, <, \leq \)). These logic programs are usually solved by tree search algorithms: backtracking [CM79, CKVC83, Nai86, SF87], intelligent backtracking [BP84, KL88], forward checking, looking ahead, or branch and bound algorithms [VH89]. The performance of the tree search algorithms depends heavily on the evaluation order among domain-value generators and among constraints [HE80, Nud83], and the instantiation order of domain values to variables, that is, the order in which domain values are generated by generators [DP88]. For ordering the evaluation among generators and constraints, existing logic programming systems ei-
ther dynamically order the evaluation of subgoals [CM79, CKVC83, BP84, Nai86, KL88, VH89], which inevitably incurs some runtime overhead, or use very primitive information for static ordering [SF87], which only achieves restricted benefits. For ordering the instantiation of domain values to variables, no existing system supports this feature yet. This chapter presents several source-level program transformation techniques for statically planning the subgoal evaluation order and the instantiation order of domain values to variables. These techniques allow automatic transformation of declaratively specified programs into more efficient programs. The techniques are based on the information about the number of solutions of constraint satisfaction problems. In addition to ordering subgoals and the instantiation of domain values to variables, knowledge about the number of solutions of constraint satisfaction problems can also be applied to reduce the domain of variables.

The decision versions of many constraint satisfaction problems, such as boolean satisfiability, subgraph isomorphism, and graph \( k \)-colorability, that decide whether there exists a solution satisfying all the constraints in the problem are NP-complete [Coo71, Kar72]. Thus if \( P \neq NP \), there is no polynomial time algorithm for computing the number of solutions for constraint satisfaction problems. Rivin and Zabih [RZ89] have proposed an algorithm for computing the number of solutions for constraint satisfaction problems by transforming a constraint satisfaction problem into an integer linear programming problem. If the constraint satisfaction problem has \( n \) variables and \( m \) domain values, and if the equivalent programming problem involves \( M \) equations, then the number of solutions can be determined in time \( O(nm2^{M-n}) \). In this chapter we apply the polynomial time algorithms discussed in Chapter 8 to compute upper bounds on the number of solutions for constraint satisfaction problems, and use the upper bound estimates to guide the procedure of program transformation and optimization.

In the remainder of this chapter, we will use an example to illustrate the program transformation and optimization techniques, and give experimental results of these techniques conducted on a set of benchmark programs. The measurements are conducted using SICStus Prolog [CW88] and running on Sun4. The benchmark programs are given in Appendix D and include a crypt-arithmetic problem: the send-more-money puzzle [Lau78]; the eight-queens problem; the five-houses puzzle [Lau78]; a job scheduling problem [Bar83, VH89]; a boolean satisfiability problem: a liar puzzle [Lau78]; the magic \( 3 \times 3 \) squares
problem [Lau78]; a map coloring problem [BP84]; and a network flow problem [Tar83].

The example problem is a job scheduling problem [Bar83, VH89]. The problem is to find schedules for a project that satisfy a variety of constraints among its jobs. Typical constraints in job scheduling problems include *precedence constraints*, which specify the precedence among the jobs; *disjunctive constraints*, which indicate the mutual exclusion among the jobs due to the sharing of resources; and so on. An example of precedence constraints is given by the following predicate:

\[
\text{precedence_constraints}(A, B, C, D, E, F, G, \text{End}) :-
\text{SB} \geq A + 2, \quad \text{SC} \geq A + 2, \quad \text{SD} \geq A + 2,
\text{SE} \geq \text{SB} + 3, \quad \text{SF} \geq \text{SC} + 5, \quad \text{SF} \geq \text{SD} + 6,
\text{SG} \geq \text{SE} + 2, \quad \text{SG} \geq \text{SF} + 3, \quad \text{SEnd} \geq \text{SG} + 1.
\]

Here variables \(A, \ldots, \text{End} \) represent the starting time of the jobs \(A, \ldots, \text{End} \). The constraint \(\text{SB} \geq A + 2\) specifies that the job \(A\) takes 2 units of time to complete and the job \(B\) can be started only after the job \(A\) has completed. The job \(\text{End}\) is a dummy job denoting the end of the project. An example of disjunctive constraints is given by the following predicate:

\[
\text{disjunctive_constraints}(B, C) :- \text{SC} \geq B + 3.
\text{disjunctive_constraints}(B, C) :- \text{SB} \geq C + 5.
\]

This predicate indicates that the jobs \(B\) and \(C\) cannot be performed at the same time due to the sharing of resources. We can then express the constraints for the project that involves the above precedence and disjunctive constraints as follows:

\[
\text{schedule_constraints}(A, B, C, D, E, F, G, \text{End}) :-
\text{precedence_constraints}(A, B, C, D, E, F, G, \text{End}),
\text{disjunctive_constraints}(B, C).
\]

Given the duration in which the project is supposed to be carried out, a schedule that satisfies the schedule constraints can be expressed as follows:

\[
\text{schedule(Duration, A, B, C, D, E, F, G, \text{End}) :-}
\text{generator(Duration, A)}, \text{generator(Duration, B)},
\text{generator(Duration, C)}, \text{generator(Duration, D)},
\]
Here the predicate generator/2 generates all possible time slots (natural numbers in this case) in a given duration.

13.2 Ordering Subgoals

Seki and Furukawa have presented a technique of transforming a generate and test program into a more efficient program by interleaving generators and constraints using the mode information (specifying the input and output characterization of arguments) of the generators [SF87]. In their method, subgoals are unfolded and rearranged so that constraints are interleaved into generators immediately after the constraints become active. Using this technique, the predicate schedule/9 can be transformed into the following more efficient predicate:

```
schedule(Duration,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
    generator(Duration,SA), generator(Duration,SB), SB >= SA + 2,
    generator(Duration,SC), SC >= SA + 2,
    disjunctive_constraints(SB,SC),
    generator(Duration,SD), SD >= SA + 2,
    generator(Duration,SE), SE >= SB + 3, SE >= SC + 5,
    generator(Duration,SF), SF >= SD + 6,
    generator(Duration,SG), SG >= SE + 2, SG >= SF + 3,
    generator(Duration,SEnd), SEnd >= SG + 1.
```

Notice that the rearrangement of the constraints is based on the order of the generators, and the order of the generators are usually unchanged. However, an appropriate rearrangement of generators may often significantly improve the program efficiency as well. We now examine two program transformation techniques that extend the technique of Seki and Furukawa by also rearranging the order of generators. These techniques are based on the fail-first principle [HE80]. This principle advises to try early the part that is most likely to fail so that the pruning can be performed early. The first transformation tech-
unique is based on global number of solutions information, while the second one is based on local number of solutions information.

13.2.1 Global Number of Solutions Information

Applying the fail-first principle, we can use the heuristic that chooses early the generators that generate the fewest values satisfying the constraints in the program. The $n$-cliques algorithms discussed in Chapter 8 can be employed to estimate this kind of global number of solutions information associated with each variable. In essence, the 1-expansion $n$-cliques algorithm is applied to each variable so that we can obtain separately the number of solutions associated with each domain value of the variables. Since the 1-expansion $n$-cliques algorithm is an upper bound estimation, for each variable we have an upper bound estimate on the number of domain values satisfying the set of constraints in the program. As a result, for a consistency graph involving $v$ vertices and $e$ edges, computing the number of solutions for all the variable values (namely, all the vertices in the graph) will require the time complexity $O(v^2 \cdot (v + e))$.

Applying the 1-expansion $n$-cliques algorithm to predicate schedule_constraints in the example program for the duration $[1..15]$, the number of solutions associated with each variable value is estimated as follows. We use a pair $(i,j)$ related to a variable $v$ to designate that there are $j$ solutions in which the value of $v$ is $i$. If there is no entry for a value $i$, then no solution is associated with $i$.

\[
\begin{align*}
\text{SA: } & (1,60), (2,6); \\
\text{SB: } & (3,24), (4,11), (8,18), (9,15); \\
\text{SC: } & (3,24), (4,9), (6,18), (7,15); \\
\text{SD: } & (3,32), (4,24), (5,10); \\
\text{SE: } & (11,24), (12,42); \\
\text{SF: } & (9,12), (10,28), (11,28); \\
\text{SG: } & (13,12), (14,54); \\
\text{SEnd: } & (14,6), (15,60).
\end{align*}
\]

Therefore, in the example, variable SA has the value 1 in 60 solutions, the value 2 in 6 solutions, and no other values of SA can satisfy the set of constraints in the program. Based on the number of solutions information, we can assign an initial order among the variables
as \{SA, SE, SG, SEnd, SD, SF, SB, SC\}. Nevertheless, this initial order is not necessary to work well because we still have to interleave the generators and the constraints appropriately. For example, note that in predicate `precedence_constraints/8` there is no constraint involving both variables \(SA\) and \(SE\). If we follow the initial order, in order to start executing the constraints, we still need to generate values for another variable. We will generate values for variable \(SD\) if we want to test the constraint \(SD > SA + 2\) first, or for variable \(SC\) if we want to test the constraint \(SE > SC + 5\) first. In the former case, there is no need to generate values for \(SE\) before the values of \(SD\) are generated; on the other hand, in the latter case, it is not necessary to put the generator for \(SA\) before the generators for \(SC\) and \(SE\).

We now describe a simple scheme for adjusting the initial order among variables by taking into account the relationships of variables in the constraints. We call a constraint \(c\) a forward-checkable constraint of a variable \(v\) if \(v\) appears in \(c\) and all the other variables in \(c\) are ordered ahead of \(v\) [VH89]. That is, a forward-checkable constraint of \(v\) will become active when \(v\) is instantiated. We will associate each variable \(v\) with two lists: a variable list that contains the variables appearing together with \(v\) in at least one constraint, and a constraint list that contains the forward-checkable constraints of \(v\) according to a partially determined ordering. For example, since variable \(SB\) appears with variable \(SA\) in the constraint \(SB > SA + 2\), \(SB\) is included in the variable list associated with \(SA\). In addition, if \(SB\) has been ordered ahead of \(SA\), then the constraint \(SB > SA + 2\) becomes a forward-checkable constraint of \(SA\) and is included in the constraint list associated with \(SA\). Thus for the example problem we have the following association at the beginning:

\[
\begin{align*}
SA: \{SB, SC, SD\}, \\
SB: \{SA, SC, SE\}, \\
SC: \{SA, SB, SE\}, \\
SD: \{SA, SF\}, \\
SE: \{SB, SC, SG\}, \\
SF: \{SD, SG\}, \\
SG: \{SE, SF, SEnd\}, \\
SEnd: \{SG\}, \\
\end{align*}
\]

To rearrange the order, we start with choosing the variable \(SA\) as the first variable because
it is the variable with the fewest number of values satisfying the constraints in the program.

After variable \( SA \) is chosen as the first variable to generate, the association becomes

\[
\begin{align*}
SB & : \{SC, SE\}, \{SB \geq SA + 2\} \\
SC & : \{SB, SE\}, \{SC \geq SA + 2\} \\
SD & : \{SF\}, \{SD \geq SA + 2\} \\
SE & : \{SB, SC, SG\}, \{\} \\
SF & : \{SD, SG\}, \{\} \\
SG & : \{SE, SF, SEnd\}, \{\} \\
SEnd & : \{SG\}, \{\}
\end{align*}
\]

Notice the updates of the constraint lists associated with variables \( SB, SC \) and \( SD \). Then among the variables in the list associated with \( SA \), i.e., \( \{SB, SC, SD\} \), we choose the next variable according to the following criteria:

1. choose the variable with the fewest number of values satisfying the constraints;

2. if there is a tie, choose the variable with the most number of forward-checkable constraints;

3. if there is still a tie, use the natural order of generators.

In this case, the variable chosen is \( SD \) because it has the fewest number of values satisfying the constraints. Thus at this point, the constraint \( 'SD \geq SA + 2' \) can be tested right after the values of \( SA \) and \( SD \) are generated. After that, the variables in both the list associated with \( SA \) and the list with \( SD \), namely, \( \{SB, SC, SF\} \), are under consideration. Among them, variable \( SF \) would be chosen as the next variable. We can continue this process and generate the new order as \( \{SA, SD, SF, SG, SE, SEnd, SB, SC\} \). According to this new order, an even more efficient predicate schedule/9 can be given as follows:

\[
schedule(Duration,SA,SB,SC,SD,SE,SF,SG,SEnd) :-
\begin{align*}
generator(Duration,SA), & \text{ generator(Duration,SD), } SD \geq SA + 2, \\
generator(Duration,SD), & \text{ generator(Duration,SF), } SF \geq SD + 6, \\
generator(Duration,SG), & \text{ generator(Duration,SE), } SG \geq SF + 3, \\
generator(Duration,SE), & \text{ generator(Duration,SEnd), } SEnd \geq SG + 1,
\end{align*}
\]
We have conducted experimental measurements on the benchmark programs. The programs are executed for a generate and test version (GT), a version using the transformation technique of Seki and Furukawa (SF), and a version using our goal ordering transformation technique based on global number of solutions information (GO). The result is shown in Table 13.1, where 3600.0\dagger denotes that the corresponding execution time is over one hour. The last column AT in the table shows the analysis time for estimating the number of solutions information using the 1-expansion n-cliques algorithm. The result shows that the goal ordering transformation based on the local number of solutions information produces more efficient program than Seki and Furukawa’s technique for all the benchmark programs except the liar puzzle program. The reason for the worse performance of the GO version of the liar puzzle program is as follows. The 1-expansion n-cliques algorithm infers that there is only one domain value satisfying the constraints in the program for every variable. Therefore, the ordering decision is solely based on the number of forward-checkable constraints associated with each variable. However, in the presented simple scheme, different types of constraints are not distinguished between themselves. We can easily see that equalities usually prune more search space than inequalities, and they should have

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>GT</th>
<th>SF</th>
<th>GO</th>
<th>SF/GO</th>
<th>AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-more-money</td>
<td>3600.0</td>
<td>34.640</td>
<td>0.0781</td>
<td>443.53</td>
<td>61.329</td>
</tr>
<tr>
<td>eight-queens</td>
<td>2496.65</td>
<td>0.9879</td>
<td>0.9879</td>
<td>1.00</td>
<td>12.860</td>
</tr>
<tr>
<td>five-houses</td>
<td>3600.0</td>
<td>0.3451</td>
<td>0.1382</td>
<td>2.50</td>
<td>20.800</td>
</tr>
<tr>
<td>job-scheduling</td>
<td>3600.0</td>
<td>4.2800</td>
<td>0.7531</td>
<td>5.68</td>
<td>5.790</td>
</tr>
<tr>
<td>liar-puzzle</td>
<td>0.0659</td>
<td>0.0067</td>
<td>0.0068</td>
<td>0.99</td>
<td>0.240</td>
</tr>
<tr>
<td>magic-squares</td>
<td>3600.0</td>
<td>0.8469</td>
<td>0.2322</td>
<td>3.65</td>
<td>36.629</td>
</tr>
<tr>
<td>map-coloring</td>
<td>2608.7</td>
<td>3.4029</td>
<td>0.8948</td>
<td>3.80</td>
<td>22.160</td>
</tr>
<tr>
<td>network-flow</td>
<td>1595.65</td>
<td>0.7855</td>
<td>0.6649</td>
<td>1.18</td>
<td>20.439</td>
</tr>
</tbody>
</table>

Table 13.1: Measurements for global goal ordering transformation (times in seconds)

generator(Duration,SB), SB >= SA + 2, SE >= SB + 3,
generator(Duration,SC), SC >= SA + 2, SE >= SC + 5,
disjunctive_constraints(SB,SC).

We have conducted experimental measurements on the benchmark programs. The programs are executed for a generate and test version (GT), a version using the transformation technique of Seki and Furukawa (SF), and a version using our goal ordering transformation technique based on global number of solutions information (GO). The result is shown in Table 13.1, where 3600.0\dagger denotes that the corresponding execution time is over one hour. The last column AT in the table shows the analysis time for estimating the number of solutions information using the 1-expansion n-cliques algorithm. The result shows that the goal ordering transformation based on the local number of solutions information produces more efficient program than Seki and Furukawa’s technique for all the benchmark programs except the liar puzzle program. The reason for the worse performance of the GO version of the liar puzzle program is as follows. The 1-expansion n-cliques algorithm infers that there is only one domain value satisfying the constraints in the program for every variable. Therefore, the ordering decision is solely based on the number of forward-checkable constraints associated with each variable. However, in the presented simple scheme, different types of constraints are not distinguished between themselves. We can easily see that equalities usually prune more search space than inequalities, and they should have
heavier weight than inequalities. The worse performance of the GO version of the liar puzzle program is mainly due to this kind of imprecision. This drawback can be alleviated in the technique using local number of solutions information.

13.2.2 Local Number of Solutions Information

Instead of using the number of solutions information subject to all the constraints, we now use the number of solutions information subject to each individual constraint. Besides, this information will be applied locally. Applying the fail-first principle, we will use the heuristic that chooses early the generators that generate the fewest values satisfying the active constraints instead of all the constraints in the program. Given the domain information for each variable, we will estimate the success probability of each constraint, and estimate the joint success probability of a set of active constraints. We will assume that a domain is given as a natural number interval or a set of atoms so that we can use simple interval arithmetic for arithmetic constraints and simple set manipulations for non-arithmetic equality or disequality constraints.

Since the set of constraints that become active after a variable $v$ is instantiated is just the set of forward-checkable constraints of $v$ with respect to the set of variables instantiated ahead of $v$, we will estimate the success probability of a constraint under the condition that it is forward-checkable. As a result, for each variable occurring in a constraint, we can associate with it a (possibly distinct) success probability of this constraint because variables may be instantiated in any order. For instance, let us consider the constraint ‘$SB >= SA + 2$’ in the example problem. Given the duration $[1..15]$, if $SA$ is ordered ahead of $SB$, then we can view the constraint as ‘$SB >= [1..15] + 2$’. Using interval arithmetic, we can obtain ‘$SB >= [3..17]$’. If we assume that all the values in $[3..17]$ have the same distribution, then the success probability of the constraint is 0.404 because there are altogether $15 \times 15$ distinct $(SA,SB)$ value pairs and $(13 \times 14)/2$ of them satisfy the constraint. We make the assumption that all the values in the computed interval have the same distribution for the sake of efficiency. It would be too expensive to maintain distribution information for each individual value in the interval. Similarly, we can obtain ‘$SA =< [-1..13]$’ if $SB$ is ordered ahead of $SA$. Assuming that all the values in $[-1..13]$ have the same distribution, the success probability of the constraint is also 0.404. However, in general, the success probability of a constraint may be different for different variables.
In this way, we can estimate the success probability for each constraint with respect to each variable occurring in the constraint.

We now consider the joint success probability of a conjunction or disjunction of constraints. Let $c_1, \ldots, c_n$ be a conjunction of forward-checkable constraints of a variable $v$ with the success probabilities $p_1, \ldots, p_n$. If we assume that the success probabilities of the constraints are independent of each other, then the joint success probability of this set of constraints can be estimated simply as $\prod_{i=1}^{n} P_i$. We also make the assumption that the success probabilities of the constraints are independent of each other for the sake of efficiency. Maintaining the dependency between the constraints will complicate the analysis significantly. Similarly, let $c_1, \ldots, c_n$ be a disjunction of forward-checkable constraints of a variable $v$ with the success probabilities $p_1, \ldots, p_n$. Assuming that the success probabilities of the constraints are independent of each other, the joint success probability of this set of constraints can be estimated simply as $1 - \prod_{i=1}^{n} (1 - P_i)$.

We now describe a simple scheme for ordering the variables for instantiation. For each variable $v$, we maintain two lists of constraints: a list that contains the forward-checkable constraints of $v$, and a list that contains the constraints in which $v$ appears but which is not forward-checkable. The first list of constraints has an immediate effect on the search space when $v$ is instantiated, while the second list of constraints has impact at the later stages of the search. Let $n$ be the domain size of $v$, and $p$ be the joint success probability of the set of forward-checkable constraints of $v$, namely, the first list of constraints associated with $v$. Then the number of values of $v$ satisfying the set of forward-checkable constraints can be estimated as $np$. It is interesting to note that in the context of forward checking algorithm this estimate can be interpreted as the number of values left in the domain of $v$. Simultaneously, we also estimate the joint success probability of the constraints in the second list associated with $v$. Let this probability be $q$. Then we will use the following criteria to order the variables:

1. choose the variable with the fewest number of values satisfying its forward-checkable constraints, i.e., with the smallest value of $np$;

2. if there is a tie, choose the variable with the smallest success probability $q$ (or the largest failure probability $1 - q$);

3. if there is still a tie, use the natural order of generators.
Table 13.2: Measurements for local goal ordering transformation (times in seconds)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SF</th>
<th>GO</th>
<th>SF/GO</th>
<th>LO</th>
<th>SF/LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-more-money</td>
<td>34.64</td>
<td>0.078</td>
<td>443.53</td>
<td>0.0828</td>
<td>418.36</td>
</tr>
<tr>
<td>eight-queens</td>
<td>0.9879</td>
<td>0.9879</td>
<td>1.00</td>
<td>0.8419</td>
<td>1.17</td>
</tr>
<tr>
<td>five-houses</td>
<td>0.3451</td>
<td>0.1382</td>
<td>2.50</td>
<td>0.0409</td>
<td>8.44</td>
</tr>
<tr>
<td>job-scheduling</td>
<td>4.2800</td>
<td>0.7531</td>
<td>5.68</td>
<td>0.6109</td>
<td>7.01</td>
</tr>
<tr>
<td>liar-puzzle</td>
<td>0.0067</td>
<td>0.0068</td>
<td>0.99</td>
<td>0.0066</td>
<td>1.02</td>
</tr>
<tr>
<td>magic-squares</td>
<td>0.8469</td>
<td>0.2322</td>
<td>3.65</td>
<td>0.2132</td>
<td>3.97</td>
</tr>
<tr>
<td>map-coloring</td>
<td>3.4029</td>
<td>0.8948</td>
<td>3.80</td>
<td>1.0014</td>
<td>3.40</td>
</tr>
<tr>
<td>network-flow</td>
<td>0.7855</td>
<td>0.6649</td>
<td>1.18</td>
<td>0.6649</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Apart from the rearrangement of the generators, using the success probabilities of the forward-checkable constraints, we can also rearrange the order of the active constraints. Applying the fail-first principle, we will test early the constraints that have the smallest success probability so that fewer tests will be performed.

Experimental measurements for the goal ordering transformation based on local number of solutions information have also been conducted. The version using the local number of solutions information (LO) is compared with the version using the technique of Seki and Furukawa (SF). The result is also shown in Table 13.2. The result shows that the goal ordering transformation based on the local number of solutions information produces more efficient program than Seki and Furukawa's technique for all the benchmark programs. The result also shows that there is no clear winner for the two transformation techniques based on the number of solutions information. For the send-more-money program and the map coloring program, the technique using global number of solutions information is better. On the other hand, for other programs, the technique using local number of solutions information is superior. We also note that for the eight-queens program, although the order of the generators is the same in all three versions, the better performance of the LO version is due to the rearrangement among the active constraints.

Note that the above simple scheme ignores the different costs among generators and constraints. It is possible to take into account the different costs of generators and con-
straints when making ordering decision. For example, we have used the number of procedure calls as the metric of cost to estimate the different costs of generators and constraints. For any two variables $v_1$ and $v_2$, assume that $v_1$ is ordered ahead of $v_2$. Then let $t_1$ be the cost for executing the generator for $v_1$ and the set of forward-checkable constraints of $v_1$, and $t_2$ be the cost for executing the generator for $v_2$ and the set of forward-checkable constraints of $v_2$. Also, let $n$ be the domain size of $v_1$, and $p$ be the joint success probability of the set of forward-checkable constraints of $v_1$. If these two variables are consecutively ordered, then the cost of executing them would be $t_1 + np t_2$. Similarly, we can also estimate the cost for the case when $v_2$ is ordered ahead of $v_1$. With these cost information, we will choose to use the order that produces the smaller cost.

We have also performed experimental measurements on the benchmark programs for this extended goal ordering transformation technique. However, we obtain the same variable orders as the ones produced by the transformation technique using local number of solutions information for all the benchmark programs. Besides, there are only a few differences on the order among active constraints. Therefore, we do not include this set of measurements here. Nevertheless, we expect that we can certainly benefit from this extended technique for programs with more complicated constraints.

In the constraint programming language CHIP [VH89], consistency techniques (e.g., forward checking and looking ahead) are used to solve combinatorial problems. To apply consistency techniques, the domains of variables are maintained at runtime so that inconsistent domain values can be eliminated actively through the propagation of constraints. With the domain information available at runtime, many heuristics for ordering variables for instantiation can be performed dynamically. For example, using the fail-first principle, the variable with the fewest values left in the domain is chosen as the next variable to instantiate. This dynamic ordering can improve the program efficiency in many cases. However, due to the runtime overhead incurred by dynamic ordering, in some cases applying only consistency techniques is better than applying both consistency techniques and dynamic ordering [DSP89, VH89]. In such cases static ordering provides an alternative approach to improving the program performance.
13.3 Reducing Variable Domains

Since the estimation of the n-cliques algorithm is an upper bound estimation, if the number of solutions associated with a variable value is inferred to be zero, we can safely remove this value from the corresponding domain at compile time. Moreover, recall that the 1-expansion n-cliques algorithm is applied to every variable to obtain the number of solutions for all the variable values. Knowing that a value cannot lead to a solution, we can also safely remove the corresponding vertex in the consistency graph without affecting correctness. This reduction can improve both the efficiency and precision of the 1-expansion n-cliques algorithm.

Therefore, for the example problem, the predicate generator/2 can be specialized into the following new generator predicates at compile time:

```
    gen_a(1). gen_a(2).
gen_b(3). gen_b(4). gen_b(8). gen_b(9).
gen_c(3). gen_c(4). gen_c(6). gen_c(7).
gen_e(11). gen_e(12).
gen_f(9). gen_f(10). gen_f(11).
gen_g(13). gen_g(14).
gen_end(14). gen_end(15).
```

Experimental measurements for domain reducing optimization have been conducted on the benchmark programs. Among them, the eight-queens program, the magic squares program and the map coloring program have no variable values that can be removed. Since the program transformation from a generator using recursive predicate (used in the versions GO and LO) to a generator using a set of facts usually also reduce the execution time, we use the version using facts to measure the sole effects from the domain reducing optimization. Combined with either the global (G-) or the local (L-) goal ordering technique, the programs are executed for a version without domains reduced (GNR and LNR) and a version with domains reduced (GDR and LDR). The results are shown in Table 13.3. From the measurements, combining the goal ordering transformation and the domain reducing optimization can speed up the execution of the send-more-money puzzle program and the job scheduling program by a factor of more than a hundred. Notice also
Table 13.3: Measurements for domain reducing optimization (times in seconds)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SF</th>
<th>GO</th>
<th>GNR</th>
<th>GDR</th>
<th>GNR/GDR</th>
<th>SF/GDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-more-money</td>
<td>34.640</td>
<td>0.0781</td>
<td>0.0514</td>
<td>0.0471</td>
<td>1.09</td>
<td>777.92</td>
</tr>
<tr>
<td>five-houses</td>
<td>0.3451</td>
<td>0.1382</td>
<td>0.0856</td>
<td>0.0297</td>
<td>2.88</td>
<td>11.62</td>
</tr>
<tr>
<td>job-scheduling</td>
<td>4.2800</td>
<td>0.7531</td>
<td>0.4187</td>
<td>0.0511</td>
<td>8.19</td>
<td>83.76</td>
</tr>
<tr>
<td>liar-puzzle</td>
<td>0.0067</td>
<td>0.0068</td>
<td>0.0065</td>
<td>0.0060</td>
<td>1.08</td>
<td>1.12</td>
</tr>
<tr>
<td>network-flow</td>
<td>0.7855</td>
<td>0.6649</td>
<td>0.4531</td>
<td>0.4418</td>
<td>1.03</td>
<td>1.78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>SF</th>
<th>LO</th>
<th>LNR</th>
<th>LDR</th>
<th>LNR/LDR</th>
<th>SF/LDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-more-money</td>
<td>34.640</td>
<td>0.0828</td>
<td>0.0542</td>
<td>0.0482</td>
<td>1.12</td>
<td>760.17</td>
</tr>
<tr>
<td>five-houses</td>
<td>0.3451</td>
<td>0.0409</td>
<td>0.03607</td>
<td>0.0312</td>
<td>1.15</td>
<td>11.06</td>
</tr>
<tr>
<td>job-scheduling</td>
<td>4.2800</td>
<td>0.6109</td>
<td>0.3138</td>
<td>0.0378</td>
<td>8.30</td>
<td>113.22</td>
</tr>
<tr>
<td>liar-puzzle</td>
<td>0.0067</td>
<td>0.0066</td>
<td>0.0062</td>
<td>0.0060</td>
<td>1.03</td>
<td>1.12</td>
</tr>
<tr>
<td>network-flow</td>
<td>0.7855</td>
<td>0.6649</td>
<td>0.4531</td>
<td>0.4418</td>
<td>1.03</td>
<td>1.78</td>
</tr>
</tbody>
</table>

that for the five-houses puzzle program and the liar puzzle program, the 1-expansion n-cliques algorithm infers that for each variable at most one value in the domain can satisfy the constraints in the problem. Consequently, after reducing the variable domains, these programs can be solved without any backtracking at runtime.

In the benchmark programs above, the goal ordering transformation is applied ahead of the domain reducing optimization. It is reasonable to reverse this application order. But, since domains will be represented as sets of atoms, instead of natural number intervals, after applying the domain reducing optimization, we will be forced to use set manipulations, rather than interval arithmetic, for arithmetic constraints when applying the goal ordering transformation. We use the original order because set manipulations are usually much more expensive than interval arithmetic. However, this may cause some loss in precision.

13.4 Determining Instantiation Order of Variable Values

For some applications, such as theorem proving, planning and vision problems, finding a single solution is sufficient. For some other applications, such as combinatorial opti-
mization problems, an optimal solution is sought. In this latter situation, usually, the branch and bound algorithm is employed, and we will try to find a first solution as soon as possible so that we can start the pruning early. In all these cases, the order in which the domain values are instantiated to the variables may have a profound effect on program performance.

In the example problem, suppose we are interested in obtaining any one of the schedules that satisfy the schedule constraints rather than all of them. Then we can use the number of solutions information to determine the instantiation order of variable values. A simple heuristic is to choose the value that is associated with the most number of solutions as the first value to be instantiated. Thus using the information about the number of solutions, we can rearrange the order of the clauses in the generator predicates as follows:

```plaintext
  gen_a(1).   gen_a(2).
  gen_b(3).   gen_b(8).   gen_b(9).   gen_b(4).
  gen_c(3).   gen_c(6).   gen_c(7).   gen_c(4).
  gen_d(3).   gen_d(4).   gen_d(5).
  gen_e(12).  gen_e(11).
  gen_f(11).  gen_f(10).  gen_f(9).
  gen_g(14).  gen_g(13).
  gen_end(15). gen_end(14).
```

Experimental measurements for this clause ordering transformation have been conducted on the benchmark programs. Among them, the 1-expansion n-cliques algorithm infers that for the five-houses puzzle program and the liar puzzle program, there is at most one variable value satisfying the constraints in the program for each variable, and for the map coloring program, all the variable values are associated with the same number of solutions for each variable. Thus the measurements for these three programs are not included. Combined with either the global (G-) or the local (L-) goal ordering technique, the programs are executed for a version using natural domain value order (GNO and LNO) and a version using the order generated by our clause ordering technique (GCO and LCO). The result is shown in Table 13.4. The result shows that for all the benchmark programs, the version using clause ordering technique performs better than the version using natural order.
Table 13.4: Measurements for clause ordering transformation (times in milliseconds)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>GNO</th>
<th>GCO</th>
<th>GNO/GCO</th>
<th>LNO</th>
<th>LCO</th>
<th>LNO/LCO</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-more-money</td>
<td>22.370</td>
<td>9.540</td>
<td>2.34</td>
<td>22.570</td>
<td>19.611</td>
<td>1.15</td>
</tr>
<tr>
<td>eight-queens</td>
<td>41.590</td>
<td>13.530</td>
<td>3.07</td>
<td>32.429</td>
<td>11.201</td>
<td>2.90</td>
</tr>
<tr>
<td>job-scheduling</td>
<td>0.292</td>
<td>0.243</td>
<td>1.20</td>
<td>0.280</td>
<td>0.241</td>
<td>1.16</td>
</tr>
<tr>
<td>magic-squares</td>
<td>19.251</td>
<td>5.089</td>
<td>3.78</td>
<td>50.240</td>
<td>0.790</td>
<td>63.59</td>
</tr>
<tr>
<td>network-flow</td>
<td>0.730</td>
<td>0.289</td>
<td>2.53</td>
<td>0.730</td>
<td>0.289</td>
<td>2.53</td>
</tr>
</tbody>
</table>

13.5 Summary

Combinatorial problems are often specified declaratively in logic programs with constraints over finite domains. This chapter presented several program transformation and optimization techniques for improving the performance of the class of program with finite-domain constraints. These techniques allow automatic transformation of declaratively specified programs into more efficient programs. The techniques are based on the information about the number of solutions of constraint satisfaction problems. The techniques include planning the evaluation order of body goals, reducing the domain of variables, and planning the instantiation order of variable values.
CHAPTER 14
CONCLUSIONS

This dissertation described research toward automatic complexity analysis of logic programs and two of its applications. The contributions of this research are as follows.

First, we extended the techniques developed for analyzing complexity of imperative and functional programs to logic programs by handling nondeterminism or generation of multiple solutions via backtracking. We presented the design and implementation of a (semi)-automatic worst-case complexity analysis system CASLOG. This system can conduct the worst-case analysis for several complexity measures such as argument size, number of solutions, and execution time, for a large class of logic programs. To deal with generation of multiple solutions via backtracking, we developed algorithms for inferring the number of solutions and the number of distinct solutions generated by each predicate, and the relation size of each predicate.

The algorithms for analyzing argument-size, number-of-solutions, and time complexity are based on the automatic solution of difference equations. We developed an efficient and flexible table-driven method for solving several common forms of difference equations. These difference equations include first order linear difference equations, second order linear difference equations with constant coefficients, divide-and-conquer difference equations, nonlinear difference equations that can be transformed into linear equations by transformation of taking logarithm, and nonlinear difference equations involving $\max$ and $\min$.

The inference of the relation size for a set of finite-domain constraints was also addressed. We investigated two special classes of constraints: linear arithmetic constraints and binary equality and disequality constraints. We showed that the problem of computing the relation size for a set of linear arithmetic constraints can be reduced to the problem of computing the number of $n$-cliques in a graph, and this latter problem is NP-hard. We developed an $O(v \cdot (v+e))$ approximation algorithm for estimating the number of $n$-cliques in a graph with $v$ vertices and $e$ edges. We also showed that the problem of computing
the relation size for a set of binary equality and disequality constraints can be reduced to the problem of computing the number of ways of coloring a graph, and this latter problem is NP-hard. We developed an $O(v^2 \cdot \log v + v \cdot e^2)$ approximation algorithm for estimating the number of ways of coloring a graph with $v$ vertices and $e$ edges.

Second, we developed a runtime mechanism for controlling task granularity in parallel logic programming systems. The performance of parallel systems often starts to degrade when the concurrent tasks in the systems become too fine-grained. Our approach to granularity control is based on time complexity information. With the information about the complexity for executing each procedure, we can compare the execution cost of a procedure with the average process creation overhead of the underlying system to determine at runtime if we should spawn a procedure call as a new concurrent task or just execute it sequentially. Through experimental measurements, we showed that this mechanism can substantially improve the performance of parallel systems in many cases.

Third, we developed several source-level program transformation techniques for optimizing the evaluation of logic programs containing finite-domain constraints. These techniques are based on number-of-solutions complexity information. The techniques include planning the evaluation order of subgoals, reducing the domain of variables, and planning the instantiation order of variable values. This application allows us to solve a problem by starting with a more declarative but less efficient program, and then automatically transforming it into a more efficient program. Through experimental measurements we showed that these program transformation techniques can significantly improve the efficiency of the class of programs containing finite-domain constraints in most cases.

Future research aims to extend both the capability and the applications of CASLOG. The capability of CASLOG can be extended in several way. First, we would like to incorporate the automatic inference of mode [Red84, Mel85, Bru84, MU87, Deb89], size measure [Plü90, VDS91], and domain [Mis84, YS87, PR89, IJ90] information into the system. Automatic inference of these properties has been extensively studied.

Second, we would like to continuously incorporate new difference equation schemas and transformation rules into the difference equation solver. This requires to collect more known methods for solving special forms of difference equations and exploit new methods for solving other forms of difference equations.

Third, we would like to extend the types of size measures in the system. Currently,
the available size measures in CASLOG are term-size, term-depth, list-length, and integer-value. In some programs, however, the complexity functions should be represented in terms of sizes under more refined size measures. For example, suppose a matrix of integers is represented by a list of rows and each row is represented by a list of integers; the number of integers in each row is equal to the number of columns of the matrix. If we use a straightforward matrix addition program to add two matrices, then the time complexity of the program should depend on both the number of rows of and the number of columns of the matrices. However, the currently available size measures are not expressive enough to represent the time complexity function in the expected manner.

Fourth, we would like to extend the system to handle nonground terms. Currently, we assume that all the inputs are given ground and all the outputs are produced ground. But the ability to handle partially instantiated, nonground structures is a very useful feature of logic programming languages, as discussed in Section 3.1. Hence, the applicability of the system can be greatly increased if it is able to handle nonground terms.

Fifth, apart from the upper bound complexity, we would like to estimate the lower bound complexity. In some applications, like parallelizing sequential language programs, lower bound information is more appropriate than upper bound information. In this example, we would spawn a procedure call into a separate task only if it has large enough amount of work underneath it. In some other applications, the average-case complexity is more desired than the worst-case complexity. In these cases, the mean of the upper and lower bounds may serve as an alternative to the average-case complexity, which is in general much harder to estimate.

There are many potential applications of CASLOG that can be exploited. First, we would like to extend the task granularity controlling mechanism by considering other aspects of managing concurrent processes in parallel and distributed systems. For example, consider using the argument-size complexity to estimate the communication overhead and using the time complexity to guide task migration between processors.

Second, we would like to combine the program transformation and optimization techniques for programs with finite-domain constraints and the techniques proposed by Warren [War81], Smith and Genesereth [SG85], and Natarajan [Nat86] to statically optimize query evaluation in general programs.

Third, we would like to investigate methods for proving program termination based on
the argument size information. It is interesting to see if considering also nonlinear argument size relationships can indeed handle larger class of programs than just considering linear argument size relationships.
APPENDIX A

SUMMARY OF BUILT-IN PREDICATES

The list of built-in predicates handled by CASLOG is given below.

arg(+ArgNo, +Term, -Arg)
   Arg is the ArgNo\textsuperscript{th} argument of a term Term.

arg(+ArgNo, +Term\textsubscript{1}, +Arg, -Term\textsubscript{2})
   Term\textsubscript{2} is the term extended from a term Term\textsubscript{1} by binding Arg to
   the ArgNo\textsuperscript{th} argument of Term\textsubscript{1}. (See also the predicate functor1/3.)

atom(+Term)
   Term is a non-number constant.

atomic(+Term)
   Term is a constant.

bagof(-Template, +Goal, -Bag)
   Bag is the bag of instances of Template such that Goal is provable.

compound(+Term)
   Term is a compound term.

fail
   Backtrack immediately.

findall(-Template, +Goal, -Bag)
   Bag is the bag of instances of Template such that an instance of Goal
   is provable.

float(+Term)
   Term is a float.

functor(+Term, -Symbol, -Arity)
   Symbol is the function symbol of a term Term and
   Arity is the arity associated with the symbol Symbol.
functor1(-Term, +Symbol, +Arity)

Term is a term whose function symbol is Symbol and has arity Arity.

The arguments of Term can be bound by using the predicate arg/4.

integer(+Term)

Term is an integer.

- N is +E N is the value of the arithmetic expression E.

nl Write a new line.

nonvar(+Term)

Term is a non-variable.

number(+Term)

Term is a number (namely, a float or an integer).

setof(-Template, +Goal, -Set)

Set is the set of instances of Template such that Goal is provable.

tab(+N) Write N spaces.

ttrue Succeed.

write(+Term)

Write a term Term.

! Cut all the choices taken in the current procedure.

+N =:= +M As numeric values, N is equal to M.

+N =\= +M As numeric values, N is not equal to M.

+N < +M As numeric values, N is less than M.

+N =< +M As numeric values, N is less than or equal to M.

+N > +M As numeric values, N is greater than M.

+N >= +M As numeric values, N is greater than or equal to M.

+Term1 == +Term2

Terms Term1 and Term2 are strictly identical.

+Term1 \=+ Term2

Terms Term1 and Term2 are not strictly identical.
APPENDIX B
PARAMETERS USED IN THE ANALYSIS OF ALGORITHMS

The list of parameters used in the complexity analysis of algorithms is as follows:

- $p$: the number of predicates in the program.
- $c$: the maximum number of clauses in a predicate in the program.
- $l$: the maximum number of literals in a clause in the program.
- $a$: the maximum arity of a predicate in the program.
- $t$: the maximum term size of a term in the program.
- $E$: the complexity of simplifying a symbolic expression.
- $L$: the complexity of evaluating the maximum of two symbolic expressions.
- $S$: the complexity of evaluating the minimum of two symbolic expressions.
- $A$: the complexity of adding two symbolic expressions.
- $M$: the complexity of multiplying two symbolic expressions.
- $D$: the complexity of solving a set of difference equations.

All of the parameters $E$, $L$, $S$, $A$, $M$, and $D$ are functions of the parameters $p$, $c$, $l$, $a$, and $t$. 
APPENDIX C
EXAMPLES

This appendix contains several examples of programs analyzed by the CASLOG system. In each case, we show the input program (including mode and size measure declarations that are currently necessary), the cost expressions inferred by CASLOG, and the total analysis time on a Sparcstation-2. The measure of time complexity, for each example, is the number of resolutions.

1. Naive Reverse: A very simple recursive program with two recursive predicates:

Input:

```prolog
:- mode(nrev/2,[+,-]).
:- measure(nrev/2,[length,length]).
nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), append(R1,[H],R).
:- mode(append/3,[+,+,-]).
:- measure(append/3,[length,length,length]).
append([],L,L).
append([H|L],L1,[H|R]) :- append(L,L1,R).
```

Cost Expressions Inferred:

\[ S_{nrev}^{(2)} \equiv \lambda x. x; \]
\[ Sol_{nrev} \equiv \lambda x. 1; \]
\[ Rel_{nrev} \equiv \infty; \]
\[ DSol_{nrev} \equiv \lambda x. 1; \]
\[ T_{nrev} \equiv \lambda x. 0.5x^2 + 1.5x + 1. \]
2. Fibonacci: A simple program illustrating double recursion:

Input:

```prolog
:- mode(fib/2, [+,-]).
:- measure(fib/2, [nat,nat]).
fib(0,0).
fib(1,1).
fib(M,N) :-
    M > 1, M1 is M-1, M2 is M-2,
    fib(M1,N1), fib(M2,N2), N is N1+N2.
```

Cost Expressions Inferred:

\[ Sx^{(2)}_{fib} \equiv \lambda x. 0.447 \times 1.618^x - 0.447 \times (-0.618)^x; \]
\[ Sol_{fib} \equiv \lambda x. 1; \]
\[ Rel_{fib} \equiv \infty; \]
\[ DSol_{fib} \equiv \lambda x. 1; \]
\[ T_{fib} \equiv \lambda x. 5.065 \times 1.618^x + 1.934 \times (-0.618)^x - 5. \]

Total Analysis Time: 0.20 secs.

3. Flatten: This program flattens nested lists into a flat list. It shows how CASLOG uses knowledge about the behavior of control constructs such as cut (!) to infer mutual exclusion between clauses, thereby allowing a more precise analysis.

Input:
:- mode(flatten/2,[+,-]).
:- measure(flatten/2,[size,length]).

flatten(X,[X]) :- atomic(X), X \= [],!.
flatten([],[]).
flatten([X|Xs],Ys) :-
    flatten(X,Ys1), flatten(Xs,Ys2), append(Ys1,Ys2,Ys).

Cost Expressions Inferred:

\[ S_{\text{flatten}}^{(2)} \equiv \lambda x. x; \]
\[ \text{Sol}_{\text{flatten}} \equiv \lambda x. 1; \]
\[ \text{Rel}_{\text{flatten}} \equiv \infty; \]
\[ \text{DSol}_{\text{flatten}} \equiv \lambda x. 1; \]
\[ T_{\text{flatten}} \equiv \lambda x. 2x^2 + x + 2. \]

Total Analysis Time: 0.21 secs.

4. Towers of Hanoi:

Input:

:- mode(hanoi/5,[+,+,+,+,+-]).
:- measure(hanoi/5,[nat,?,?,?,length]).

hanoi(1,A,B,C,[mv(A,C)]).

hanoi(N,A,B,C,M) :-
    N > 1, N1 is N-1,
    hanoi(N1,A,C,B,M1), hanoi(N1,B,A,C,M2),
    append(M1,[mv(A,C)],T), append(T,M2,M).

Cost Expressions Inferred:

\[ S_{\text{hanoi}}^{(5)} \equiv \lambda x. 2^x - 1; \]
\[ \text{Sol}_{\text{hanoi}} \equiv \lambda x. 1; \]
\[ R_{\text{hanoi}} \equiv \infty; \]
\[ DS_{\text{hanoi}} \equiv \lambda x. 1; \]
\[ T_{\text{hanoi}} \equiv \lambda x. x2^x + 2^{x+1} - 4. \]

Total Analysis Time: 0.49 secs.

5. Quicksort: A divide-and-conquer program. CASLOG has trouble with this one because it does not keep track of the fact that the size of the two outputs of \texttt{part/4} are not independent, and as a result gives a rather pessimistic estimate of the time complexity of \texttt{qsort/2}.

Input:

\begin{verbatim}
:- mode(qsort/2, [+,-]).
:- measure(qsort/2, [length,length]).
qusort([], []).
qusort([First|L1], L2) :-
    part(First, L1, Ls, Lg),
qusort(Ls, Ls2), qsort(Lg, Lg2),
    append(Ls2, [First|Lg2], L2).

:- mode(part/4, [+,-,-,-]).
:- measure(part/4, [?, length, length, length]).
part(F, [], [], []).
part(F, [X|Y], [X|Y1], Y2) :- X < F, part(F, Y, Y1, Y2).
part(F, [X|Y], Y1, [X|Y2]) :- X > F, part(F, Y, Y1, Y2).
\end{verbatim}

Cost Expressions Inferred:

\[ S_{\text{qsort}}^{(2)} \equiv \lambda x. 2^x - 1; \]
\[ Sol_{\text{qsort}} \equiv \lambda x. 1; \]
\[ Rel_{\text{qsort}} \equiv \infty; \]
\[ DS_{\text{qsort}} \equiv \lambda x. 1; \]
\[ T_{\text{qsort}} \equiv \lambda x. \sum_{i=1}^{x} (3i2^{x-i}) + x2^{x-1} + 1. \]
\[ S_{x_{\text{part}}}^{(a)} \equiv \lambda x. x; \]
\[ S_{x_{\text{part}}}^{(4)} \equiv \lambda x. x; \]
\[ \text{Sol}_{\text{part}} \equiv \lambda x. 1; \]
\[ \text{Rel}_{\text{part}} \equiv \infty; \]
\[ D\text{Sol}_{\text{part}} \equiv \lambda x. 1; \]
\[ T_{\text{part}} \equiv \lambda x. 3x + 1. \]

**Total Analysis Time**: 0.55 secs.

6. **N-Queens**: A nondeterministic predicate that can generate multiple solutions via backtracking:

**Input**: 

```prolog
:- mode(safe/2, [+,-]).
:- measure(safe/2, [nat,length]).
safe(N,Queens) :- extend(N,N,Queens).

:- mode(extend/3, [+,-]).
:- measure(extend/3, [nat,nat,length]).
extend(0,_,[]).
extend(M,N,[q(M,Q)|Selected]) :-
    M > 0, M1 is M-1,
    extend(M1,N,Selected), choose(N,Q), consistent(q(M,Q),Selected).

:- mode(consistent/2, [+,-]).
:- measure(consistent/2, [? ,length]).
consistent(_,[]).
consistent(Q,[Q1|Rest]) :- noattack(Q,Q1), consistent(Q,Rest).

:- mode(noattack/2, [+,-]).
:- measure(noattack/2, [? ,?]).
noattack(q(X1,Y1),q(X2,Y2)) :-
```

```prolog
```
\[ Y_1 = Y_2, \ X \ is \ X_1 - X_2, \ Y \ is \ Y_1 - Y_2, \ Z \ is \ Y_2 - Y_1, \ X = Y, \ X = Z. \]

\[ \text{:} \text{- mode(choose/2, [+,-]).} \]
\[ \text{:} \text{- measure(choose/2, [nat,nat]).} \]
\[ \text{choose}(N, N) : - N > 0. \]
\[ \text{choose}(N, M) : - N > 0, \ N_1 \ is \ N-1, \ \text{choose}(N_1, M). \]

Cost Expressions Inferred:

\[ S_{\text{safe}}^{(2)} \equiv \lambda x. x; \]
\[ S_{\text{safe}} \equiv \lambda x. x^r; \]
\[ \text{Rel}_{\text{safe}} \equiv \infty; \]
\[ D\text{Sol}_{\text{safe}} \equiv \lambda x. x^r; \]
\[ T_{\text{safe}} \equiv \lambda x. \sum_{i=1}^{r}(8ix^i) - 2x^{r+1}/(x-1) + 2x/(x-1) + 3x + 3. \]

\[ S_{\text{extend}}^{(3)} \equiv \lambda(x, y). x; \]
\[ S_{\text{extend}} \equiv \lambda(x, y). y^r; \]
\[ \text{Rel}_{\text{extend}} \equiv \infty; \]
\[ D\text{Sol}_{\text{extend}} \equiv \lambda(x, y). y^r; \]
\[ T_{\text{extend}} \equiv \lambda(x, y). \sum_{i=1}^{r}(8iy^i) - 2y^{r+1}/(y-1) + 2y/(y-1) + 3x + 2. \]

\[ S_{\text{consistent}} \equiv \lambda x. 1; \]
\[ \text{Rel}_{\text{consistent}} \equiv \infty; \]
\[ D\text{Sol}_{\text{consistent}} \equiv \lambda x. 1; \]
\[ T_{\text{consistent}} \equiv \lambda x. 8x + 1. \]

\[ S_{\text{noattack}} \equiv \lambda x. 1; \]
\[ \text{Rel}_{\text{noattack}} \equiv \infty; \]
\[ D\text{Sol}_{\text{noattack}} \equiv \lambda x. 1; \]
\[ T_{\text{noattack}} \equiv \lambda x. 7. \]

\[ Sz_{\text{choose}}^{(2)} \equiv \lambda x. x + 1; \]
7. Permutation: A nondeterministic program that generates permutations of a list. Note that for the predicate select/3, implicit failure in the base case has to be accounted for explicitly.

Input:

```
:- mode(perm/2, [+,-]).
:- measure(perm/2, [length, length]).
perm([], []).
perm([X|Xs], [R|Rs]) :- select(R, [X|Xs], Y), perm(Y, Rs).

:- mode(select/3, [-,+,-]).
:- measure(select/3, [\_, length, length]).
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

Cost Expressions Inferred:

\( \text{Sz}_{\text{perm}}^{(2)} = \lambda x. \ x; \)
\( \text{Sol}_{\text{perm}} = \lambda x. \ x!; \)
\( \text{Rel}_{\text{perm}} = \infty; \)
\( \text{DSol}_{\text{perm}} = \lambda x. \ x!; \)
\( \text{T}_{\text{perm}} = \lambda x. \sum_{i=1}^{x} (2ix/i!) + \sum_{i=1}^{x} (x!/i!) + x!. \)

\( \text{Sz}_{\text{select}}^{(1)} = \lambda x. \bot; \)
\( \text{Sz}_{\text{select}}^{(2)} = \lambda x. \ x - 1; \)
\( \text{Sol}_{\text{select}} = \lambda x. \ x; \)
8. Eight-Queens: A very different program from the \( n \)-queens program shown above, this illustrates how conjunction of linear arithmetic constraints are handled.

Input:

\[
\begin{align*}
\text{:- mode(eight_queens/8,[-,-,-,-,-,-,-]).} \\
\text{:- measure(eight_queens/8,[nat,nat,nat,nat,nat,nat,nat]).} \\
\text{eight_queens(X1,X2,X3,X4,X5,X6,X7,X8) :-} \\
\quad \text{generator(8,8,[X1,X2,X3,X4,X5,X6,X7,X8]),} \\
\quad \text{queens(X1,X2,X3,X4,X5,X6,X7,X8).} \\
\text{:- mode(queens/8,[+;++;++;++;++;++]).} \\
\text{:- measure(queens/8,[nat,nat,nat,nat,nat,nat,nat]).} \\
\text{:- domain(queens/8,[1-8,1-8,1-8,1-8,1-8,1-8,1-8,1-8]).} \\
\text{queens(X1,X2,X3,X4,X5,X6,X7,X8) :- safe([X1,X2,X3,X4,X5,X6,X7,X8]).} \\
\text{:- mode(safe/1,[+]).} \\
\text{:- measure(safe/1,[length]).} \\
\text{safe([]).} \\
\text{safe([X|L]) :- noattacks(L,X,1), safe(L).} \\
\text{:- mode(noattacks/3,[+,+,+]).} \\
\text{:- measure(noattacks/3,[length,nat,?]).} \\
\text{noattacks([],-,-).} \\
\text{noattacks([Y|L],X,D) :-} \\
\quad \text{noattack(X,Y,D), D1 is D+1, noattacks(L,X,D1).} \\
\text{:- mode(noattack/3,[+,+,+]).}
\end{align*}
\]
\(-\) measure(noattack/3, [nat, nat, ?]).

noattack(X, Y, D) :- X \neq Y, Y - X =\neq D, Y - X =\neq -D.

\(-\) mode(generator/3, [+ , + , -]).

\(-\) measure(generator/3, [nat, nat, length]).

generator(0, _, []).

generator(M, N, [Q | L]) :-
    M > 0, choose(N, Q), M is M - 1, generator(M1, N, L).

Cost Expressions Inferred:

\[
\begin{align*}
S_{\text{eight-queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). \bot; \\
\text{Sol}_{\text{eight-queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). 16,777,216; \\
\text{Rel}_{\text{eight-queens}} & \equiv 46,312; \\
\text{DSol}_{\text{eight-queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). 46,312; \\
\text{T}_{\text{eight-queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). 3,273,953,859.
\end{align*}
\]

\[
\begin{align*}
\text{Sol}_{\text{queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). 1; \\
\text{Rel}_{\text{queens}} & \equiv 46,312; \\
\text{DSol}_{\text{queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). 1; \\
\text{T}_{\text{queens}} & \equiv \lambda (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8). 186.
\end{align*}
\]

\[
\begin{align*}
\text{Sol}_{\text{safe}} & \equiv \lambda x. 1; \\
\text{Rel}_{\text{safe}} & \equiv \infty; \\
\text{DSol}_{\text{safe}} & \equiv \lambda x. 1; \\
\text{T}_{\text{safe}} & \equiv \lambda x. 3x^2 - x + 1.
\end{align*}
\]

\[
\begin{align*}
\text{Sol}_{\text{noattacks}} & \equiv \lambda (x, y). 1; \\
\text{Rel}_{\text{noattacks}} & \equiv \infty; \\
\text{DSol}_{\text{noattacks}} & \equiv \lambda (x, y). 1; \\
\text{T}_{\text{noattacks}} & \equiv \lambda (x, y). 6x + 1.
\end{align*}
\]
9. Scheduling: A program that illustrates the handling of disjunction of linear arithmetic constraints and the all-solutions predicate setof/3.

Input:

```prolog
:- mode(schedule/1, [-]).
:- measure(schedule/1, []).

schedule(MinTime) :-
    setof(H, A-B-C-D-E-F-G-legal_schedule(A,B,C,D,E,F,G,H),Time),
    earliest_completion(Time, MinTime).

:- mode(legal_schedule/8, [-,-,-,-,-,-,-,-]).
:- measure(legal_schedule/8, [nat,nat,nat,nat,nat,nat,nat,nat]).
legal_schedule(A,B,C,D,E,F,G,H) :-
    generator(8,14,[A,B,C,D,E,F,G,H]),
    schedule_constraints(A,B,C,D,E,F,G,H).

:- mode(schedule_constraints/8, [+,-,+,-,+,-,+]).
:- measure(schedule_constraints/8, [nat,nat,nat,nat,nat,nat,nat,nat]).
:- domain(schedule_constraints/8, [1-14,1-14,1-14,1-14,1-14,1-14,1-14,1-14,1-14,1-14]).
```

Total Analysis Time: 2.51 secs.
schedule_constraints(A,B,C,D,E,F,G,H) :-
    precedence_constraints(A,B,C,D,E,F,G,H),
    disjunctive_constraints(B,C).

:- mode(precedence_constraints/8, [+,-,-,-,-,-,-,-]).
:- measure(precedence_constraints/8, [nat,nat,nat,nat,nat,nat,nat,nat]).
:- domain(precedence_constraints/8, [1-14,1-14,1-14,1-14,1-14,1-14,1-14,1-14]).
precedence_constraints(A,B,C,D,E,F,G,H) :-
    B >= A+2, C >= A+2, D >= A+2, E >= B+3, E >= C+6,
    F >= D+6, G >= E+2, G >= F+3, H >= G+1.

:- mode(disjunctive_constraints/2, [+,-]).
:- measure(disjunctive_constraints/2, [nat,nat]).
:- domain(disjunctive_constraints/2, [1-14,1-14]).
disjunctive_constraints(B,C) :- C >= B+3.
disjunctive_constraints(B,C) :- B >= C+5.

:- mode(earliest_completion/3, [+,-]).
:- measure(earliest_completion/3, [length,?]).
earliest_completion([], 10000).
earliest_completion([T|Time], MinTime) :-
    earliest_completion(Time, MTime), min(T, MTime, MinTime).

:- mode(min/3, [+,-,-]).
:- measure(min/3, [?,?,-]).
min(X,Y,X) :- X <= Y.
min(X,Y,Y) :- X > Y.

Cost Expressions Inferred:
\( S_{\text{schedule}} \equiv 1; \)
\( \text{Rel}_{\text{schedule}} \equiv 1; \)
\( D S_{\text{schedule}} \equiv 1; \)
\( T_{\text{schedule}} \equiv 80,914,118,300. \)

\( S_{2(1, 2, 3, 4, 5, 6, 7)} \equiv \perp; \)
\( S_{\text{legal}_1} \equiv 5,125,781,250; \)
\( \text{Rel}_{\text{legal}_1} \equiv 6; \)
\( D S_{\text{legal}_1} \equiv 6; \)
\( T_{\text{legal}_1} \equiv 60,410,993,298. \)

\( S_{\text{schedule}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5, x_6, x_7). 2; \)
\( \text{Rel}_{\text{schedule}_{\text{constraints}}} \equiv 6; \)
\( D S_{\text{schedule}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5, x_6, x_7). 2; \)
\( T_{\text{schedule}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5, x_6, x_7). 15. \)

\( S_{\text{precedence}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5, x_6, x_7). 1; \)
\( \text{Rel}_{\text{precedence}_{\text{constraints}}} \equiv 228; \)
\( D S_{\text{precedence}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5, x_6, x_7). 1; \)
\( T_{\text{precedence}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5, x_6, x_7). 10. \)

\( S_{\text{disjunctive}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3). 2; \)
\( \text{Rel}_{\text{disjunctive}_{\text{constraints}}} \equiv 111; \)
\( D S_{\text{disjunctive}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3). 2; \)
\( T_{\text{disjunctive}_{\text{constraints}}} \equiv \lambda(x_1, x_2, x_3). 4. \)

\( S_{\text{earliest}_{\text{completion}}} \equiv \lambda(x). 1; \)
\( \text{Rel}_{\text{earliest}_{\text{completion}}} \equiv \infty; \)
\( D S_{\text{earliest}_{\text{completion}}} \equiv \lambda(x). 1; \)
\( T_{\text{earliest}_{\text{completion}}} \equiv \lambda(x). 4x + 1. \)

\( S_{\text{min}} \equiv \lambda(x, y). 1; \)
Rel_{\text{min}} \equiv \infty;
DSol_{\text{min}} \equiv \lambda(x,y).1;
T_{\text{min}} \equiv \lambda(x,y).3.

Total Analysis Time: 3.57 secs.

10. Map Coloring: A simple program that illustrates the handling of binary nonequality constraints. The predicate c/3 is included to illustrate the use of unfolding during analysis.

Input:

```prolog
:- mode(map.color/5,[,-,-,-,-]).
:- measure(map.color/5,[size,size,size,size,size]).
map.color(A,B,C,D,E) :-
    color(A), color(B), color(C), color(D), color(E),
    legal.coloring(A,B,C,D,E).

:- mode(legal.coloring/5, [+,+,+,+]).
:- measure(legal.coloring/5,[size,size,size,size,size]).
:- domain(legal.coloring/5,[[blue,green,orange,red,yellow],
    [blue,green,orange,red,yellow], [blue,green,orange,red,yellow],
    [blue,green,orange,red,yellow], [blue,green,orange,red,yellow]]).
legal.coloring(A,B,C,D,E) :-
    A =\= B, A =\= C, A =\= D, A =\= E, c(B,C,D), C =\= E.

:- mode(c/3,[+,+,+]).
:- measure(c/3,[size,size,size]).
:- domain(c/3,[[blue,green,orange,red,yellow],
    [blue,green,orange,red,yellow], [blue,green,orange,red,yellow],
    [blue,green,orange,red,yellow], [blue,green,orange,red,yellow]]).
c(X,Y,Z) :- X =\= Y, X =\= Z.

:- mode(color/1,[-]).
:- measure(color/1,[size]).
```
Cost Expressions Inferred:

\[
S_{z_{\text{map.color}}}^{(1, 2, 3, 4, 5)} \equiv 1; \\
S_{\text{Sol}_{\text{map.color}}} \equiv 3,125; \\
R_{\text{Rel}_{\text{map.color}}} \equiv 540; \\
D_{\text{Sol}_{\text{map.color}}} \equiv 540; \\
T_{\text{map.color}} \equiv 32,031.
\]

\[
S_{\text{Sol}_{\text{legal.coloring}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5). 1; \\
R_{\text{Rel}_{\text{legal.coloring}}} \equiv 540; \\
D_{\text{Sol}_{\text{legal.coloring}}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5). 1; \\
T_{\text{legal.coloring}} \equiv \lambda(x_1, x_2, x_3, x_4, x_5). 9.
\]

\[
S_{\text{Sol}_c} \equiv \lambda(x_1, x_2, x_3). 1; \\
R_{\text{Rel}_c} \equiv 80; \\
D_{\text{Sol}_c} \equiv \lambda(x_1, x_2, x_3). 1; \\
T_{\text{c}} \equiv \lambda(x_1, x_2, x_3). 3.
\]

\[
S_{\text{color}}^{(1)} \equiv 1; \\
S_{\text{Sol}_c} \equiv 5; \\
R_{\text{Rel}_c} \equiv 5; \\
D_{\text{Sol}_c} \equiv 5; \\
T_{\text{color}} \equiv 5.
\]

Total Analysis Time: 0.63 secs.
APPENDIX D

BENCHMARKS OF FINITE-DOMAIN CONSTRAINTS

This appendix contains the benchmarks for the program transformation and optimization techniques discussed in Chapter 13.

1. Send-More-Money: A crypt-arithmetic puzzle: Assign a different digit from \{0, \ldots, 9\} to each letter in \{S, E, N, D, M, O, R, Y\} such that SEND + MORE = MONEY.

sendmore(R1,R2,R3,R4,S,E,N,D,M,O,R,Y) :-
generator(R1,R2,R3,R4,S,E,N,D,M,O,R,Y),
constraints(R1,R2,R3,R4,S,E,N,D,M,O,R,Y).

generator(R1,R2,R3,R4,S,E,N,D,M,O,R,Y) :-
choose(O,1,R1), choose(O,1,R2), choose(O,1,R3), choose(O,1,R4),
choose(O,9,S), choose(O,9,E), choose(O,9,N), choose(O,9,D),
choose(O,9,M), choose(O,9,O), choose(O,9,R), choose(O,9,Y).

choose(N,M,N) :- N =< M.
choose(N,M,C) :- N < M, N1 is N+1, choose(N1,M,C).

constraints(R1,R2,R3,R4,S,E,N,D,M,O,R,Y) :-
alloecdifferent([S,E,N,D,M,O,R,Y]),
S =\= 0, M =\= 0, R1 =:= M,
R2 + S + M =:= O + 10*R1,
R3 + E + O =:= N + 10*R2,
R4 + N + R =:= E + 10*R3,
alldifferent([]).
alldifferent([X|Y]) :- outof(X,Y), alldifferent(Y).

outof([],[]).
outof(X,[F|T]) :- X =\= F, outof(X,T).

2. Eight-Queens: The eight-queens problem: Place 8 "queens" on an 8 x 8 chessboard such that each queen is placed in a different row and no two queens are in the same column, or diagonals.

eight_queens(X1,X2,X3,X4,X5,X6,X7,X8) :-
generator(X1,X2,X3,X4,X5,X6,X7,X8),
constraints(X1,X2,X3,X4,X5,X6,X7,X8).

generator(X1,X2,X3,X4,X5,X6,X7,X8) :-
choose(1,8,X1), choose(1,8,X2), choose(1,8,X3), choose(1,8,X4),
choose(1,8,X5), choose(1,8,X6), choose(1,8,X7), choose(1,8,X8).

choose(N,M,N) :- N =< M.
choose(N,M,C) :- N < M, N1 is N+1, choose(N1,M,C).

constraints(X1,X2,X3,X4,X5,X6,X7,X8) :-
safe([X1,X2,X3,X4,X5,X6,X7,X8]).

safe([]).
safe([X|L]) :- noattacks(L,X,1), safe(L).

noattacks([],_,_).
noattacks([Y|L],X,D) :-
noattack(X,Y,D), D1 is D+1, noattacks(L,X,D1).

noattack(X,Y,D) :- X =\= Y, Y-X =\= D, Y-X =\= -D.
3. **Five-Houses:** The five houses puzzle: Five men with different nationalities (N) lives in five consecutive houses. They practice five different professions (P), and each of them has a different favorite animal (A) and a different favorite drink (D). The five houses are painted different colors (C). The following fact are known:

1. The Englishman lives in a red house.
2. The Spaniard owns a dog.
3. The Japanese is a painted.
4. The Italian drinks tea.
5. The Norwegian lives in the first house on the left.
6. The owner of the green house drinks coffee.
7. The green house is on the right of the white one.
8. The sculptor breeds snails.
9. The diplomat lives in the yellow house.
10. Milk is drunk in the middle house.
11. The Norwegian's house is next to the blue one.
12. The violinist drinks fruit juice.
13. The fox is in the house next to that of the doctor.
14. The horse is in the house next to that of the diplomat.

The problem is to identify who owns a zebra and who drinks water.

```prolog
five_houses(N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,P1,P2,P3,P4,P5,
A1,A2,A3,A4,A5,D1,D2,D3,D4,D5) :-
generator(N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,P1,P2,P3,P4,P5,
A1,A2,A3,A4,A5,D1,D2,D3,D4,D5),
constraints(N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,P1,P2,P3,P4,P5,
A1,A2,A3,A4,A5,D1,D2,D3,D4,D5).

generator(N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,P1,P2,P3,P4,P5,
A1,A2,A3,A4,A5,D1,D2,D3,D4,D5) :-
choose(1,5,N1), choose(1,5,N2), choose(1,5,N3), choose(1,5,N4),
choose(1,5,N5), choose(1,5,C1), choose(1,5,C2), choose(1,5,C3),
choose(1,5,C4), choose(1,5,C5), choose(1,5,P1), choose(1,5,P2),
```

223
choose(1,5,P3), choose(1,5,P4), choose(1,5,P5), choose(1,5,A1),
choose(1,5,A2), choose(1,5,A3), choose(1,5,A4), choose(1,5,A5),
choose(1,5,D1), choose(1,5,D2), choose(1,5,D3), choose(1,5,D4),
choose(1,5,D5).

choose(N,M,N) :- N =< M.
choose(N,M,C) :- N < M, N1 is N+1, choose(N1,M,C).

constraints(N1,N2,N3,N4,N5,C1,C2,C3,C4,C6,P1,P2,P3,P4,P6,
A1,A2,A3,A4,A5,D1,D2,D3,D4,D6) :-
N1 =:= C2, N2 =:= A1, N3 =:= P1, N4 =:= D3, N5 =:= 1,
D5 =:= 3, P3 =:= D1, C1 =:= D4, P5 =:= A4, P2 =:= C3,
C1 =:= C5 + 1,
plusorminus(A3,P4,1),
plusorminus(A5,P2,1),
plusorminus(N5,C4,1),
alldifferent([N1,N2,N3,N4,N5]),
alldifferent([C1,C2,C3,C4,C5]),
alldifferent([P1,P2,P3,P4,P5]),
alldifferent([A1,A2,A3,A4,A5]),
alldifferent([D1,D2,D3,D4,D5]).

plusorminus(X,Y,C) :- X =:= Y-C.
plusorminus(X,Y,C) :- X =:= Y+C.

alldifferent([]).
alldifferent([X|Y]) :- outof(X,Y), alldifferent(Y).

outof(X,[]).
outof(X,[F|T]) :- X \= F, outof(X,T).
4. **Job-Scheduling:** A job-scheduling problem: Find a schedule for the set of jobs \{A, B, C, D, E, F, G, End\} in the duration [1..16] such that the precedence constraints specified in the following graph and a disjunctive constraint between jobs B and C are satisfied.

```prolog
job_scheduling(A,B,C,D,E,F,G,End) :-
generator(A,B,C,D,E,F,G,End),
constraints(A,B,C,D,E,F,G,End).

generator(A,B,C,D,E,F,G,End) :-
    choose(1,15,A), choose(1,15,B), choose(1,16,C),
    choose(1,15,D), choose(1,15,E), choose(1,16,F),
    choose(1,15,G), choose(1,15,End).

choose(N,M,N) :- N =< M.
choose(N,M,C) :- N < M, N1 is N+1, choose(N1,M,C).

constraints(A,B,C,D,E,F,G,End) :-
    precedence_constraints(A,B,C,D,E,F,G,End),
    disjunctive_constraints(B,C).

precedence_constraints(A,B,C,D,E,F,G,End) :-
    B =\geq A+2, C =\geq A+2, D =\geq A+2, E =\geq B+3, E =\geq C+5,
    F =\geq D+6, G =\geq E+2, G =\geq F+3, End =\geq G+1.

disjunctive_constraints(B,C) :- C =\geq B+3.
disjunctive_constraints(B,C) :- B =\geq C+5.
```

![Graph of the job scheduling problem](image-url)
5. **Liar-Puzzle**: A boolean satisfiability problem: A vote on an important issue is taken in a parliament. Whereas party $\alpha$ gets the better of party $\beta$, some journalists wrote that Mr. Z, a member of party $\beta$, received from party $\alpha$, a large sum of money so that he would vote for $\alpha$. The committee of the inquiry gathers the evidence of five persons from party $\alpha$, but knows that each of these five persons makes exactly one true and one false declaration.

Mr. A states: none of these five persons is the treasurer of the $\alpha$ secret funds, but I know that Mr. Z received the money from Mr. D via Mr. B. Mr. B states: Mr. Z did indeed receive the money, and Mr. C is the treasurer. Mr. C states: Mr. A’s second assertion is true, so is the second statement of Mr. B. Mr. D states: Mr. Z received no money at all, and I am myself the treasurer of our party. Finally, Mr. D states: Mr. D was never treasurer, and I received money from Mr. B and forwarded it to Mr. Z.

\[
\text{liar_puzzle}(A1, A2, B1, B2, C1, C2, D1, D2, E1, E2) :-
\]

\[
\text{generator}(A1, A2, B1, B2, C1, C2, D1, D2, E1, E2),
\]

\[
\text{constraints}(A1, A2, B1, B2, C1, C2, D1, D2, E1, E2).
\]

\[
\text{generator}(A1, A2, B1, B2, C1, C2, D1, D2, E1, E2) :-
\]

\[
\text{choose}(0, 1, A1), \text{choose}(0, 1, A2), \text{choose}(0, 1, B1), \text{choose}(0, 1, B2),
\]

\[
\text{choose}(0, 1, C1), \text{choose}(0, 1, C2), \text{choose}(0, 1, D1), \text{choose}(0, 1, D2),
\]

\[
\text{choose}(0, 1, E1), \text{choose}(0, 1, E2).
\]

\[
\text{choose}(N, M, N) :- N =< M.
\]

\[
\text{choose}(N, M, C) :- N < M, N1 is N+1, \text{choose}(N1, M, C).
\]

\[
\text{constraints}(A1, A2, B1, B2, C1, C2, D1, D2, E1, E2) :-
\]

\[
A1 + A2 =:= 1, B1 + B2 =:= 1, C1 + C2 =:= 1,
\]

\[
D1 + D2 =:= 1, E1 + E2 =:= 1,
\]

\[
A1 + B2 =< 1, A1 + D2 =< 1, D2 + E1 =< 1,
\]

\[
D2 + B2 =< 1, D1 + B1 =< 1,
\]

\[
\]
6. Magic-Square: The magic $3 \times 3$ square problem: Assign a different digit from \{1, \ldots, 9\} to each entry of a $3 \times 3$ square such that the sum of the digits in each row, column, and diagonal is the same.

\[
\text{magic_squares}(F_1,F_2,F_3,F_4,F_5,F_6,F_7,F_8,F_9) :-
\text{generator}(F_1,F_2,F_3,F_4,F_5,F_6,F_7,F_8,F_9),
\text{constraints}(F_1,F_2,F_3,F_4,F_5,F_6,F_7,F_8,F_9).
\]

\[
\text{generator}(F_1,F_2,F_3,F_4,F_5,F_6,F_7,F_8,F_9) :-
\text{choose}(1,9,F_1), \text{choose}(1,9,F_2), \text{choose}(1,9,F_3), \text{choose}(1,9,F_4),
\text{choose}(1,9,F_5), \text{choose}(1,9,F_6), \text{choose}(1,9,F_7), \text{choose}(1,9,F_8),
\text{choose}(1,9,F_9).
\]

\[
\text{choose}(N,M,N) :- N \leq M.
\]

\[
\text{choose}(N,M,C) :- N < M, N_1 \equiv N+1, \text{choose}(N_1,M,C).
\]

\[
\text{constraints}(F_1,F_2,F_3,F_4,F_5,F_6,F_7,F_8,F_9) :-
\text{alldifferent}([F_1,F_2,F_3,F_4,F_5,F_6,F_7,F_8,F_9]),
F_1+F_2+F_3 \equiv 15, F_4+F_5+F_6 \equiv 15, F_7+F_8+F_9 \equiv 15,
F_1+F_4+F_7 \equiv 15, F_2+F_5+F_8 \equiv 15, F_3+F_6+F_9 \equiv 15,
F_1+F_5+F_9 \equiv 15, F_3+F_5+F_7 \equiv 15.
\]

\[
\text{alldifferent}([]).
\]

\[
\text{alldifferent}([X|Y]) :- \text{outof}(X,Y), \text{alldifferent}(Y).
\]

\[
\text{outof}(X,[]).
\]

\[
\text{outof}(X,[F|T]) :- X \neq F, \text{outof}(X,T).
\]
7. Map-Coloring: A map coloring problem: Color the following map using four colors.

```
map_coloring(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13) :-
generator(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13),
constraints(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13).

generator(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13) :-
choose(R1), choose(R2), choose(R3), choose(R4), choose(R5),
choose(R6), choose(R7), choose(R8), choose(R9), choose(R10),
choose(R11), choose(R12), choose(R13).

constraints(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13) :-
R2 \= R1, R3 \= R2, R4 \= R2, R4 \= R3,
R5 \= R3, R5 \= R4, R6 \= R5, R7 \= R1,
R7 \= R2, R7 \= R4, R8 \= R7, R9 \= R4,
R9 \= R8, R10 \= R4, R10 \= R5, R10 \= R6,
R10 \= R9, R11 \= R6, R11 \= R10, R12 \= R9,
R12 \= R11, R13 \= R1, R13 \= R2, R13 \= R3,
R13 \= R5, R13 \= R6, R13 \= R7, R13 \= R8,
R13 \= R9, R13 \= R11, R13 \= R12.

choose(blue). choose(green). choose(red). choose(yellow).
```
8. **Network-Flow:** A network flow problem: Find a flow of the following network such that the capacities of the pipes in the network are satisfied.

![Network Flow Diagram]

```
network_flow(F,F12,F13,F24,F25,F32,F35,F46,F56) :-
generator(F,F12,F13,F24,F25,F32,F35,F46,F56),

generator(F,F12,F13,F24,F25,F32,F36,F46,F66) :-
    choose(O,14,F), choose(O,6,F12), choose(O,8,F13),
    choose(O,6,F24), choose(O,4,F25), choose(O,2,F32),
    choose(O,6,F36), choose(O,8,F46), choose(O,6,F66).

choose(N,M,N) :- N =< M.
choose(N,M,C) :- N < M, N1 is N+1, choose(N1,M,C).

constraints(F,F12,F13,F24,F25,F32,F35,F46,F56) :-
    F12 + F13 =:= F, F12 + F32 =:= F24 + F25,
    F13 =:= F32 + F35, F24 =:= F46,
    F25 + F35 =:= F56, F46 + F56 =:= F.
```
APPENDIX E

NOTATION

\[
\begin{align*}
(V, E, W), \ & 122 & \| \cdot \|_p, \ & 150 \\
Adj_G(v), \ & 23 & d_G(v), \ & 21 \\
C(G, x), \ & 135 & \text{tits}(v), \ & 104 \\
G_1(\omega), \ & 136 & m_a, \ & 63 \\
G - V, \ & 23 & \text{mgv}(t_1, t_2); \ & 25 \\
G_1(\omega), \ & 136 & x_i \leftarrow d_j, \ & 120 \\
K(G, n), \ & 122 & B_{\{i\}}, \ & 102 \\
L(G, x), \ & 140 & G, \ & 85 \\
N_G(v), \ & 23 & \mathcal{H}, \ & 24 \\
U(G, x), \ & 140 & \mathcal{N}, \ & 42 \\
W(e), \ & 122 & \mathcal{N}_{\perp, \infty}, \ & 47 \\
\Delta, \ & 143 & \mathcal{N}_{\perp}, \ & 42 \\
\Theta_G, \ & 63 & \mathcal{T}, \ & 24 \\
\Theta_G(\theta, i), \ & 63 & \text{diff}_m, \ & 58 \\
\bot, \ & 42 & \text{instance}(t), \ & 104 \\
\chi(G), \ & 135 & \text{preds}(b), \ & 65 \\
\infty, \ & 36 & \text{size}_m, \ & 58 \\
\emptyset, \ & 123 & \text{vars}(\bar{t}), \ & 104 \\
\rho(G), \ & 138 & \text{DSol}_p, \ & 104 \\
\tau, \ & 42, \ & 148 & \text{Rel}_p, \ & 104 \\
\mathcal{O}(G, x), \ & 140 & \text{Sol}_p, \ & 83 \\
\ominus a, \ & 62 & \mathcal{S}_p^{(i)}, \ & 58 \\
|G|, \ & 21 & \mathcal{T}_p, \ & 147 \\
\cdot |, \ & 42 & \text{soln}(s, d), \ & 166 \\
\cdot |_{m}, \ & 44 & \mathcal{I}_{C, M}, \ & 154 \\
\cdot |_{r}, \ & 42 & \text{sol}_i, \ & 85 \\
\| \cdot \|_P, \ & 85 & \text{sz}(@b), \ & 63
\end{align*}
\]
$t_C$, 147
$t_C$, 147
$t_M$, 154
APPENDIX F

INDEX

\begin{itemize}
  \item \textit{k}-consistent, 128
  \item n-clique, 23
  \item n-partite graph, 23
  \item 2-consistent, 128
  \item abstract interpretation, 16
  \item accumulating subgraphs, 136
  \item acyclically well-moded, 40
  \item adjacency graph, 23
  \item all-solutions predicate, 109
  \item argument dependency graph, 53
  \item argument-size complexity function, 58
  \item arity of a symbol, 24
  \item assignment memorization, 128
  \item assignment set, 121
  \item atom, 24
  \item attributed probabilistic grammars, 15
  \item backtrack, 30
  \item backtracking, 17
  \item binding, 26
  \item binding pattern, 104
  \item Bondy number, 138
  \item boolean satisfiability problem, 184
  \item branch, 23
  \item call, 26
  \item call graph, 49
  \item CASLOG, 36
  \item chromatic number, 135
  \item chromatic polynomial, 135
  \item clause, 24
  \item clause body, 24
  \item clause head, 24
  \item clique, 23
  \item combinatorial optimization problems, 184
  \item complementary tests, 51
  \item complete n-partite graph, 23
  \item complexity analysis, 13
  \item component graph, 123
  \item compound term, 24
  \item computation, 25
  \item computation rule, 26
  \item computational complexity, 13
  \item connected graph, 21
  \item consistency graph, 120
  \item consistent, 120
  \item constant, 24
  \item constraint graph, 134
  \item constraint satisfaction problems, 184
  \item cycle, 21
  \item DAG, 23
  \item definite clause, 24
  \item definite mode, 33
\end{itemize}
degree of a vertex, 21
dependent argument position, 53
determinate mode, 32
determinate predicate, 83
difference equation schema, 166
directed acyclic graph, 23
directed graph, 21
disjunctive constraints, 186
domain, 47
domain declaration, 48
domain expression, 48
dominate, 112
degree of a vertex, 21
dependent argument position, 53
determinate mode, 32
determinate predicate, 83
difference equation schema, 166
directed acyclic graph, 23
directed graph, 21
disjunctive constraints, 186
domain, 47
domain declaration, 48
domain expression, 48
dominate, 112
degree of a vertex, 21
dependent argument position, 53
determinate mode, 32
determinate predicate, 83
difference equation schema, 166
directed acyclic graph, 23
directed graph, 21
disjunctive constraints, 186
domain, 47
domain declaration, 48
domain expression, 48
dominate, 112
degree of a vertex, 21
dependent argument position, 53
determinate mode, 32
determinate predicate, 83
difference equation schema, 166
directed acyclic graph, 23
directed graph, 21
disjunctive constraints, 186
domain, 47
domain declaration, 48
domain expression, 48
dominate, 112

Horn clause, 24
implicit failure, 93
indefinite mode, 33
indeterminate mode, 32
indeterminate predicate, 83
inductive assertions, 14
infinite branch, 23
input position, 31
instance function, 104
interfacing subgraphs, 136

leaf, 23
literal, 24
literal dependency graph, 56
logic program, 24
loop unrolling, 182
measure declaration, 43
mode, 32, 39
mode declaration, 39
more general substitution, 25
most general unifier, 25
mutually exclusive clauses, 51
mutually exclusive cluster, 52

negative literal, 24
neighbor, 23
network consistency, 128
nondecreasing-composed, 166
nondeterminism, 17
nondeterministic, 17, 32
nondirectional, 31

nonground, 32
nonrecurrent expression, 164
nonrecurrent factor, 164
nonrecurrent term, 164
normal form expression, 164
number of solutions, 17
number-of-solutions complexity function, 83

order, 120
order of graph, 21
output position, 31

path, 21
perfect elimination ordering, 138
perfect-smallest-last ordering, 140
positive literal, 24
precedence constraints, 186
predecessor, 21
predicate definition, 24
predicate symbol, 24
principal functor, 24

random sampling, 17
recurrent expression, 164
recurrent factor, 165
recurrent term, 164
recursion induction principle, 16
relation, 17
relation size, 101
relatively nondecreasing, 166
removal of vertices, 23
resolution step, 26
resolved atom, 26

root, 23
rooted tree, 23

search rule, 30
search tree, 27
size measure, 42
SLD-derivation, 25
SLD-resolution, 25
SLD-tree, 27

solution function, 166
solution measure, 85
solution size, 101

sound transformation rule, 166
strongly connected components graph, 50
strongly connected graph, 23
subgraph isomorphism problem, 184
substitution, 25
succeed, 26
success set, 26
successor, 21

term, 24
term norm, 42
test, 51
time complexity function, 147
time complexity metric, 148
time measure, 150
topological order, 51
transformation of variables, 170
transformation rule, 166
triangulated graph, 139
type, 42
type graph, 42
type norm, 42
undirected graph, 21
unifiable, 25
unification, 31
unifier, 25
variable, 24
vertex, 21
weight, 122
weighted adjacency graph, 124
weighted consistency graph, 122
REFERENCES


S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs," ACM Transactions on Programming Languages and Systems 11, 3 (July 1989), pp. 419–450.

Y. Deville, personal communication, March 1990.


