Cost Analysis of Logic Programs *

Saumya K. Debray   Nai-Wei Lin
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

August 10, 1995

Abstract

Cost analysis of programs has been studied in the context of imperative and functional programming languages. For logic programs, the problem is complicated by the fact that programs may be nondeterministic and produce multiple solutions. A related problem is that because failure of execution is not an abnormal situation, it is possible to write programs where implicit failures have to be dealt with explicitly in order to get meaningful results. This paper addresses these problems and develops a method for (semi-)automatic analysis of the worst-case cost of a large class of logic programs. The primary contribution of this paper is the development of techniques to deal with nondeterminism and the generation of multiple solutions via backtracking. Applications include program transformation and synthesis, software engineering, and in parallelizing compilers.

Categories and Subject Descriptors: D.1 [**Software**]: Programming Techniques; D.1.6 [**Programming Techniques**]: Logic Programming; D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications—*nonprocedural languages*; I.2 [**Computing Methodologies**]: Artificial Intelligence; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*automatic analysis of algorithms.*

General Terms: Languages, Performance

Additional Key Words and Phrases: Program Analysis, Complexity, PROLOG.

*A preliminary version of this paper appeared in *Proceedings of the Eighth International Conference on Logic Programming*, Paris, June 1991. This work was supported in part by the National Science Foundation under grant number CCR-8901283.

1 Introduction

(Semi-)Automatic cost analysis of programs has been widely studied in the context of functional languages. A major difference between logic programs and functional programs in this regard is that logic programs are nondeterministic in general, and may produce multiple solutions. A related problem is that because failure of execution is not an abnormal situation, it is possible to write programs where implicit failures have to be accounted for and dealt with explicitly if meaningful results are to be obtained. Because of such behavior, the details of low-level control flow are significantly more complex in logic programs, compared to programs in functional or imperative languages. Because of this, cost analysis for logic programs is considerably harder than for programs in more traditional languages. The primary contribution of this paper is to show how nondeterminism, and the generation of multiple solutions via backtracking, can be handled within a uniform framework for cost analysis. In particular, we show how properties of unification may be exploited in the treatment of comparison operators ($=, \neq, >, \geq, <, \leq$) to improve the analysis of nondeterministic predicates.

In principle, the cost of a procedure depends on some measure of the input size. Therefore, it is necessary to keep track of the sizes of arguments to procedures at each program point (procedure entry and exit). In addition, in order to handle nondeterministic procedures, knowledge about the number of solutions generated by each predicate is required. In this paper, size relationships between arguments and the number of solutions each predicate can generate are inferred using data dependency information.

Not unexpectedly, the size relationships between arguments, the number of solutions and the time complexity functions for recursive procedures are obtained in the form of difference equations. To get closed form expressions, these difference equations need to be solved. The automatic solution of general difference equations is a difficult problem, but there is a wide class of programs for which the difference equations can be solved automatically [5, 16, 31]. Our approach consists of the following steps:

1. Use data dependency information to compute the relative sizes of variable bindings at different program points. This size information can be used to determine the space requirements of variable bindings, which in turn can be used to compute the space complexity;
2. use the size information to compute the number of solutions generated by each procedure;
3. use the size and the number of solutions information to compute the time complexity.

Automatic cost analysis of programs has applications in many areas. In program transformation and automatic program synthesis, it can be used as a criterion to choose among several possible alternatives [6, 22, 27]. In software engineering, programmers can employ it to understand program behavior. In compilers for parallel systems, knowledge about the cost of different procedures can be used 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 [9, 35]. Information about the number of solutions generated by different procedures can be used to improve the performance of deductive database programs, e.g., to plan the order in which subgoals are evaluated [10]. In addition, knowledge about the size relationships between arguments is important for reasoning about program termination [32, 38, 40].

The remainder of this paper is organized as follows: Section 2 introduces some preliminary notions on the subject. Section 3 presents the method for the inference of argument size relationships. Section

4 describes the analysis for estimating the number of solutions generated by each procedure. Section 5 shows the scheme for the composition of time complexity functions. Section 6 describes a mechanism for obtaining (approximate) solutions for difference equations. Section 7 shows the organization of a prototype implementation. Section 8 sketchs a soundness proof of our method. Section 9 illustrates an application of automatic cost analysis: task granularity analysis for parallel logic programs. Finally, Section 10 discusses some related work, and Section 11 gives conclusions.

2 Preliminaries

Most logic programming languages are based on a subset of the 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 being mutually disjoint. Without loss of generality, we assume that with each function symbol f and each predicate symbol p is associated 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 either a variable, or a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A literal is either an atom $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, 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 predicate definition consists of a finite number of definite clauses, all whose heads have the same predicate symbol; a goal is a set of negative literals. A logic program consists of a finite set of predicate definitions. We adhere to the syntax of Edinburgh Prolog and write a definite clause as

$$p \ :- \ q_1, \ldots, q_n$$

read declaratively as "p *if* q_1 *and* ... *and* q_n". Names of variables begin with upper case letters, while names of non-variable (i.e. function and predicate) symbols begin with lower case letters.

We assume that each argument position of a predicate is annotated as an *input* or *output* position, depending on whether or not it is bound to a term when that predicate is invoked.[1] In this paper we consider well-moded clauses with ground bindings. A clause is said to be *well-moded* if

1. every variable appearing in an input position in a body literal also appears either in an input position in the head of the clause, or in an output position of some other body literal; and
2. every variable occurring in an output position in the head of the clause also appears either in an input position in the head, or in an output position in the body.

The intuition is that the binding for any variable in the clause is either a term given as an input argument, or a term produced as an output argument by a body literal. A term is said to be *ground* if it contains no variable. A clause with *ground bindings* demands that all input arguments are bound to ground terms on invocation and all output arguments are bound to ground terms on success. Strictly speaking, this groundness requirement can be relaxed as long as no predicate binds any variable occurring in any of its input argument positions: as an example, consider the familiar **append** program for

[1] The input/output character of argument positions can be inferred via dataflow analysis [7, 28].

concatenating lists, which does not require that the elements of the lists being processed be ground. While it is possible to give syntactic characterizations that imply this property, however, such characterizations, in attempting to cope with aliasing effects, quickly become verbose and cumbersome while shedding little light on the essential property they are attempting to characterize. For this reason, we shall consider only programs with ground bindings in this paper, with the understanding that the notion can be appropriately generalized where necessary.

The *call graph* for a program is a directed graph which represents the caller-callee relationships between predicates in the program. Each node in the graph denotes a predicate in the program. There is an edge from a node p_1 to a node p_2 if a literal with predicate symbol p_2 appears in the body of a clause defining the predicate p_1. A body literal in a clause is called a *recursive* literal if it is part of a cycle that contains the head of that clause in the call graph for the program. A clause is called *nonrecursive* if no body literal is recursive, and is called *direct recursive* if it contains recursive literals and all the recursive literals have the same predicate symbol as the head; otherwise, it is called *indirect recursive*. A clause is *recursive* if it is either direct or indirect recursive.

Operationally, given an output position a_1 and an input position a_2 in a clause, a_2 is *dependent* on a_1 if the variable bindings generated at a_1 are used to construct the term occurring at a_2, i.e., if the terms occurring at positions a_1 and a_2 have variables in common. The data dependencies between argument positions can be represented by a directed acyclic graph $G = (V, E)$, called an *argument dependency graph*, where V is a set of vertices and E a set of edges. Each vertex in the graph denotes an argument position. There is an edge $\langle a_1, a_2 \rangle$ from an argument position a_1 to an argument position a_2 if a_2 is dependent on a_1: in this case, a_1 is said to be a *predecessor* of a_2, and a_2 a *successor* of a_1. Note 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. A *path* in the graph is a sequence of vertices $v_1, \ldots, v_n$ such that $\langle v_i, v_{i+1} \rangle$ is an edge in the graph, for $1 \leq i < n$. Argument dependency graphs are induced by the control strategy of the system, and may be inferred via dataflow analysis [3, 7].

It is sometimes convenient to abstract an argument dependency graph into a graph that represents the data dependencies between literals. A *literal dependency graph* is a directed acyclic graph. Each vertex in the graph denotes a literal and consists of the set of vertices in argument dependency graph that correspond to the argument positions in the denoted literal. There is an edge between two vertices in literal dependency graph if there exists at least one edge between the two corresponding sets of vertices in argument dependency graph. The head of the clause is treated specially. It is divided into two vertices, one consists of the input positions in the head, and the other consists of the output positions in the head. Paths in literal dependency graph are defined in the same way as in argument dependency graph. Hereafter we assume that the argument dependency graphs and literal dependency graphs are given.

Example 2.1 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, [R|Rs]) :- select(R, X, Y), perm(Y, Rs).

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

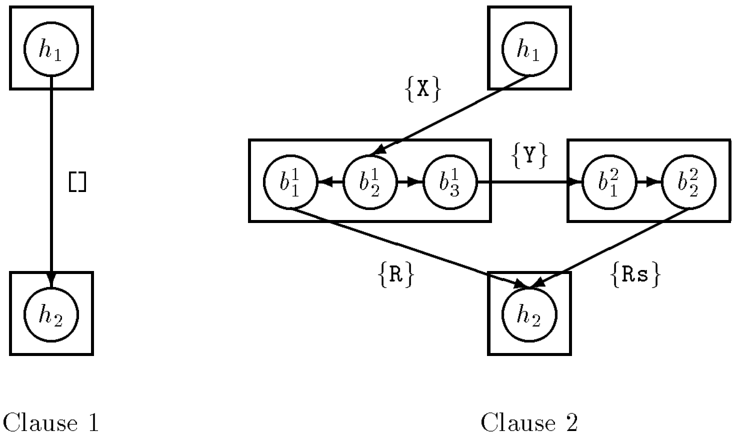

Figure 1: Argument dependency graphs and literal dependency graphs for the clauses of predicate `perm`.

Given a list `X` as its first (input) argument, `perm/2` generates a permutation of `X` by nondeterministically selecting an element `R` of `X`, permuting the remainder `Y` of the input list into a list `Rs`, and returning the list `[R|Rs]` as its second (output) argument. The argument dependency graphs and literal dependency graphs for the clauses of the predicate `perm/2` are shown in Figure 1, where h_i denotes the i^{th} argument position in the head, while b_i^j denotes the i^{th} argument position in the j^{th} body literal. The circles represent the vertices in the argument dependency graphs, while rectangles represent vertices in the literal dependency graph. □

Because logic programs can be nondeterministic, there may in general be more than one clause whose head unifies with a call. The results of cost analysis can be greatly improved if situations where this cannot happen, i.e., 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 detection of mutual exclusion between clauses is discussed in [8]. The clauses of a predicate can be partitioned into a set of mutually exclusive "clusters" such that two clauses are in the same cluster if and only if they are not mutually exclusive. Then, cost analysis can be performed separately for each cluster, with the total cost for the predicate being given in terms of the cost of the most expensive cluster. In this paper, we assume that the partitioning of clauses into mutually exclusive clusters has been carried out (for details, see [8]).

3 Argument Size Relations: Space Complexity

This section presents a method for the inference of argument size relations based on data dependency information. The purpose of this inference is to represent the input size to each body literal as a function in terms of the input size to the head. The input size to body literals would be used later to infer the cost of body literals. We will first discuss notions related to various "size measures" for terms. This is followed by a discussion of how (based on the data dependency in a clause) size relationships between each argument position and its predecessors may be inferred using appropriate size measures. We then describe how these size relationships can be propagated so that the size relation corresponding to an input position in a literal can be transformed into a function in terms of the size of the input positions in the head. Finally, a characterization of conditions under which such size functions are well-defined is given.

3.1 Size Measures

Various measures can be used to determine the "size" of an input, e.g., term-size, term-depth, list-length, integer-value, etc. The measure(s) appropriate in a given situation can in most cases be determined by examining the types of argument positions, the general idea being to use the "back edges" in the type graph of a predicate to determine how that predicate recursively traverses its input terms (or constructs its output terms), and thereby synthesize a measure for the predicate [32, 40]. Type information may be inferred via program analysis [13, 29, 36, 42], and is not discussed further here.

We first discuss how to determine the size of ground terms. Let $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ be a function that maps ground terms to their sizes under a specific measure m, where $\mathcal{H}$ is the Herbrand universe, i.e. the set of ground terms of the language, and $\mathcal{N}_\perp$ the set of natural numbers augmented with a special symbol $\perp$, denoting "undefined". Examples of such functions are "list_length", which maps ground lists to their lengths and all other ground terms to $\perp$; "term_size", which maps every ground term to the number of constants and function symbols appearing in it; "term_depth", which maps each ground term to the height of its tree representation; and so on. Thus, $|[\mathtt{a},\mathtt{b}]|_{\text{list\_length}} = 2$, $|\mathtt{f(1,f(2,nil,nil),nil)}|_{\text{term\_depth}} = 2$, but $|\mathtt{f(a)}|_{\text{list\_length}} = \perp$.

Based on $|\cdot|_m$, the size properties of general terms can be described using two functions $size_m$ and $diff_m$. the function $size_m(t)$ defines the size of a term t under a measure m:

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

Thus, $size_{\text{list\_length}}([\mathtt{L},\mathtt{a}]) = 2$, and $size_{\text{list\_length}}([\mathtt{a}|\mathtt{L}]) = \perp$. A detailed realization of the *size* function for some commonly encountered measures is given in Figure 2. The function $diff_m(t_1, t_2)$ gives the size difference between two terms t_1 and t_2 under a measure m:

$$diff_m(t_1,t_2) = \begin{cases} d & \text{if } t_2 \text{ is a subterm of } t_1 \text{ and } |\theta(t_2)|_m - |\theta(t_1)|_m \le d \text{ for every} \\ & \quad \text{substitution } \theta \text{ such that } \theta(t_1) \text{ and } \theta(t_2) \text{ are ground} \\ \perp & \text{otherwise.} \end{cases}$$

Thus, $diff_{\text{list\_length}}([\mathtt{a},\mathtt{b}|\mathtt{L}],\mathtt{L}) = -2$, $diff_{\text{term\_depth}}(\mathtt{f(1,X,Y)},\mathtt{X}) = -1$, and $diff_{\text{term\_size}}(\mathtt{X},\mathtt{f(X)}) = \perp$. A detailed realization of the *diff* function for some commonly encountered measures is given in Figure 3. Where the particular measure under consideration is clear from the context in the discussion that follows, we will omit the subscript in the *size* and *diff* functions.

3.2 Size Relations

We now show how *size* and *diff* functions can be used to extract size relationships between each argument position and its predecessors. We use the notation $@a$ to denote the term occurring at an argument position a, m_a to denote the size measure associated with a, and $\mathtt{sz}(@a)$ to denote the size of the term occurring at argument position a. Further, let $\mathsf{Sz}_p^b : \mathcal{N}_{\perp,\infty}^n \rightarrow \mathcal{N}_{\perp,\infty}$ be a function that represents the size of the b^{th} (output) argument position in a predicate p, which has n input argument positions, in

If m is integer_value, then

$$size_m(t) = \begin{cases} n & \text{if } t \text{ is an integer } n \\ \odot(size_m(t_1), \ldots, size_m(t_n)) & \text{if } t = \odot(t_1, \ldots, t_n) \text{ for some evaluable arithmetic functor } \odot \\ \perp & \text{otherwise} \end{cases}$$

If m is list_length, then

$$size_m(t) = \begin{cases} 0 & \text{if } t \text{ is the empty list} \\ 1 + size_m(t_1) & \text{if } t \text{ is of the form } [\_|t_1] \text{ for some term } t_1 \\ \perp & \text{otherwise} \end{cases}$$

If m is term_depth, then

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

If m is term_size, then

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

Figure 2: The function $size_m(t)$ for some common size measures

If m is integer_value, then

$$diff_m(t1, t2) = \begin{cases} 0 & \text{if } t1 \equiv t2 \\ \perp & \text{otherwise} \end{cases}$$

If m is list_length, then

$$diff_m(t1, t2) = \begin{cases} 0 & \text{if } t1 \equiv t2 \\ diff_m(t, t2) - 1 & \text{if } t1 \text{ is of the form } [\_|t] \text{ for some term } t \\ \perp & \text{otherwise} \end{cases}$$

If m is term_depth, then

$$diff_m(t1, t2) = \begin{cases} 0 & \text{if } t1 \equiv t2 \\ max\{diff_m(t_i, t2) \mid 1 \leq i \leq n\} - 1 & \text{if } t1 = f(t_1, \ldots, t_n) \\ \perp & \text{otherwise} \end{cases}$$

If m is term_size, then

$$diff_m(t1, t2) = \begin{cases} 0 & \text{if } t1 \equiv t2 \\ \mathsf{arg}(t1, i) - size_m(t1) & \text{if } t1 = f(t_1, \ldots, t_n) \text{ and } t_i \equiv t2 \text{ for some } i,\ 1 \leq i \leq n \\ \perp & \text{otherwise} \end{cases}$$

where $\mathsf{arg}(t1, i)$ is a symbolic expression denoting the term-size of the i^{th} argument of the term $t1$.

Figure 3: The function $diff_m(t1, t2)$ for some common size measures m

terms of the size of its input positions, where $\mathcal{N}_{\perp,\infty}$ denotes the set of natural numbers plus two special symbols $\perp$ and ∞, denoting undefined and infinite respectively.

We first consider the argument positions in a body literal. Let L be a body literal in a clause, with input argument positions $a_1, \ldots, a_n$. An argument position in L is either an input or an output position. First, consider an output position b in L. There are two possibilities:

(1) If L is recursive, then $\mathtt{sz}(@b)$ is symbolically expressed as

$$\mathtt{sz}(@b) \leq \mathsf{Sz}_L^{(b)}(\mathtt{sz}(@a_1), \ldots, \mathtt{sz}(@a_n)).$$

(2) If L is not recursive, assume the argument size function for argument position b in literal L have been recursively computed as $\mathsf{Sz}_L^{(b)}(x_1, \ldots, x_n)$, then $\mathtt{sz}(@b)$ can be expressed as

$$\mathtt{sz}(@b) \leq \mathsf{Sz}_L^{(b)}(\mathtt{sz}(@a_1), \ldots, \mathtt{sz}(@a_n)).$$

Note that since the function giving the size relationship between the input and output positions of L has already been computed recursively, we are able, in this case, to express the relationship between $\mathtt{sz}(@b)$ and $\{\mathtt{sz}(@a_1), \ldots, \mathtt{sz}(@a_n)\}$ explicitly in terms of this function.

Next, consider an input position a in literal L, and let $\mathit{preds}(a)$ denote the set of predecessors of a. The size of the term occurring at a can be determined as follows:

(1) If $\mathit{size}_{m_a}(@a) \neq \perp$, then $\mathtt{sz}(@a) \leq \mathit{size}_{m_a}(@a)$;

(2) otherwise, if $m_c = m_a$ and $\delta = \mathit{diff}_{m_a}(@c, @a) \neq \perp$ for some argument position $c \in \mathit{preds}(a)$, then $\mathtt{sz}(@a) \leq \mathtt{sz}(@c) + \delta$;

(3) otherwise, if $\mathit{size}_{m_a}(@a)$ can be expanded using the definition in Figure 2, then:

   (a) expand $\mathit{size}_{m_a}(@a)$ one step accordingly;

   (b) recursively compute $\mathit{size}_{m_a}(t_i)$ for the appropriate subterms t_i (depending on the size measure involved) of $@a$ with respect to the same set of predecessors $\mathit{preds}(a)$;

   (c) if each of these recursive size computations has a defined result, then use them to compute $\mathtt{sz}(@a)$ as appropriate (depending on the particular size measure under consideration); if the result of any of the recursive size computations is undefined, $\mathtt{sz}(@a) = \perp$.

(4) otherwise, $\mathtt{sz}(@a) = \perp$.

We now use $\mathit{subterms}(a)$ to denote the set of subterms t occurring in $@a$ that are used in the step (2) such that either $\mathit{diff}_{m_a}(@c, t) \neq \perp$ for some argument position $c \in \mathit{preds}(a)$, or $\mathit{diff}_{m_a}(@c, t) = \perp$ for all argument position $c \in \mathit{preds}(a)$ and $\mathit{size}_{m_a}(t)$ cannot be further expanded using the definition in Figure 2 in step (3). In other words, $\mathit{subterms}(a)$ contains the set of subterms occurring in $@a$ whose sizes are required in order to determine the size of $@a$. We also use $\mathit{def}(t)$ to denote the (output) argument position where a term $t \in \mathit{subterms}(a)$ becomes bound. The argument positions and literals where a variable becomes bound are easily available from the argument and literal dependency graphs.

Example 3.1 Consider the clause

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

We use the following notation to refer to the size of the term occurring in an argument position in a clause: $head[i]$ denotes the size of the i^{th} argument position in the head, while $body_j[i]$ denotes the size of the i^{th} argument position in the j^{th} body literal. Assume that for nrev/2, the first argument is an input position while the second is an output position; and for append/3, the first two argument positions are input positions while the third argument is an output position. Further, assume that the size measure under consideration is list_length. Then, the size relations for the body literals are given by

$$
\begin{aligned}
&body_1[1] \le size(\mathtt{L}) = head[1] + \mathit{diff}([\mathtt{H}|\mathtt{L}], \mathtt{L}) = head[1] - 1,\\
&body_1[2] \le \mathsf{Sz}^{(2)}_{\mathtt{nrev}}(body_1[1]),\\
&body_2[1] \le size(\mathtt{R1}) = body_1[2] + \mathit{diff}(\mathtt{R1}, \mathtt{R1}) = body_1[2],\\
&body_2[2] \le size([\mathtt{H}]) = 1,\\
&body_2[3] \le \mathsf{Sz}^{(3)}_{\mathtt{append}}(body_2[1], body_2[2]).
\end{aligned}
$$

Let a be the first argument position of literal nrev/2. Then $subterms(a) = \{\mathtt{L}\}$, and $def(\mathtt{L})$ is the first argument position in the head. □

Similarly, a set of size relations can be obtained for output argument positions in the head.

Example 3.2 Continuing the previous example, the size relations for the output position in the head is

$$head[2] \le size(\mathtt{R}) = body_2[3] + \mathit{diff}(\mathtt{R}, \mathtt{R}) = body_2[3].$$

□

3.3 Size Functions

We now show how the size relations can be propagated to transform a size relation corresponding to an input position in a literal into a function in terms of the input size to the head. However, for recursive clauses, we need to solve the symbolic expression due to recursive literals into an explicit function first. We can then use the explicit function to infer the input size to the literals that succeed the recursive ones.

Given the size relations for the body literals of a clause, it is possible to transform the size relations for the output argument positions in the head into functions in terms of the sizes of the input argument positions in the head. The basic idea here is to repeatedly substitute size relations for body literals "into" size relations for head arguments: given size relations $R_1 \equiv$ '$\varphi \le \psi$' for a body literal, and $R_2 \equiv$ '$\xi \le E(\varphi)$' for some output argument position in the head, where $E(\varphi)$ is some expression involving φ, the substitution of R_1 into R_2 yields the size relation '$\xi \le E(\psi)$'. The process of repeatedly subsituting size relations for body literals into those for output positions in the head is called *normalization*. An algorithm for realizing normalization is given in Figure 4.

Let $\mathcal{E}_B$ be the set of size relations for body literals, and h be the size relation for an output position in the head or an input position in a body literal. The algorithm proceeds as follows:

begin
  repeat
    if there is at least one occurrence of a term t in the RHS of h
      and t is the LHS of a relation $b \in \mathcal{E}_B$
    then replace each occurrence of t in the RHS of h by the RHS of b
  until there is no change
end

Figure 4: An algorithm for normalization

Example 3.3 Consider the predicate `perm/2` defined in Example 2.1. 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. Assume that the size relations for the output argument positions of the predicate `select/3` have been recursively computed as $\mathsf{Sz}^{(1)}_{\mathtt{select}} \equiv \lambda x.\bot$ and $\mathsf{Sz}^{(3)}_{\mathtt{select}} \equiv \lambda x.x-1$ (see the Appendix A for details). Using list_length as the size measure, the size relations for the body literals in the recursive clause are

$$
\begin{aligned}
&body_1[1] \le \mathsf{Sz}^{(1)}_{\mathtt{select}}(body_1[2]) = \bot,\\
&body_1[2] \le size(\mathtt{X}) = head[1] + diff(\mathtt{X},\mathtt{X}) = head[1],\\
&body_1[3] \le \mathsf{Sz}^{(3)}_{\mathtt{select}}(body_1[2]) = body_1[2] - 1,\\
&body_2[1] \le size(\mathtt{Y}) = body_1[3] + diff(\mathtt{Y},\mathtt{Y}) = body_1[3],\\
&body_2[2] \le \mathsf{Sz}^{(2)}_{\mathtt{perm}}(body_2[1]),
\end{aligned}
$$

and the size relation for the output argument position in the head is

$$
\begin{aligned}
head[2] &\le size([\mathtt{R}|\mathtt{Rs}]) = size(\mathtt{Rs}) + 1\\
&= body_2[2] + diff(\mathtt{Rs},\mathtt{Rs}) + 1\\
&= body_2[2] + 1.
\end{aligned}
$$

On normalization, this gives

$$
\begin{aligned}
head[2] &\le \mathsf{Sz}^{(2)}_{\mathtt{perm}}(body_2[1]) + 1\\
&\le \mathsf{Sz}^{(2)}_{\mathtt{perm}}(body_1[3]) + 1\\
&\le \mathsf{Sz}^{(2)}_{\mathtt{perm}}(body_1[2] - 1) + 1\\
&\le \mathsf{Sz}^{(2)}_{\mathtt{perm}}(head[1] - 1) + 1.
\end{aligned}
$$

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

$$\mathsf{Sz}^{(2)}_{\mathtt{perm}}(head[1]) = \mathsf{Sz}^{(2)}_{\mathtt{perm}}(head[1] - 1) + 1.$$

In addition, from the first clause, we can obtain the equation $\mathsf{Sz}^{(2)}_{\mathtt{perm}}(0) = 0$ as the boundary condition. They can be solved to yield $\mathsf{Sz}^{(2)}_{\mathtt{perm}} \equiv \lambda x.x$, i.e. the size of the output of `perm/2` is bounded by the size of its input. □

Recall that in a recursive clause, the size of the output arguments of recursive literals in the body are expressed symbolically in terms of its input sizes. Once the size functions for the output positions in the head have been determined, they can be substituted for these symbolic expressions in the set of size relations for the body literals. We can then apply normalization again to compute the size of each input position for each body literal, so that it is defined in terms of the size of the input arguments in the head of the clause. These size functions can be used later in computing the number of solutions and the time complexity of the clause.

Example 3.4 Consider the clause defined in Example 3.1. Suppose the argument size function for the output argument position of `nrev/2` has been computed as $\mathsf{Sz}^{(2)}_{\mathtt{nrev}} \equiv \lambda x.x$. Then the size for the first (input) argument position of literal `append/3` can be obtained as

$$\begin{aligned} body_2[1] &\leq body_1[2] \\ &\leq \mathsf{Sz}^{(2)}_{\mathtt{nrev}}(body_1[1]) \\ &\leq body_1[1] \\ &\leq head[1] - 1. \end{aligned}$$

□

Theorem 3.1 *Normalization of size relations terminates for all clauses.*

Proof At each iteration of the normalization algorithm, only finitely many substitutions are made. The number of iterations of the algorithm is bounded by the height of the argument dependency graph. □

3.4 Well-connectedness

We now give a characterization of conditions under which the inferred argument size functions are well-defined. Let C be a well-moded clause with the input argument positions $\alpha_1, \ldots, \alpha_n$ in the head. Suppose further that C is not indirect recursive. Let A be the set of output positions in the head and input positions in the body literals. A clause C is said to be *well-connected* if

1. for each argument position $\gamma \in A$, and for each term $t \in subterms(\gamma)$, the following hold:
   - (i) $\mathsf{Sz}_l^{(def(t))}$ is defined if $def(t)$ occurs in a nonrecursive literal l;
   - (ii) $m_{def(t)} = m_\gamma$; and
   - (iii) $\mathit{diff}(T_{def(t)}, t)$ is defined.
2. For each recursive literal L with input argument positions $\beta_1, \ldots, \beta_n$, the following hold:
   - (i) there is no recursive literal on the path from the input of the head to L in the literal dependency graph of C;
   - (ii) $\mathtt{sz}(@\beta_i) \leq \mathtt{sz}(@\alpha_i)$ for all i, $1 \leq i \leq n$; and
   - (iii) $\mathtt{sz}(@\beta_i) < \mathtt{sz}(@\alpha_i)$, for some i, $1 \leq i \leq n$.

Intuitively, condition (1) guarantees that all the argument sizes are functions defined in terms of the input sizes, while condition (2) ensures that the functions are in the form of difference equations for recursive clauses. The notion of well-connectedness can be extended in a straightforward way to deal with indirect recursion.

Example 3.5 We give two examples of non-well-connected clauses. First, consider the program:

```
sum([], 0).
sum([H|T], S) :- sum(T, S1), S is H + S1.
```

which computes the sum of a list of numbers. Suppose the first argument is the input, and the size measures under consideration are list_length for the first argument position and integer_value for the second argument position. Since the size of `H` for the literal `is`/2 cannot be extracted from the term `[H|T]`, the size of the output `S` cannot be expressed as a function in terms of the size of the input `[H|T]`. Therefore, if the size measure under consideration is not fine enough to capture the size relationships between argument positions, the clauses are usually non-well-connected. However, notice that though the argument size function is undefined for `sum`/2, because the size of the input to the recursive literal `sum`/2, i.e., `T`, can be computed and the time complexity for the builtin predicate `is`/2 is known in advance, it is still possible to estimate the time complexity for `sum`/2. Next, consider the program:

```
rev([], L, L).
rev([H|T], L, R) :- rev(T, [H|L], R).
```

which reverses a list of elements. Suppose the first and second arguments are input and list_length is used as the size measure under consideration. Since the size of `[H|L]` is larger than the size of `L` in the recursive clause, the resulting size function for the output argument position of `rev`/3 is not in the form of difference equation. Thus, in general, clauses using accumulator style programming are non-well-connected. □

The analysis for argument size relations based on normalization is applicable to well-connected clauses:

Theorem 3.2 *If a clause C is well-connected, then the following hold after the size relations that hold in C are normalized:*

1. *If C is nonrecursive, then the sizes for the output argument positions in the head of C are obtained as a closed form function of is the sizes of the input argument positions in the head of C;*
2. *if C is direct recursive, then the sizes for the output argument positions in the head of C are obtained as a difference equation in terms of the sizes of the input argument positions in the head of C; and*
3. *if C is indirect recursive, then the sizes for the output argument positions in the head of C are obtained as a difference equation, which is part of a system of difference equations for mutually recursive clauses, in terms of the sizes of the input argument positions in the head of C.*

Proof By induction on the number of literals in the body of C. $\square$

The preceding discussion has shown how to infer the argument size functions for a clause. The argument size functions for a predicate can then be obtained by taking the maximum among the expressions for the argument size functions obtained for each of its clauses.

With knowledge about the size relationships between argument positions in the clauses, given the size of input, the space required by each argument position can be estimated. This information can then be used to compute the space required by a predicate in a specific computational model and implementation.

4 Number of Solutions

This section describes the analysis for estimating the number of solutions generated by a predicate. This can be characterized in terms of two parameters: the *relation size* of the predicate, which usually does not depend on the input; and the *solution size*, i.e., the maximum number of outputs that can be generated by a single input to the predicate, which usually depends on the size of the input. Although, among them, only solution size information is used to compute the cost of a predicate, relation size information can greatly improve the estimation of solution size. We first present a general algorithm, based on the properties of unification, for estimating these two parameters for each predicate. Then we give two algorithms that estimate the relation size for two special classes of predicates based on the properties of comparison operators ($=, \neq, >, \geq, <, \leq$) and information about argument types. The predicates in the first class are the ones that can be "unfolded" into a conjunction and/or disjunction of linear arithmetic constraints; and the predicates in the second class are the ones that can be "unfolded" into a conjunction of binary nonequality constraints. Compared with the general algorithm, the two special algorithms can considerably improve the precision of the analysis for these two classes of predicates. Finally, we show how to combine these algorithms in relation size analysis.

4.1 A General Algorithm

The basic idea behind the algorithm is counting the number of possible bindings for each variable in a clause. Using data dependency information, the number of instances for the term occurring at an argument position is estimated from the number of instances for the terms occurring at its predecessors. Based on properties of unification, the number of bindings for a variable is estimated from the number of instances for the terms in which it appears. We first assume that no duplicate solutions are generated during execution; we will discuss how to deal with duplicates at the end of this subsection.

We use $\mathcal{B}_{\{T\}}$ to denote the number of instances for a (tuple of) term(s) T in a clause. In the discussion of relation size and solution size analyses, we will overload this notation. For a variable V, in relation size analysis, $\mathcal{B}_{\{V\}}$ denotes the number of distinct bindings for V that are generated by all possible inputs to the clause; in solution size analysis, it denotes the number of distinct bindings that are generated by a single input 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 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 invocation to a predicate ensures that different occurrences of a variable in a clause must bind to the same term, the number of bindings for `Y` in this clause should be 2, i.e., the bindings to `a` and `b`. Thus the number of bindings for a variable in the clause is bounded above by the minimum of the number of possible bindings for different occurrences of that variable:

Theorem 4.1 *If a variable X 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 $\mathcal{B}_{\{X\}} \leq min\{k_1, \ldots, k_n\}$.* $\square$

Recall our assumption that the input arguments to any predicate are bound to ground terms when that predicate is called, and its output positions are also bound to ground terms if the call succeeds: Theorem 4.1 may not hold, in general, if nonground bindings are considered [11]. 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`. The total number of instances thus 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 contained 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:

Theorem 4.2 *Let T be a (tuple of) term(s) in a clause. If T contains a set of variables $S = \{X_1, \ldots, X_m\}$, such that $\mathcal{B}_{\{X_i\}} \leq N_{\{X_i\}}$, for $1 \leq i \leq m$, then $\mathcal{B}_{\{T\}} = \mathcal{B}_S \leq \prod_{i=1}^{m} N_{\{X_i\}}$, with $\mathcal{B}_\emptyset = 1$.* $\square$

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, $S = \{X_1, \ldots, X_m\}$, and $N_{\{X_i\}}$ be the determined number of bindings possible for X_i. We define a function, called *instance function* and denoted by $instance(T)$, as follows:[2]

$$instance_1(T) = \prod_{i=1}^{m} N_{\{X_i\}}.$$

Because the *diff* functions defined for size measures are based on structural differences between terms, well-connected recursive clauses usually apply recursion on (terms derived from) subterms of the input arguments. Since the invocation to a well-connected recursive clause may succeed for any instance of the input arguments, the relations they define are, in general, infinite. Thus, it is more desirable, for recursive predicates, to obtain information about the maximum number of outputs that can be generated by a single input. For example, the predicate `select/3` defined in Example 2.1 succeeds for any nonempty input list, so the size of its relation is infinite. However, given a nonempty list of length n as input, `select/3` always generates n outputs. We associate each predicate p in the program with a pair $\langle \mathsf{Rel}_p, \mathsf{Sol}_p \rangle$, called the *binding pattern* of p, where Rel_p denotes an upper bound of the relation size for p, and Sol_p denotes an upper bound of the solution size for p. In general, for a predicate p with n input positions, Rel_p is in $\mathcal{N}_\infty$, namely, a natural number or the symbol ∞, denoting an infinite relation; while $\mathsf{Sol}_p : \mathcal{N}_{\perp,\infty}^n \to \mathcal{N}_\infty$ is a function in terms of the size of the input.

[2] Since we will successively improve the realization of the *instance* function, we use subscripts to distinguish the different versions of this function.

Consider a clause $p(\bar{x}_0, \bar{y}_0)\ :-\ q_1(\bar{x}_1, \bar{y}_1), \ldots, q_n(\bar{x}_n, \bar{y}_n)$, where the body literals are sorted in topological order from the literal dependency graph, and $\bar{x}_i$ are input arguments and $\bar{y}_i$ are output arguments. Let $vars(\bar{t})$ be the set of variables in tuple $\bar{t}$, and $lits(v)$ be the set of literals in which variable v appears. Further, let $\bar{n}_i$ be the input size to literal q_i. Assume that the binding patterns for the nonrecursive body literals have been (recursively) computed and the binding patterns for recursive literals have been represented in symbolic form as a function of input size. The algorithm proceeds as follows:

begin
    `/* compute` Rel_p `*/`
    if p is recursive **then** $\mathsf{Rel}_p := \infty$;
    else do
        for each variable $v \in vars(\bar{x}_0)$ **do**
            $N_{\{v\}} := min\{\mathsf{Rel}_j \mid j \in lits(v)\}$;
        od
        for $i := 1$ **to** n **do**
            $I_i := min\{instance(\bar{x}_i), \mathsf{Rel}_{q_i}\}$;
            $O_i := min\{I_i \times \mathsf{Sol}_{q_i}(\bar{n}_i), \mathsf{Rel}_{q_i}\}$;
            for each variable $v \in vars(\bar{y}_i)$ **do**
                $N_{\{v\}} := min\{O_i, \mathsf{Rel}_j \mid j \in lits(v)\}$;
            od
        od
        $\mathsf{Rel}_p := instance((\bar{x}_0, \bar{y}_0))$;
    od

    `/* compute` Sol_p `*/`
    for each variable $v \in vars(\bar{x}_0)$ **do**
        $N_{\{v\}} := 1$;
    od
    for $i := 1$ **to** n **do**
        $I_i := min\{instance(\bar{x}_i), \mathsf{Rel}_{q_i}\}$;
        $O_i := min\{I_i \times \mathsf{Sol}_{q_i}(\bar{n}_i), \mathsf{Rel}_{q_i}\}$;
        for each variable $v \in vars(\bar{y}_i)$ **do**
            $N_{\{v\}} := min\{O_i, \mathsf{Rel}_j \mid j \in lits(v)\}$;
        od
    od
    $\mathsf{Sol}_p := instance(\bar{y}_0)$;
end

Figure 5: An algorithm for computing binding patterns

Based on Theorems 4.1 and 4.2, we can devise a simple algorithm to compute the binding pattern for each predicate. The algorithm is presented in Figure 5, and can be summarized as follows: consider a clause: '$p(\bar{x}_0, \bar{y}_0)$:− $q_1(\bar{x}_1, \bar{y}_1), \ldots, q_n(\bar{x}_n, \bar{y}_n)$', where the body literals are sorted in topological order from the literal dependency graph, and $\bar{x}_i$ are input arguments and $\bar{y}_i$ output arguments. Let $vars(\bar{t})$ be the set of variables in tuple $\bar{t}$, and $lits(v)$ be the set of literals in which variable v appears. First the binding patterns of its nonrecursive body literals are recursively computed and the binding patterns of its recursive literals are represented in symbolic form. To compute Rel_p, the relation size for p, if the predicate p is recursive, then Rel_p is set to be infinite; otherwise, the number of bindings, $\mathcal{B}_{\{v\}}$, for each variable v in the input arguments $\bar{x}_0$ is estimated using Theorem 4.1:

$$\mathcal{B}_{\{v\}} \leq N_{\{v\}} = min\{\mathsf{Rel}_j \mid j \in lits(v)\}.$$

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, $\mathcal{B}_{\{\bar{x}_i\}}$, for the input arguments $\bar{x}_i$ is bounded by instance function applied on $\bar{x}_i$ by Theorem 4.2; and it should also be bounded by the size of the relation defined by q_i, i.e., Rel_{q_i}. Thus the smaller of these two quantities is taken to be the estimated value of $\mathcal{B}_{\{\bar{x}_i\}}$:

$$\mathcal{B}_{\{\bar{x}_i\}} \leq I_i = min\{instance(\bar{x}_i), \mathsf{Rel}_{q_i}\}.$$

Let $\bar{n}_i$ denote the input size to literal q_i. The number of instances, $\mathcal{B}_{\{\bar{y}_i\}}$, for the output arguments $\bar{y}_i$ of q_i is bounded by the product of $\mathcal{B}_{\{\bar{x}_i\}}$ and $\mathsf{Sol}_{q_i}(\bar{n}_i)$; and it should also be bounded by Rel_{q_i}. Their minimum is taken to be the estimated value of $\mathcal{B}_{\{\bar{y}_i\}}$:

$$\mathcal{B}_{\{\bar{y}_i\}} \leq O_i = min\{I_i \times \mathsf{Sol}_{q_i}(\bar{n}_i), \mathsf{Rel}_{q_i}\}.$$

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 Theorem 4.1:

$$\mathcal{B}_{\{v\}} \leq N_{\{v\}} = min\{O_i, \mathsf{Rel}_j \mid j \in lits(v)\}.$$

Once all the body literals are processed, the number of bindings possible for all the variables in the clause are estimated. Finally, using Theorem 4.2 again, we can estimate the number of instances, $\mathcal{B}_{\{\bar{x}_0, \bar{y}_0\}}$, for the arguments in the head:

$$\mathcal{B}_{\{\bar{x}_0, \bar{y}_0\}} \leq \mathsf{Rel}_p = instance((\bar{x}_0, \bar{y}_0)).$$

Example 4.1 Consider the program:

```
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).
```

Suppose the first argument of predicate p/3 is the input, and the binding patterns for predicates q/2 and r/2 have been recursively computed as $\langle 2, \lambda x.1\rangle$ and $\langle 3, \lambda x.1\rangle$, respectively. In this example, because all the predicates are nonrecursive, the corresponding solution sizes do not depend on the input size; therefore, we will ignore the size of arguments in the discussion. To compute $\mathsf{Rel}_\mathtt{p}$, we first compute

$$N_{\{\mathtt{X}\}} = \mathsf{Rel}_\mathtt{q} = 2$$

using Theorem 4.1. Using Theorems 4.1 and 4.2,

$$\begin{aligned}
&I_1 = min\{instance(\mathtt{X}), \mathsf{Rel}_\mathtt{q}\} = min\{2, 2\} = 2,\\
&O_1 = min\{I_1 \times \mathsf{Sol}_\mathtt{q}, \mathsf{Rel}_\mathtt{q}\} = min\{2 \times 1, 2\} = 2,\\
&N_{\{\mathtt{Y}\}} = min\{O_i, \mathsf{Rel}_\mathtt{r}\} = min\{2, 3\} = 2;
\end{aligned}$$

$$\begin{aligned}
&I_2 = min\{instance(\mathtt{Y}), \mathsf{Rel}_\mathtt{r}\} = min\{2, 3\} = 2,\\
&O_2 = min\{I_2 \times \mathsf{Sol}_\mathtt{r}, \mathsf{Rel}_\mathtt{r}\} = min\{2 \times 1, 3\} = 2,\\
&N_{\{\mathtt{Z}\}} = O_2 = 2.
\end{aligned}$$

The relation size for p/3 is bounded by

$$instance((\mathtt{X}, \mathtt{Y}, \mathtt{Z})) = N_{\{\mathtt{X}\}} \times N_{\{\mathtt{Y}\}} \times N_{\{\mathtt{Z}\}} = 8,$$

by Theorem 4.2. Thus, we have $\mathsf{Rel}_\mathtt{p} \equiv 8$. $\square$

The computation of Sol_p, the solution size for p, can be carried out in the same way as the computation of Rel_p. The only differences are that at the beginning the number of bindings for each variable in the input arguments $\bar{x}_0$ in the head is not estimated using Theorem 4.1, instead it is assigned to be 1; and at the end we only estimate the number of possible output arguments for the head, rather than both input and output arguments.

Example 4.2 Continuing the previous example, to compute $\mathsf{Sol}_\mathtt{p}$, we set $N_{\{\mathtt{X}\}} = 1$ because variable X is the input. Then we follow the same procedure as the computation of $\mathsf{Rel}_\mathtt{p}$. Using Theorems 4.1 and 4.2,

$$\begin{aligned}
&I_1 = min\{instance(\mathtt{X}), \mathsf{Rel}_\mathtt{q}\} = min\{1, 2\} = 1,\\
&O_1 = min\{I_1 \times \mathsf{Sol}_\mathtt{q}, \mathsf{Rel}_\mathtt{q}\} = min\{1 \times 1, 2\} = 1,\\
&N_{\{\mathtt{Y}\}} = min\{O_i, \mathsf{Rel}_\mathtt{r}\} = min\{1, 3\} = 1;
\end{aligned}$$

$$\begin{aligned}
&I_2 = min\{instance(\mathtt{Y}), \mathsf{Rel}_\mathtt{r}\} = min\{1, 3\} = 1,\\
&O_2 = min\{I_2 \times \mathsf{Sol}_\mathtt{r}, \mathsf{Rel}_\mathtt{r}\} = min\{1 \times 1, 3\} = 1,\\
&N_{\{\mathtt{Z}\}} = O_2 = 1.
\end{aligned}$$

Using Theorem 4.2 again, the number of outputs generated by a single input to p/3 is bounded by

$$instance((\mathtt{Y}, \mathtt{Z})) = N_{\{\mathtt{Y}\}} \times N_{\{\mathtt{Z}\}} = 1 \times 1 = 1.$$

So we have $\mathsf{Sol_p} \equiv \lambda x.1$, i.e., p/3 generates at most one output for each input. □

Theorem 4.2 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 Theorem 4.2 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, e.g., in the program

```
p(X, f(Y, Z)) :- q(X, Y, Z).
q(a, b, c).   q(a, d, e).
```

Suppose X is the input of p/2. Though 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 theorem 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.

Theorem 4.3 *Let $\bar{y}$ be the output arguments of a literal, and O be the computed number of instances for $\bar{y}$. If $S \subseteq vars(\bar{y})$ is a set of variables, then $\mathcal{B}_S \leq min(\prod_{v \in S} N_{\{v\}}, O)$.*

Proof By Theorem 4.2, $\mathcal{B}_S \leq \prod_{v \in S} N_{\{v\}}$. Since $\mathcal{B}_S \leq \mathcal{B}_{vars(\bar{y})}$ and $\mathcal{B}_{vars(\bar{y})} \leq O$, we have $\mathcal{B}_S \leq O$. □

Using Theorem 4.3, we can improve instance function as follows: let T be a tuple of terms, the variables in T can be divided into $V_{k_1}, \ldots, V_{k_n}$ sets of variables such that the variables in V_{k_i} become bound in literal q_{k_i}, and O_{k_i} be the computed number of output instances for literal q_{k_i}. Then a new realization of instance function can be defined as:

$$instance_2(T) = \prod_{i=1}^{n} min\{\prod_{v \in V_{k_i}} N_{\{v\}}, O_{k_i}\}.$$

Other cases in which the variable bindings are dependently generated occur between variables in the input and output arguments of the same literal. Consider the program

```
p(X, f(Y, Z)) :- q(X, Y), r(Y, Z).
q(a, b).        q(a, c).
r(b, d).        r(c, e).
```

Suppose X is the input of p/2. Then 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 theorem 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.

Theorem 4.4 *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 $S \subseteq vars((\bar{x}, \bar{y}))$, then $\mathcal{B}_S \leq min(\prod_{v \in S} N_{\{v\}}, O)$.*

Proof By Theorem 4.2, $\mathcal{B}_S \leq \prod_{v \in S} N_{\{v\}}$. Let $\bar{n}$ denote the input size to literal q. Since $\mathcal{B}_{vars((\bar{x},\bar{y}))} \leq \mathcal{B}_{\{\bar{x}\}} \times \mathsf{Sol}_q(\bar{n})$, and $\mathcal{B}_{vars((\bar{x},\bar{y}))} \leq \mathsf{Rel}_q$, $\mathcal{B}_{vars((\bar{x},\bar{y}))} \leq O$. Because $\mathcal{B}_S \leq \mathcal{B}_{vars((\bar{x},\bar{y}))}$, we have $\mathcal{B}_S \leq O$. □

Using Theorem 4.4, we can further improve instance function as follows: let T be a tuple of terms, such that the variables in T can be divided into $V_{k_1}, \ldots, V_{k_n}$ sets of variables such that the variables in V_{k_i} become bound in literal q_{k_i}. The improvement can be achieved by merging these variable sets using Theorem 4.4 such that the number of resulting variable sets is fewer. This merging process can proceed by considering the literals in reverse topological order from literal dependency graph. Let q_{k_i} be the literal under consideration, if the variable set corresponding to q_{k_i} is nonempty, then we can move all the variables, which occur in the input arguments of q_{k_i} and which are in the variable set corresponding to a predecessor of q_{k_i}, into the set corresponding to q_{k_i}. Let $V_{l_1}, \ldots, V_{l_m}$ be the resulting sets of variables from the merging process with $m \leq n$, and O_{l_i} be the computed number of output instances for literal q_{l_i}. Then we can define a new realization of instance function as:

$$instance_3(T) = \prod_{i=1}^{m} min\{\prod_{v \in V_{l_i}} N_{\{v\}}, O_{l_i}\}.$$

Theorems 4.3 and 4.4 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:

```
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).
```

Suppose `X` is the input of `p/2`. Then 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 `Z` 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. The following theorem provides a rule for improving the estimation in such cases:

Theorem 4.5 *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. Let $S \subseteq vars((\bar{x}_i, \bar{y}_i, \bar{x}_j, \bar{y}_j))$ be a set of variables. If*

1. $vars(\bar{x}_j) \subseteq vars((\bar{x}_i, \bar{y}_i))$,
2. $instance(\bar{x}_j) = O_i$,
3. $instance(\bar{x}_j) \leq \mathsf{Rel}_{q_j}$,
4. $instance(\bar{x}_j) \times \mathsf{Sol}_{q_j}(\bar{n}_j) \leq \mathsf{Rel}_{q_j}$,

then $\mathcal{B}_S \leq min(\prod_{v \in S} N_{\{v\}}, O_j)$.

Proof By Theorem 4.2, $\mathcal{B}_S \leq \prod_{v \in S} N_{\{v\}}$. Also,

$$\begin{aligned}
\mathcal{B}_{vars((\bar{x}_i,\bar{y}_i,\bar{x}_j,\bar{y}_j))} &\leq \mathcal{B}_{vars((\bar{x}_i,\bar{y}_i,\bar{x}_j))} \times \mathsf{Sol}_{q_j}(\bar{n}_j) & \\
&= \mathcal{B}_{vars((\bar{x}_i,\bar{y}_i))} \times \mathsf{Sol}_{q_j}(\bar{n}_j) & \text{(from } \mathit{1}\text{)} \\
&\leq O_i \times \mathsf{Sol}_{q_j}(\bar{n}_j) & \\
&= instance(\bar{x}_j) \times \mathsf{Sol}_{q_j}(\bar{n}_j) & \text{(from } \mathit{2}\text{)} \\
&= I_j \times \mathsf{Sol}_{q_j}(\bar{n}_j) & \text{(from } \mathit{3}\text{)} \\
&= O_j. & \text{(from } \mathit{4}\text{)}
\end{aligned}$$

Because $\mathcal{B}_S \leq \mathcal{B}_{vars((\bar{x}_i,\bar{y}_i,\bar{x}_j,\bar{y}_j))}$, we have $\mathcal{B}_S \leq O_j$. □

Let q_i and q_j be two literals. We say literal q_j *subsumes* literal q_i if they satisfy the four conditions specified in Theorem 4.5. Using a proof similar to Theorem 4.5, it is very easy to verify by induction that the result of Theorem 4.5 can be generalized to any number of literals:

Theorem 4.6 *Let $q_1, \ldots, q_n$ be literals, $\bar{x}_1, \bar{y}_1, \ldots, \bar{x}_n, \bar{y}_n$ be the corresponding input and output arguments, and O_n be the computed number of output instances for literal q_n. Let $S \subseteq vars((\bar{x}_1, \bar{y}_1, \ldots, \bar{x}_n, \bar{y}_n))$ be a set of variables. If q_{i+1} subsumes q_i for $1 \leq i < n$, then $\mathcal{B}_S \leq min(\prod_{v \in S} N_{\{v\}}, O_n)$.*

Using Theorem 4.6, we can improve instance function once again. We apply a merging process similar to that of $instance_3$. For each literal under consideration, if the corresponding variable set is nonempty, we first move all the variables which satisfy Theorem 4.6 into the variable set, then we move all the variables which satisfy Theorem 4.4 into the variable set. Let $V_{k_1}, \ldots, V_{k_n}$ be the resulting variable sets from the merging process, and O_{k_i} be the computed number of output instances for literal q_{k_i}. Then the new instance function is defined as:

$$instance_4(T) = \prod_{i=1}^{n} min\{\prod_{v \in V_{k_i}} N_{\{v\}}, O_{k_i}\}.$$

Example 4.3 Consider the predicate `perm/2` defined in Example 2.1. Assume the binding pattern $\langle \mathsf{Rel}_{\mathtt{select}}, \mathsf{Sol}_{\mathtt{select}} \rangle$ for predicate `select/3` has been recursively computed as $\langle \infty, \lambda x.x \rangle$ (see the Appendix A for details). Since predicate `perm/2` succeeds for every input list, we obtain $\mathsf{Rel}_{\mathtt{perm}} \equiv \infty$. To compute $\mathsf{Sol}_{\mathtt{perm}}$, we first set $N_{\{\mathtt{X}\}} = 1$. Using Theorem 4.1, we obtain

$$\begin{aligned}
&O_1 = N_{\{\mathtt{X}\}} \times \mathsf{Sol}_{\mathtt{select}}(body_1[2]) = head[1], \\
&N_{\{\mathtt{R}\}} = min\{O_1, \mathsf{Rel}_{\mathtt{select}}\} = head[1], \\
&N_{\{\mathtt{Y}\}} = min\{O_1, \mathsf{Rel}_{\mathtt{select}}, \mathsf{Rel}_{\mathtt{perm}}\} = head[1];
\end{aligned}$$

$$\begin{aligned}
&O_2 = N_{\{\mathtt{Y}\}} \times \mathsf{Sol}_{\mathtt{perm}}(body_2[1]) = head[1] \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1), \\
&N_{\{\mathtt{Rs}\}} = min\{O_2, \mathsf{Rel}_{\mathtt{perm}}\} = head[1] \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1),
\end{aligned}$$

Using Theorem 4.5, Since $\{\mathtt{Y}\} \subseteq \{\mathtt{R}, \mathtt{X}, \mathtt{Y}\}$, $instance(\mathtt{Y}) = head[1] = O_1$, $instance(\mathtt{Y}) \leq \mathsf{Rel}_{\mathtt{perm}}$, and $instance(\mathtt{Y}) \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1) = head[1] \times \mathsf{Sol}_{perm}(head[1]-1) \leq \mathsf{Rel}_{\mathtt{perm}}$, we obtain

$$instance([\mathtt{R}|\mathtt{Rs}]) = min\{N_{\{\mathtt{R}\}} \times N_{\{\mathtt{Rs}\}}, O_2\} = head[1] \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1).$$

Notice that variables R and Rs are from distinct literals. Thus, we have the equation

$$\mathsf{Sol}_{\mathtt{perm}}(x) = x \times \mathsf{Sol}_{\mathtt{perm}}(x-1).$$

This equation can be solved, with the boundary condition $\mathsf{Sol}_{\mathtt{perm}}(0) = 1$ from the first clause of perm/2, to obtain $\mathsf{Sol}_{\mathtt{perm}} \equiv \lambda x.x!$. □

Note that in the general algorithm of Figure 5, 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 techniques similar to those specified in Theorems 4.3 – 4.6. In general, however, the more information is used in the algorithm, the less effective are the derived optimization techniques. Because of this, it is difficult to predict how beneficial such additional information is with regard to the precision of the final result.

Up to now we have assumed that all the solutions generated are distinct. However, in practice, a single input may generate duplicate solutions. In general it is necessary to account for duplicate solutions, since otherwise erroneous results may be obtained for time complexity analysis. For programs that generate duplicate solutions, Theorems 4.1 and 4.2 may not hold any more, and we have to use more conservative methods. For example, consider the program:

```
p(X, W) :- q(X, Y), r(Y, Z), s(Z, W).
q(a, b1).       q(a, b2).
r(b1, c).       r(b2, c).
s(c, d).
```

Suppose X is the input of p/2. Then variable Z would be bound to c twice due to the distinct bindings, b1 and b2, of Y. Using Theorem 4.2, since the relation size for predicate s/2 is 1, we would infer that the number of possible bindings for Z is 1. However, because of the duplicates generated for Z, duplicates are generated for W by the input a to p/2. Thus, in practice, we need to make sure that the predicates are duplicate-free in order to apply the techniques described in this subsection. If not, we can only infer that literal s/2 would be invoked twice, and since for each input to s/2 only one output is generated, W would get two bindings, instead of just one. A sufficient condition for duplicate-free predicates is given in [26].

Finally, it may sometimes be necessary to explicitly account for implicit failures. The problem is illustrated by the following program to check membership in a list:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

A straightforward analysis would infer the equation '$\mathsf{Sol}_{\mathtt{member}}(n) = 1$' for the first clause, and '$\mathsf{Sol}_{\mathtt{member}}(n) = \mathsf{Sol}_{\mathtt{member}}(n-1)$' for the second, and since the two clauses are not mutually exclusive, the resulting equation would be obtained as

$$\mathsf{Sol}_{\mathtt{member}}(n) = \mathsf{Sol}_{\mathtt{member}}(n-1) + 1.$$

The problem is that there is no base case from which this equation can be solved. In this case, we must explicitly account for the fact that the base case fails and yields no solutions: this requires adding the equation

$$\mathsf{Sol}_{\texttt{member}}(0) = 0.$$

The resulting equations can be solved to give the expected result.

4.2 Linear Arithmetic Constraints

We now present a simple algorithm for estimating the relation size for predicates which can be "unfolded" into a conjunction and/or disjunction of linear arithmetic constraints on a set of variables. The constraints may involve any of the following comparison operators: $=, \neq, >, \geq, <$, and $\leq$. The types of the variables in the predicates are assumed to be given as integer intervals.

Since the manipulation of general n-ary constraints (involving n variables) can incur exponential cost [30], we approximate the set of n-ary constraints by a set of binary constraints (involving at most 2 variables) through the "projection" of an n-ary relation onto a set of binary relations. As shown in [30], the set of projected binary constraints is a minimal extra relation of the original n-ary constraints. If the set of n-ary arithmetic constraints is linear and the types for the variables can be represented as integer intervals, then a set of binary constraints can be easily projected from n-ary constraints via interval arithmetic manipulation.

The set of binary arithmetic constraints can be represented as a graph $G = (V, E)$, called a *consistency graph*. Each vertex $v_{i,j}$ in V denotes the variable binding of a value d_j to a variable x_i, $x_i \leftarrow d_j$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. There is an edge $\langle v_{p,g}, v_{q,h} \rangle$ between two vertices $v_{p,g}$ and $v_{q,h}$ if the two bindings $x_p \leftarrow d_g$ and $x_q \leftarrow d_h$ satisfy all the constraints involving variables x_p and x_q. The two bindings are then said to be *consistent*. The set of vertices $V_i = \{v_{i,1}, \ldots, v_{i,m}\}$ corresponding to a variable x_i is called the *binding set* of x_i. The *order* of a consistency graph G is (n, m) if G corresponds to a set of constraints involving n variables and m values. Because two distinct values cannot be bound to the same variable at the same time, no pair of vertices in a binding set are adjacent. Therefore, the consistency graph of a set of constraints involving n variables is an n-partite graph.

A *clique* of a graph G is a subgraph of G whose vertices are pairwise adjacent. An *n-clique* is a clique with n vertices. Because a solution S for a set of constraints C involving n variables is an n-tuple of bindings of values to variables such that all the constraints are satisfied, every pair of variable bindings in S is consistent. Thus S corresponds to an n-clique of the consistency graph of C, and the number of solutions for C is equal to the number of n-cliques in the consistency graph of C.

From now on we will concentrate on describing an algorithm for estimating the number of n-cliques in a consistency graph. The basic idea behind the algorithm is to identify the number of n-cliques each vertex and edge in the graph may belong to. To this end, we associate a weight with each edge in the graph. A *weighted consistency graph* $G = (V, E, W)$ is a consistency graph associated with a function $W : V \times V \rightarrow \mathcal{N}$ such that W assigns a positive integer to an edge $\langle u, v \rangle$ if $\langle u, v \rangle \in E$, and assigns 0 to $\langle u, v \rangle$ if $\langle u, v \rangle \notin E$.

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 $K(G,n) = \sum_{H \in S} K(H,n)$. Initially every edge is assigned a weight of 1.

The number of n-cliques a vertex may belong to depends on the connectivity with its adjacent vertices. Let $G = (V, E, W)$ be a weighted consistency graph and $N_{G,v} = \{w \in V \mid \langle v, w\rangle \in E\}$ be the *neighbors* of a vertex v. The *adjacency graph* of v, $Adj_G(v)$, is the subgraph of G *induced* by $N_{G,v}$, namely, $Adj_G(v) = (N_{G,v}, E_{G,v}, E_{G,v})$, where $E_{G,v}$ is the set of edges in E that join the vertices in $N_{G,v}$ and $W_{G,v}(\langle u, w\rangle) = min(W(\langle v, u\rangle), W(\langle v, w\rangle), W(\langle u, w\rangle))$, for each edge $\langle u, w\rangle \in E_{G,v}$. The following theorem shows the relationship between the number of n-cliques in a graph and the number of $(n-1)$-cliques in the adjacency graphs corresponding to the vertices in a binding set.

Theorem 4.7 *Let G be a weighted consistency graph of order (n,m). Then for each binding set $V = \{v_1, \ldots, v_m\}$,*

$$K(G,n) = \sum_{i=1}^{m} K(Adj_G(v_i), n-1). \tag{1}$$

Proof Let $G_i = (V_i, E_i, W_i)$ 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 $Adj_G(v_i) = (V_A, E_A, W_A)$. Because v_i is adjacent to every vertex in N_{G,v_i}, $min\{W_i(e) \mid e \in E_i\} = min\{W_A(e) \mid e \in E_A\}$. Therefore, $K(G_i, n) = K(Adj_G(v_i), n-1)$. □

Theorem 4.7 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, a direct computation requires exponential time $O(m^n)$. Therefore, we define an operator to combine the set of subgraphs $Adj_G(v_1), \ldots, Adj_G(v_m)$ in Formula (1) into a graph H such that $K(H, n-1)$ is an upper bound on $K(G,n)$.

We define a binary operator $\oplus$, called *graph addition*, on two weighted consistency graphs as follows: let $G_1 = (V, E_1, W_1)$ and $G_2 = (V, E_2, W_2)$ be two weighted consistency graphs with the same set of vertices. 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 $e \in E_{1\oplus 2}$. The effect of graph addition on the number of n-cliques is shown in the following theorems:

Theorem 4.8 *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*

$$K(G_1 \oplus G_2, n) \geq K(G_1, n) + K(G_2, n). \tag{2}$$

Proof By a straightforward case analysis. □

Theorem 4.9 *Let G be a weighted consistency graph of order (n,m). Then for each binding set $V = \{v_1, \ldots, v_m\}$,*

$$K(G,n) \leq K(\bigoplus_{i=1}^{m} Adj_G(v_i), n-1). \tag{3}$$

Let $G = (V, E, W)$ be a weighted consistency graph of order (n, m). The algorithm proceeds as follows:

begin
  $G_1 := G;$ /* $G_1 = (V_1, E_1, W_1)$ */
  for $i := 1$ **to** $n - 2$ **do**
    $G_{i+1} := \bigoplus_{j=1}^{m} Adj_{G_i}(v_{i,j});$ /* $G_{i+1} = (V_{i+1}, E_{i+1}, W_{i+1})$ */
  od
  $K(G, n) = \sum_{e \in E_{n-1}} W_{n-1}(e);$
end

Figure 6: An algorithm for estimating the number of n-cliques in a weighted consistency graph

Proof By Theorems 4.7 and 4.8. □

We now summarize the algorithm for computing an upper bound on $K(G, n)$ for a weighted consistency graph G of order (n, m). We apply Theorem 4.9 repeatedly to a sequence of consecutively smaller graphs. By starting with the graph G, at each iteration, one binding 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 binding set. This binding set elimination process continues until there are only two binding sets left. The resultant graph is now a bipartite graph. By definition, the number of 2-cliques in a bipartite weighted consistency graph is the sum of the weights of the edges (2-cliques) in the graph. This algorithm is shown in Figure 6. The time complexity for this algorithm is $O(n^3m^3)$ for a graph of order (n, m) [24].

Example 4.4 Consider the following predicate which specifies a set of precedence constraints among a set of tasks:

```
schedule(SA, SB, SC, SD, SE, SF, SEnd) :-
    SB ≥ SA + 1,     SC ≥ SA + 1,     SD ≥ SA + 1,     SE ≥ SB + 5,
    SE ≥ SC + 3,     SF ≥ SD + 5,     SF ≥ SE + 2,     SEnd ≥ SF + 1.
```

Suppose the integer interval $[0, 10]$ is given as the type for each of the variables SA, SB, ..., SEnd, we can then use the algorithm to estimate the number of legal schedules or the relation size for predicate schedule/7 to be 71. In this case, this estimate is exact. □

4.3 Binary Nonequality Constraints

We now present a simple algorithm for estimating the relation size for predicates which can be "unfolded" into a conjunction of binary nonequality constraints on a set of variables. The constraints are in the form of $X \neq Y$ for any two variables X and Y. The types of the variables in a predicate are assumed to be the same finite set of constants.

We first show that the problem of computing the number of bindings that satisfy a set of binary nonequality constraints on a set of variables with the same type can be transformed into the problem of computing the chromatic polynomial of a graph. The *chromatic polynomial* of a graph G, denoted by

$C(G,k)$, is a polynomial in k and represents the number of different ways G can be colored by using no more than k colors. The transformation of the problem goes as follows: let G be the graph consisting of a set of vertices, each of them corresponds to a variable, and there is an edge between two vertices if there is a binary nonequality constraint between the corresponding two variables. If the size of the type for variables is k, then the number of ways of coloring G using no more than k colors is exactly the number of bindings that satisfy the set of binary nonequality constraints.

Unfortunately, the problem of computing the chromatic polynomial of a graph is NP-hard, because the problem of k-colorability of a graph G is equivalent to the problem of deciding whether $C(G,k) > 0$ and the problem of graph k-colorability is NP-complete [19]. However, it turns out that if we can efficiently compute a lower bound on the chromatic number of a graph, then we can efficiently compute an upper bound on the chromatic polynomial of a graph. The *chromatic number* of a graph G, written as $\chi(G)$, is the minimum number of colors necessary to color G so that adjacent vertices have different colors. The following theorem by Bondy [1] gives a lower bound on the chromatic number of a graph.

Theorem 4.10 *Let G be a graph with m vertices $\{1,\ldots,m\}$, with the degree of vertex i denoted by $d(i)$, such that $d(1) \geq \cdots \geq d(n)$. Let σ_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

$$\sum_{j=1}^{k-1} \sigma_j < n. \tag{4}$$

Then $\chi(G) \geq k$. □

Let $\rho(G)$ denote the largest integer k satisfying Equation (4) for a graph G. Then we can design an algorithm to compute an upper bound on the chromatic polynomial of a graph G. The basic idea is to start with a subgraph that consists of only a single vertex of the graph, then repeatedly build larger and larger subgraphs by adding a vertex at a time into the previous subgraph. When a vertex is added, the edges connecting that vertex to vertices in the previous subgraph are also added. At each iteration, the chromatic polynomial for the corresponding subgraph is computed using the computed polynomial for the previous subgraph and the number of ways of coloring the current added vertex.

Let $G = (V, E)$ be a graph with n vertices. Suppose $\omega = v_1, \ldots, v_n$ is an ordering of V. We define two sequences of subgraphs of G according to ω. The first is a sequence of subgraphs $G_1, \ldots, G_n$, called *accumulating subgraphs*, where $G_i = (V_i, E_i)$, $V_i = \{v_1, \ldots, v_i\}$, and E_i is the set of edges of G that join the vertices of V_i, for $1 \leq i \leq n$, The second is a sequence of subgraphs $G'_2, \ldots, G'_n$, called *interfacing subgraphs*, where $G'_i = (V'_i, E'_i)$, V'_i is the set of vertices of G_{i-1} that are adjacent to vertex v_i, and E'_i is the set of edges of G_{i-1} that join the vertices of V'_i, for $1 < i \leq n$.

An algorithm for computing the chromatic polynomial of a graph, based on the construction of accumulating subgraphs and interfacing subgraphs, is shown in Figure 7. This algorithm constructs the accumulating subgraphs according to an ordering on the set of vertices. At each iteration, the number

Let $G = (V, E)$ be a graph with n vertices. The algorithm proceeds as follows:

begin
  generate an ordering $\omega = v_1, \ldots, v_n$ of V;
  $C(G, k) := k$;
  $G_1 := (\{v_1\}, \emptyset)$;
  for $i := 2$ **to** n **do**
    compute the interfacing subgraph G'_i;
    $C(G, k) := C(G, k) \times (k - \rho(G'_i))$;
    compute the accumulating subgraph G_i;
  od
end

Figure 7: An algorithm for computing the chromatic polynomial of a graph

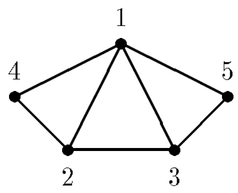

Figure 8: An example

of ways of coloring the new added vertex is computed based on a lower bound on the chromatic number of the corresponding interfacing subgraph.

Example 4.5 Consider the graph shown in Figure 8. The imposed ordering is denoted by the labels of vertices. The corresponding accumulating subgraphs and interfacing subgraphs are shown in Figure 9. The computed chromatic polynomial is $k(k-1)(k-2)^3$. In this case, that is also the exact chromatic polynomial of the graph. □

Theorem 4.11 *Let $G = (V, E)$ be a graph with n vertices and ω be an ordering of V. Suppose the interfacing subgraphs of G corresponding to ω are $G'_2, \ldots, G'_n$. Then*

$$C(G, k) \leq k \prod_{i=2}^{n} (k - \rho(G'_i)). \tag{5}$$

Proof Suppose $G_1, \ldots, G_n$ are the accumulating subgraphs of G corresponding to ω. The proof is by induction on G_j, for $1 \leq j \leq n$. In base case, G_1 is a graph consisting of one vertex v_1, so $C(G_1, k) = k$. Suppose Equation (5) is satisfied by G_j for some j, $1 \leq j < n$. Then consider adding the vertex n_{j+1}

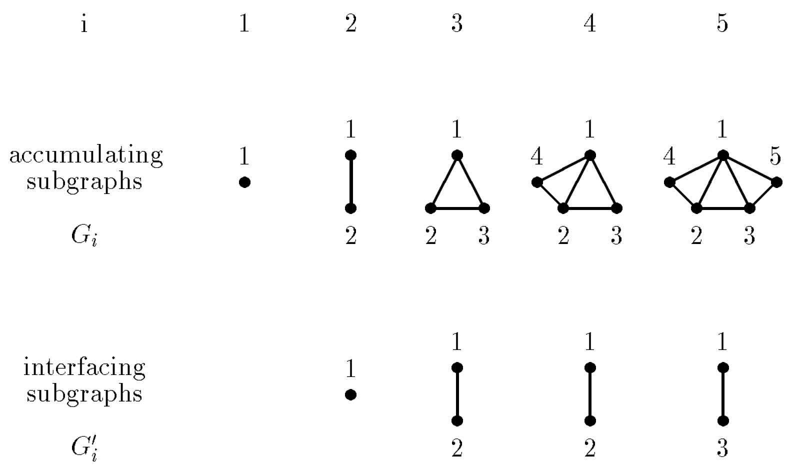

Figure 9: The accumulating and interfacing subgraphs of the graph in Figure 8

and associated edges into G_j to form G_{j+1}. Since $\chi(G'_{j+1})$ is the least number of colors necessary for coloring G'_{j+1}, we have

$$C(G_{j+1}, k) \leq (k - \chi(G'_{j+1}))C(G_j, k).$$

By Theorem 4.10, we have

$$(k - \chi(G'_{j+1}))C(G_j, k) \leq (k - \rho(G'_{j+1}))C(G_j, k).$$

Therefore, $C(G_{j+1}, k) \leq (k - \rho(G'_{j+1}))C(G_j, k)$. From the hypothesis, we obtain

$$C(G_{j+1}, k) \leq k \prod_{i=2}^{j+1} (k - \rho(G'_i)).$$

Since $G_n = G$, the theorem is proved. $\square$

Since the bound in Equation (5) may depend on the ordering of the vertices in the graph, we use a heuristic to find a "good" ordering. The intuition is that if the degrees of the interfacing subgraphs are smaller, then the lower bound of the chromatic number is more likely to be closer to the chromatic number. Therefore, we use the decreasing order on the degrees of vertices. This is also the ordering proposed by Welsh and Powell for coloring a graph [43]. The ordering in the graph of Figure 8 is such an ordering. The complexity of the algorithm for computing chromatic polynomial of a graph is $O(n^2 \log n + nm)$ for a graph with n vertices and m edges [23].

Example 4.6 Consider the following predicate:

```
map_color(X1, X2, X3, X4, X5) :-
```

```
color(X1), color(X2), color(X3), color(X4), color(X5),
X1 ≠ X2, X1 ≠ X3, X1 ≠ X4, X1 ≠ X5, X2 ≠ X3, X2 ≠ X4, X3 ≠ X5.
```

Suppose a finite set of colors is given as the type for variables X1, ..., X5. We can then use the algorithm for computing chromatic polynomial of a graph to estimate the number of solutions generated by predicate map_color/5. The corresponding graph for predicate map_color/5 is that in Example 4.5. From Example 4.5, we know that the number of solutions generated by predicate map_color/5 is bounded above by $k(k-1)(k-2)^3$, where k is the number of colors in the type. In particular, we can immediately conclude that no solutions are possible for this predicate if fewer than 3 colors are used. □

4.4 Combining the Algorithms

We now show how to properly combine the algorithms described above in relation size analysis. When type information is available for a predicate, each of its clauses is first checked to see if it can be unfolded into a conjunction of binary nonequality constraints where the variables range over the same finite set of constants. In this case, the constraint graph is constructed and the algorithm for estimating chromatic polynomial of a graph is utilized to estimate the number of solutions possible for those variables. Otherwise, the clause is checked to see if it can be unfolded into a conjunction and/or disjunction of linear arithmetic constraints, and if the types of variables are represented as integer intervals. In this case, the algorithm for estimating the number of n-cliques of a consistency graph is employed to estimate the number of bindings possible. In other cases, the general algorithm is used. As in the case of size relationships, recursive literals are handled by using symbolic expressions to denote the number of solutions generated by them, and solving (or giving upper bound estimates to) the resulting difference equations.

The number of solutions for a predicate can then be obtained by combining the expressions for the number of solutions obtained for each of its clauses. Notice that the combining operation for argument size relations is maximization, while summation is used for the expressions for the number of solutions. The number of solutions a predicate can generate is the maximum of the number of solutions that can be generated by each mutually exclusive cluster of clauses; the number of solutions any cluster can generate is bounded by the sum of the number of solutions that can be generated by each clause within the cluster.

5 Time Complexity

This section presents the analysis for estimating the time complexity of predicates. Let $\mathsf{T}_p : \mathcal{N}^n_{\perp,\infty} \to \mathcal{N}_\infty$ be a function that denotes the time complexity for a predicate p with n input positions. The time complexity of a clause can be bounded by the time complexity of head unification together with the time complexity of each of its body literals. Consider a clause C defined as '$\mathtt{H} :- \mathtt{L}_1, \ldots, \mathtt{L}_m$'. Because of backtracking, the number of times a literal will be executed depends on the number of solutions that the literals preceding it can generate. Suppose that the input size to clause C is $\bar{n}$, and the input size to literal L_i is $\bar{n}_i$. Then the time complexity of clause C can be expressed as

$$\mathsf{T}_C(\bar{n}) \le \tau + \sum_{i=1}^{m} (\prod_{j \prec i} \mathsf{Sol}_{L_j}(\bar{n}_j)) \mathsf{T}_{L_i}(\bar{n}_i), \tag{6}$$

where τ is the time needed to resolve the head `H` of the clause with the literal being solved. Here we use $j \prec i$ to denote that $\mathtt{L}_j$ precedes $\mathtt{L}_i$ in the literal dependency graph for the clause.

There are a number of different metrics that can be used as the unit of time complexity in these expressions, e.g., 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 τ is 1; if it is the number of unifications, then τ is the arity of the clause head.

Example 5.1 Consider again the predicate `perm/2` defined in Example 2.1. The time complexity of the recursive clause of `perm/2` can be expressed as

$$\mathsf{T}_{\mathtt{perm}}(head[1]) = \tau + \mathsf{T}_{\mathtt{select}}(head[1]) + \mathsf{Sol}_{\mathtt{select}}(head[1]) \times \mathsf{T}_{\mathtt{perm}}(\mathsf{Sz}^{(3)}_{\mathtt{select}}(head[1])).$$

Assume the time complexity metric is the number of resolutions, the time complexity for predicate `select/3` has been computed as $\mathsf{T}_{\mathtt{select}} \equiv \lambda x.2x$ (see the Appendix A for details), and $\mathsf{Sol}_{\mathtt{select}} \equiv \lambda x.x$ and $\mathsf{Sz}^{(3)}_{\mathtt{select}} \equiv \lambda x.x - 1$ have been computed as in previous sections. Then the time complexity for the recursive clause of `perm/2` can be simplified to

$$\mathsf{T}_{\mathtt{perm}}(head[1]) = head[1] \times \mathsf{T}_{\mathtt{perm}}(head[1] - 1) + 2 \times head[1] + 1.$$

This equation can be solved to obtain the time complexity

$$\mathsf{T}_{\mathtt{perm}} \equiv \lambda x. \sum_{i=1}^{x} (3x!/i!) + 3x! - 2.$$

with the boundary condition $\mathsf{T}_{\mathtt{perm}}(0) = 1$ from the first clause. $\square$

As in the case of estimating the number of solutions, the clauses are partitioned into mutually exclusive clusters. The time complexity for each such cluster can be obtained by summing the time complexity for each of its clauses. In addition to that, however, we also need to take into account the failure cost introduced by trying to solve the clauses in other clusters. The failure cost from solving a clause in another cluster can be estimated by considering the sources leading to the mutual exclusion among clauses. This information can be easily produced by mutual exclusion analysis. After the failure costs are added into the time complexity for each cluster, the time complexity of a predicate is then obtained as the maximum of the time complexities of these clusters.

As for the analysis for the number of solutions, it may be necessary to explicitly account for implicit failures, e.g., for the predicate `member/2` discussed earlier, in order to produce difference equations that can be solved. This can be done in a manner analogous to that for the number of solutions analysis.

6 Automatic Solution of Difference Equations

Algorithms for the automatic solution of difference equations have been studied by a number of researchers [5, 16, 31]. It is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable, so it suffices to consider the solution of a single linear difference equation in one variable. The programs in [5, 16, 31] solve linear difference equations with constant coefficients using either characteristic equations or generating functions. Using exponential generating functions, the problem of solving linear difference equations with polynomial coefficients can be reduced to that of solving 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.

Nonlinear difference equations may arise in the analysis for argument size functions, where the size functions for a literal that succeeds a recursive literal are nonlinear. They may also arise in the analysis for the number of solutions, where multiplication is applied to compute the number of instances of arguments. Furthermore, maximum and minimum functions also introduce nonlinearity into the equations. 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. In addition, although there is no algorithm for solving arbitrary nonlinear difference equations, there are many special form nonlinear difference equations which have known solutions.

Finally, to automate the whole analysis, it is necessary to return a closed form solution for all the difference equations. 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 particular, difference equations involving *max* and *min*—which occur frequently when analyzing logic programs—are 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 $max(e_1, e_2)$, 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 $e_1 + e_2$. 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, and the appropriate choice is left to the implementors.

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 equations. Now the exact form of the difference equations that are obtained for a predicate depend on the size measures under consideration, i.e., 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 get quite messy, and are beyond the scope of this paper.

7 Implementation

CASLOG (Complexity Analysis System for LOGic) is a prototype implementation of the techniques described in previous sections. It consists of five major components: a preprocessor, argument size analyzer, number of solution analyzer, time complexity analyzer and difference equation solver. The organization of CASLOG is shown in Figure 10.

The preprocessor consists of five modules: mode analysis, data dependency analysis, mutual exclusion analysis, type analysis and size measure analysis. At this time, mode analysis, type analysis and size measure analysis have not been implemented (as indicated by dashed boxes in Figure 10) and the users have to supply this information via declarations in the program. Data dependency analysis uses mode information to build an argument dependency graph and a literal dependency graph for each clause, while mutual exclusion analysis classifies the clauses into mutually exclusive clusters for each predicate.

The argument size analyzer applies size measure information to derive the set of argument size relations associated with each clause, and computes the argument size functions for each output position in the clause head by performing normalization on the set of argument size relations.

The number of solution analyzer is divided into two subcomponents: the relation size analyzer and the solution size analyzer. In relation size analysis, when type information is available for a predicate, the predicate is checked to see if it can be unfolded into a conjunction of binary nonequality constraints, or a conjunction and/or disjunction of linear arithmetic constraints. In this case, the appropriate special algorithm is used to estimate the relation size of the predicate. Otherwise, the general algorithm is used to estimate the relation size of the predicate. The general algorithm is used for solution size analysis.

The time complexity analyzer combines information about argument sizes and number of solutions to estimate the time complexity function for each predicate.

In argument size analysis, solution size analysis and time complexity analysis, the cost expressions for recursive clauses are in the form of difference equations. A difference equation solver has been implemented in CASLOG. It can solve a number of common classes of difference equations, e.g., first order linear difference equations, second order linear difference equations with constant coefficients, difference equations from divide-and-conquer paradigm and a special class of difference equations derivable from clauses with the size measure term-size. Apart from this, we have incorporated the Maple Symbolic Computation System [4] into CASLOG, so that the system can resort to Maple when the difference equations encountered cannot be handled by its difference equation solver. If neither our difference equation solver nor Maple can deal with the difference equations encountered, a conservative upper bound, $\lambda\bar{x}.\infty$, is returned.

The core modules of the system consists of the argument size analyzer, the relation size and solution size analyzer, and the time complexity analyzer. These modules share several common features: depending on data dependency information to compute the complexity expressions for each clause, relying on mutual exclusion information to compose the complexity expressions for a predicate from the expressions of its clauses, and resorting to difference equation solver to analyze recursive clauses. These common features allow the analyses be performed in a unified framework that simplifies proofs of correctness and the implementation of the algorithms. This framework is then enhanced by incorporating two special algorithms to improve the relation size analysis for two special classes of predicates. The

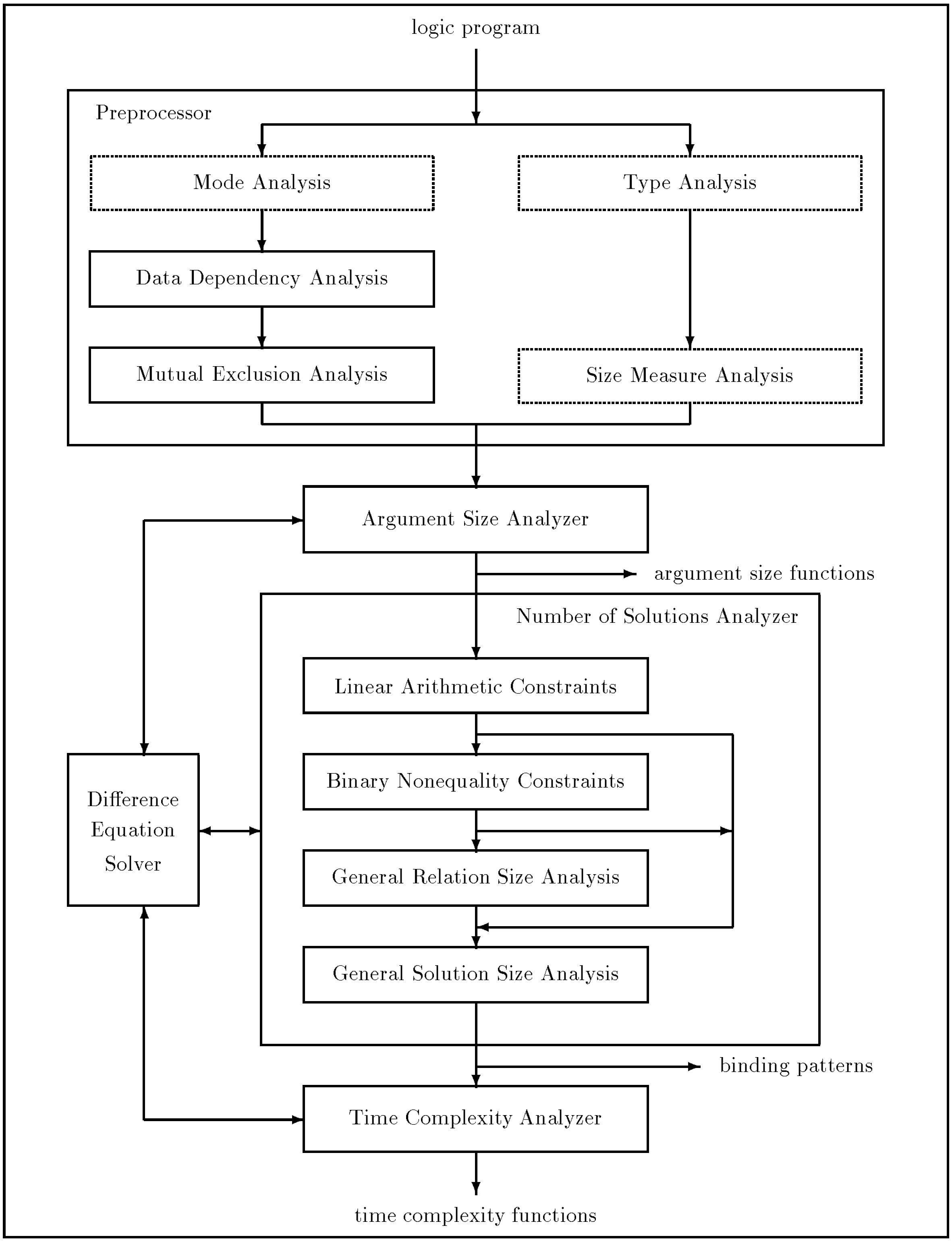

Figure 10: Organization of CASLOG

system is implemented on top of SICStus Prolog [2]. Preliminary results on the speed of the system, running on a Sparcstation-2 with 64 Mbytes of memory, is given in Appendix B.

There are some programs that the system cannot deal with very well. These include transitive closure programs, where the problem is that it is not clear what size measures to use; chain programs, i.e., programs where the output of a recursive literal is used as input by another recursive literal, as is the case, for example, in a doubly recursive transitive closure program—such programs are not well connected, and as a result yield difference equations that cannot be solved; and programs that use accumulators, where the resultant complexity functions may not be solvable (however, these programs are analyzable provided that the user indicates, in the size measure annotations, which arguments are being used as accumulators and should therefore be ignored). The system also suffers an undesirable loss in precision when dealing with some divide-and-conquer programs, where the sizes of output positions for "divide" predicates may be dependent, while we handle them independently: because of this, the complexity of the `qsort`/2 predicate in Example 5 of Appendix B is inferred to be exponential rather than quadratic.

8 Soundness

In this section we sketch a soundness proof of our method. We call a predicate *size-monotonic* if its argument size functions are monotonic on its input size. Here we assume that all the predicates in the program are size-monotonic.

The size relations in argument size analysis are sound because if $size(t)$ is defined on t and $diff(t_1, t_2)$ is defined on (t_1, t_2), then by definition $size(\theta(t)) = size(t)$ and $diff(\theta(t_1), \theta(t_2)) = diff(t_1, t_2)$ for any substitution θ. Since the transformations applied during normalization replace a term by another equal or larger term, and all the predicates are size-monotonic, the normalization is also sound. Therefore, the soundness of argument size analysis is reduced to the soundness of difference equation solver. A sound difference equation solver can be achieved by always returning an upper bound of the solution to the original equations, as described in Section 6.

In Theorems 4.1 – 4.6, 4.9 and 4.11, all the properties described satisfy the upper bound requirement. Since the computation for relation size and solution size involves only the summations and products of positive quantities, and the expression for an input term is the sum of the expressions for its subterms for recursive clauses (i.e., monotonicity is satisfied), the soundness of number of solutions analysis follows immediately from the soundness of argument size analysis and difference equation solver.

Equation (6), which is used to compute time complexity, involves only summations and products of positive quantities, and the expression for an input term is the sum of the expressions for its subterms for recursive clauses (i.e., monotonicity is satisfied), Consequently, the soundness of time complexity analysis follows immediately from the soundness of argument size analysis, number of solutions analysis and difference equation solver.

9 An Application: Task Granularity Analysis

While logic programming languages offer a great deal of scope for parallelism, there is usually some overhead associated with the execution of goals in parallel because of the work involved in task creation, communication, scheduling, migration and so on. In practice, therefore, the "granularity" of a goal, i.e. an estimate of the work available under it, should be taken into account when deciding whether or not to execute a goal concurrently as a separate task. The cost analysis described in the previous sections can be applied to this problem: the idea is to compute a estimate of the time complexity $\mathsf{T}_p(n)$ of a predicate p on an input of size n at compile time. This expression is evaluated at runtime, when the size of the input is known, and yields an estimate of the work available in a call to the predicate. 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, so that these literals are candidates for concurrent execution. Suppose the expression $\mathsf{T}_q(n)$ 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

$n := size$(`H`);
if $3n^2 < 48$ **then** *execute* `q` *and* `p` *sequentially as a single task*
else *execute* `q` *and* `p` *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

if *size*(`H`) < 4 **then** *execute* `q` *and* `p` *sequentially as a single task*
else *execute* `q` *and* `p` *concurrently as separate tasks*

The ideas described above were tested by experiments on a four-processor Sequent Symmetry, using two different Prolog systems: ROLOG [17] and &-Prolog [14]. Most programs reported some performance improvement due to granularity control: the speedups ranged from 2% to 32% on ROLOG, and from 0% to 29% on &-Prolog. On a few programs, there was a net slowdown (19.5% in one case in ROLOG, and about 16% in one &-Prolog benchmark), because the cost analysis did not take into account the additional cost of maintaining input size information and testing it at each level of recursion. The interested reader is referred to [9] for details. While many of the compilation and code generation issues remain to be worked out in full detail, these experiments suggest that reasonable performance improvements can be obtained from appropriate control of task granularity in parallel logic programs.

10 Related Work

Van Gelder has investigated an approach to reasoning about the constraints between the argument sizes of predicates, using concepts from computational geometry [39]. To reasoning about the termination of procedures, he uses linear inequalities to represent the size relationships among the arguments of a predicate, while for each output argument we represent its size as a function in terms of the input size and the function may be nonlinear.

Lipton and Naughton have applied adaptive sampling techniques to estimate the number of solutions of database query [25]. Their method estimates the query size dynamically at run-time, in contrast our method is a static analysis performed at compile-time.

Much of the work on automatic complexity analysis is in the context of functional programming languages [12, 15, 21, 34, 37, 41, 44]. We extend their work by being able to handle nondeterminism and the generation of multiple solutions via backtracking in logic programs. Kaplan considers the analysis of the average-case complexity of logic programs [18], but his approach cannot handle programs that can produce multiple solutions, thereby excluding many interesting programs.

Knuth [20] and Purdom [33] have exploited random sampling techniques to estimate the efficiency of backtracking algorithms. The primary difference is that their method is dynamic, whereas our method is static.

11 Conclusions

This paper develops a method for (semi-)automatic analysis of the worst-case cost of a large class of logic programs. The primary contribution of this paper is that it shows how to deal with nondeterminism and the generation of multiple solutions via backtracking. Nondeterminism and the ability to backtrack and produce multiple solutions complicates control flow in logic programs considerably, making cost analysis of nondeterministic programs much harder than for traditional languages. It turns out that knowledge about the number of solutions each predicate can generate is required in addition to knowledge about the size relationships between arguments of predicates. The method is sound for predicates whose argument size functions are monotonic on the size of input. Applications include program transformation, automatic program synthesis, software engineering and parallelization of programs.

Acknowledgements

Thanks are due to Y. Deville for many illuminating comments on the material of this paper. We are also grateful to J. Cohen and T. Hickey for constructive discussions on the subject. Comments by the anonymous referees were very helpful in improving the presentation of the paper.

References

[1] J. A. Bondy, "Bounds for the Chromatic Number of a Graph," *Journal of Combinatorial Theory* 7, (1969), pp. 96–98.

[2] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Kista, Sweden, 1988.

[3] J. Chang, A. M. Despain and D. DeGroot, "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis," in *Digest of Papers, Compcon 85*, IEEE Computer Society, Feb. 1985.

[4] B. Char, K. Geddes, G. Gonnet, M. Monagan and S. Watt, *MAPLE: Reference Manual*, University of Waterloo, 1988.

[5] J. Cohen and J. Katcoff, "Symbolic Solution of Finite-Difference Equations," *ACM Transactions on Mathematical Software* 3, 3 (Sept. 1977), pp. 261–271.

[6] J. Darlington, "Program Transformation," *Functional Programming and Its Applications*, J. Darlington, P. Henderson and D. Turner (eds), Cambridge University Press, 1982.

[7] 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.

[8] S. K. Debray and D. S. Warren, "Functional Computations in Logic Programs," *ACM Transactions on Programming Languages and Systems* 11, 3 (July 1989), pp. 451–481.

[9] S. K. Debray, N. Lin and M. Hermenegildo, "Task Granularity Analysis in Logic Programs," *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 174–188.

[10] S. K. Debray and N. Lin, "Static Estimation of Query Sizes in Horn Programs," *Proc. Third International Conference on Database Theory*, Paris, France, December 1990, pp. 514–528.

[11] Y. Deville, personal communication, March 1990.

[12] P. Flajolet, B. Salvy and P. Zimmermann, "Automatic average-case analysis of algorithms," *Theoretical Computer Science*, 79 (1991), pp. 37–109.

[13] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs," *Proc. ACM Symposium Principles of Programming Languages*, San Francisco, California, Jan. 1990, pp. 197–209.

[14] M. V. Hermenegildo and K. J. Greene, "&-Prolog and its Performance: Exploiting Independent And-Parallelism", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, June 1990, pp. 253-270. MIT Press.

[15] T. Hickey and J. Cohen, "Automating Program Analysis," *Journal of the ACM* 35, 1 (Jan. 1988), pp. 185–220.

[16] J. Ivie, "Some MACSYMA Programs for Solving Recurrence Relations," *ACM Transactions on Mathematical Software* 4, 1 (March 1978), pp. 24–33.

[17] L. V. Kalé, *Parallel Architectures for Problem Solving*, PhD Thesis, SUNY, Stony Brook, 1985.

[18] S. Kaplan, "Algorithmic Complexity of Logic Programs," *Logic Programming, Proc. Fifth International Conference and Symposium*, Seattle, Washington, 1988, pp. 780–793.

[19] R. M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (eds), Plenum Press, New York, 1972, pp. 85–103.

[20] D. E. Knuth, "Estimating the Efficiency of Backtracking Programs," *Math. Comp.* 29, (1975), pp. 121–136.

[21] D. Le Métayer, "ACE: An Automatic Complexity Evaluator," *ACM Transactions on Programming Languages and Systems* 10, 2 (April 1988), pp. 248–266.

[22] R. C. T. Lee, C. L. Chang and R. J. Waldinger, "An Improved Program-Synthesizing Algorithm and its Correctness," *Communications of the ACM* 17, 4 (April 1974), pp. 211–217.

[23] N. Lin, "Approximating the Chromatic Polynomial of a Graph," Technical Report 91-5, Department of Computer Science, The University of Arizona, Tucson, Arizona, Feb. 1991.

[24] N. Lin, "Estimating the Number of Solutions for Constraint Satisfaction Problems," Technical Report 92-14, Department of Computer Science, The University of Arizona, Tucson, Arizona, March 1992.

[25] R. Lipton and J. Naughton, "Query Size Estimation by Adaptive Sampling," *Proc. Ninth ACM Symposium on Principles of Database Systems*, April 1990.

[26] M. Maher and R. Ramakrishnan, "Déjà Vu in Fixpoints of Logic Programs," *Proc. North American Conference on Logic Programming*, Cleveland, OH, 1989, pp. 963–980.

[27] Z. Manna and R. J. Waldinger, Z. Manna and R. J. Waldinger, "Toward Automatic Program Synthesis," *Communications of the ACM* 14, 3 (March 1971), pp. 151–164.

[28] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming* 2, 1 (April 1985), pp. 43–66.

[29] P. Mishra, "Toward a Theory of Types in Prolog", *Proc. 1984 IEEE Symposium on Logic Programming*, Atlantic City, 1984, pp. 289-298

[30] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences* 7, (1974), pp. 95–132.

[31] M. Petkovsek, *Finding Closed-Form Solutions of Difference Equations by Symbolic Methods*, PhD Thesis, School of Computer Science, Carnegie Mellon University, 1991.

[32] L. Plümer, *Termination Proofs of Logic Programs*, LNCS 446, Springer-Verlag, Berlin, German, 1990.

[33] P. W. Purdom, "Tree Size by Partial Backtracking," *SIAM Journal of Computing* 7, (1978), pp. 481–491.

[34] L. H. Ramshaw, "Formalizing the Analysis of Algorithms," Report SL-79-5, Xerox Palo Alto Research Center, Palo Alto, California, 1979.

[35] F. A. Rabhi and G. A. Manson, "Using Complexity Functions to Control Parallelism in Functional Programs," Research Report CS-90-1, Department of Computer Science, University of Sheffield, England, Jan. 1990.

[36] C. Pyo and U. S. Reddy, "Inference of Polymorphic Types for Logic Programs", *Proc. North American Conference on Logic Programming*, Cleveland, OH, 1989, pp. 1115-1134.

[37] M. Rosendahl, "Automatic Complexity Analysis," *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, 1989, pp. 144–156.

[38] J. D. Ullman and A. Van Gelder, "Efficient Tests for Top-Down Termination of Logical Rules," *Journal of the ACM* 35, 2 (1988), pp. 345–373.

[39] A. Van Gelder, "Deriving Constraints among Argument Sizes in Logic Programs," *Proc. Ninth ACM Symposium on Principles of Database Systems*, Nashville, TN, April 1990.

[40] K. Verschaetse and D. De Schreye, "Deriving Termination Proofs for Logic Programs, using Abstract Procedures," *Proc. Eighth International Conference on Logic Programming*, Paris, June 1991, pp. 301–315.

[41] B. Wegbreit, "Mechanical Program Analysis," *Communications of the ACM* 18, 9 (Sept. 1975), pp. 528–539.

[42] E. Yardeni and E. Shapiro, "A Type System for Logic Programs", in *Concurrent Prolog: Collected Papers*, vol. 2, ed. E. Shapiro, pp. 211-244.

[43] D. J. A. Welsh and M. J. D. Powell, "An Upper Bound for the Chromatic Number of a Graph and its Applications to Timetabling Problems," *Computer Journal* 10, (1967), pp. 85–87.

[44] P. Zimmermann and W. Zimmermann, "The Automatic Complexity Analysis of Divide-and-Conquer Algorithms," Research Report 1149, INRIA, France, Dec. 1989.

A An Example

This appendix considers in detail the analysis of the `perm` program. This example has been chosen because it is simple, yet shows the interaction of different recursive predicates. The program is as follows:

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

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

We start with the analysis for argument size functions. First, consider the predicate `select/3`, called with the second argument as the input argument. Using the size measure list_length, the size relations for the body literal of the recursive clause are

$$
\begin{aligned}
&body_1[1] \le \mathsf{Sz}^{(1)}_{\texttt{select}}(\underline{body_1[2]}),\\
&body_1[2] \le size(\texttt{Ys}) = head[2] + \mathit{diff}([\texttt{Y}|\texttt{Ys}], \texttt{Ys}) = head[2] - 1,\\
&body_1[3] \le \mathsf{Sz}^{(3)}_{\texttt{select}}(\underline{body_1[2]}),
\end{aligned}
$$

where and hereafter the expressions being substituted for during normalization are underlined for clarity. The size relations for the output positions in the head of the recursive clause are

$$
\begin{aligned}
&head[1] \le size(\texttt{X}) = body_1[1] + \mathit{diff}(\texttt{X}, \texttt{X}) = \underline{body_1[1]},\\
&head[3] \le size([\texttt{Y}|\texttt{Zs}]) = size(\texttt{Zs}) + 1 = body_1[3] + \mathit{diff}(\texttt{Zs}, \texttt{Zs}) + 1 = \underline{body_1[3]} + 1.
\end{aligned}
$$

Normalization then yields the relations

$$
\begin{aligned}
&head[1] \le \mathsf{Sz}^{(1)}_{\texttt{select}}(head[2] - 1),\\
&head[3] \le \mathsf{Sz}^{(3)}_{\texttt{select}}(head[2] - 1) + 1.
\end{aligned}
$$

In addition, from the first clause, we can obtain the following relations as boundary conditions

$$
\begin{aligned}
&head[1] \le size(\texttt{X}) = head[2] + \mathit{diff}([\texttt{X}|\texttt{Xs}], \texttt{X}) = \perp,\\
&head[3] \le size(\texttt{Xs}) = head[2] + \mathit{diff}([\texttt{X}|\texttt{Xs}], \texttt{Xs}) = head[2] - 1.
\end{aligned}
$$

Therefore, we have the following two sets of equations

$$
\mathsf{Sz}^{(1)}_{\texttt{select}}(x) = \mathsf{Sz}^{(1)}_{\texttt{select}}(x - 1),
$$

$$\mathsf{Sz}^{(1)}_{\texttt{select}}(x) = \bot;$$

$$\mathsf{Sz}^{(3)}_{\texttt{select}}(x) = \mathsf{Sz}^{(3)}_{\texttt{select}}(x-1)+1,$$
$$\mathsf{Sz}^{(3)}_{\texttt{select}}(x) = x-1.$$

These equations are solved to obtain $\mathsf{Sz}^{(1)}_{\texttt{select}} \equiv \lambda x.\bot$ and $\mathsf{Sz}^{(3)}_{\texttt{select}} \equiv \lambda x.x-1$.

Next, consider the predicate `perm`/2, called with the first argument as the input argument. The size relations for the body literals of the recursive clause are

$$\begin{aligned}
&body_1[1] \le \mathsf{Sz}^{(1)}_{\texttt{select}}(\underline{body_1[2]}),\\
&body_1[2] \le size(\texttt{X}) = head[1] + \mathit{diff}(\texttt{X},\texttt{X}) = head[1],\\
&body_1[3] \le \mathsf{Sz}^{(3)}_{\texttt{select}}(\underline{body_1[2]}),\\
&body_2[1] \le size(\texttt{Y}) = body_1[3] + \mathit{diff}(\texttt{Y},\texttt{Y}) = \underline{body_1[3]},\\
&body_2[2] \le \mathsf{Sz}^{(2)}_{\texttt{perm}}(\underline{body_2[1]}),
\end{aligned}$$

and the size relation for the output position in the head of the recursive clause is

$$head[2] \le size([\texttt{R}|\texttt{Rs}]) = size(\texttt{Rs}) + 1 = body_2[2] + \mathit{diff}(\texttt{Rs},\texttt{Rs}) + 1 = \underline{body_2[2]} + 1.$$

When normalized, this yields the relation

$$head[2] \le \mathsf{Sz}^{(2)}_{\texttt{perm}}(head[1]-1)+1.$$

In addition, from the first clause, we can obtain the relation $head[2] \le 0$ as a boundary condition. Thus we have the following set of equations

$$\mathsf{Sz}^{(2)}_{\texttt{perm}}(x) = \mathsf{Sz}^{(2)}_{\texttt{perm}}(x-1)+1,$$
$$\mathsf{Sz}^{(2)}_{\texttt{perm}}(0) = 0.$$

These equations can be solved to obtain $\mathsf{Sz}^{(2)}_{\texttt{perm}} \equiv \lambda x.x$, i.e. the size of the output of predicate `perm`/2 is bounded by the size of its input.

This shows how normalization of size relations can be used to track argument sizes. We then continue with the analysis for the number of solutions a predicate can generate. First, consider the predicate `select`/3. Since predicate `select`/3 is recursive, we obtain $\mathsf{Rel}_{\texttt{select}} \equiv \infty$. To compute $\mathsf{Sol}_{\texttt{select}}$, given the size relations and functions computed earlier, the number of bindings possible for variables in the recursive clause can be computed as follows. We first set $\mathcal{N}_{\{\texttt{Y}\}} = \mathcal{N}_{\{\texttt{Ys}\}} = 1$. Using Theorem 4.1, we obtain

$$\begin{aligned}
&O_1 = \mathcal{N}_{\{\texttt{Ys}\}} \times \mathsf{Sol}_{\texttt{select}}(\underline{body_1[2]}) = \mathsf{Sol}_{\texttt{select}}(head[2]-1),\\
&\mathcal{N}_{\{\texttt{X}\}} = \mathcal{N}_{\{\texttt{Zs}\}} = min\{O_1, \mathsf{Rel}_{\texttt{select}}\} = \mathsf{Sol}_{\texttt{select}}(head[2]-1).
\end{aligned}$$

Using Theorem 4.4, the number of possible outputs for the head is

$$instance((\mathtt{X},[\mathtt{Y}|\mathtt{Zs}])) = min\{\mathcal{N}_{\{\mathtt{X}\}} \times \mathcal{N}_{\{\mathtt{Y}\}} \times \mathcal{B}_{\{\mathtt{Zs}\}}, O_1\} = \mathsf{Sol}_{\mathtt{select}}(head[2]-1).$$

Thus we have the equation

$$\mathsf{Sol}_{\mathtt{select}}(x) = \mathsf{Sol}_{\mathtt{select}}(x-1).$$

In addition, from the first clause, we obtain the equation $\mathsf{Sol}_{\mathtt{select}}(x) = 1$. Because the two clauses are in the same mutually exclusive cluster, we sum these two equations, and obtain

$$\mathsf{Sol}_{\mathtt{select}}(x) = \mathsf{Sol}_{\mathtt{select}}(x-1) + 1.$$

This equation can be solved, with boundary condition $\mathsf{Sol}_{\mathtt{select}}(0) = 0$ (accounting explicitly for implicit failure in the base case) to obtain $\mathsf{Sol}_{\mathtt{select}} \equiv \lambda x.x$, i.e. the predicate `select/3` will generate at most x solutions for an input of size x.

Next, consider the predicate `perm/2`. As in the case of predicate `select/3`, we obtain $\mathsf{Rel}_{\mathtt{perm}} \equiv \infty$. To compute $\mathsf{Sol}_{\mathtt{perm}}$, we first set $N_{\{\mathtt{X}\}} = 1$. Using Theorem 4.1, we obtain

$$\begin{aligned}
O_1 &= N_{\{\mathtt{X}\}} \times \underline{\mathsf{Sol}_{\mathtt{select}}(\underline{body_1[2]})} = head[1],\\
N_{\{\mathtt{R}\}} &= min\{O_1, \mathsf{Rel}_{\mathtt{select}}\} = head[1],\\
N_{\{\mathtt{Y}\}} &= min\{O_1, \mathsf{Rel}_{\mathtt{select}}, \mathsf{Rel}_{\mathtt{perm}}\} = head[1];
\end{aligned}$$

$$\begin{aligned}
O_2 &= N_{\{\mathtt{Y}\}} \times \mathsf{Sol}_{\mathtt{perm}}(\underline{body_2[1]}) = head[1] \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1),\\
N_{\{\mathtt{Rs}\}} &= min\{O_2, \mathsf{Rel}_{\mathtt{perm}}\} = head[1] \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1),
\end{aligned}$$

Using Theorem 4.5, Since $\{\mathtt{Y}\} \subseteq \{\mathtt{R},\mathtt{X},\mathtt{Y}\}$, $instance(\mathtt{Y}) = head[1] = O_1$, $instance(\mathtt{Y}) \leq \mathsf{Rel}_{\mathtt{perm}}$, and $instance(\mathtt{Y}) \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1) = head[1] \times \mathsf{Sol}_{perm}(head[1]-1) \leq \mathsf{Rel}_{\mathtt{perm}}$, we obtain

$$instance([\mathtt{R}|\mathtt{Rs}]) = min\{N_{\{\mathtt{R}\}} \times N_{\{\mathtt{Rs}\}}, O_2\} = head[1] \times \mathsf{Sol}_{\mathtt{perm}}(head[1]-1).$$

Thus, we have the equation

$$\mathsf{Sol}_{\mathtt{perm}}(x) = x \times \mathsf{Sol}_{\mathtt{perm}}(x-1).$$

This equation can be solved, with the boundary condition $\mathsf{Sol}_{\mathtt{perm}}(0) = 1$ from the first clause of `perm/2`, to obtain $\mathsf{Sol}_{\mathtt{perm}} \equiv \lambda x.x!$.

The analysis for time complexity now proceeds as follows: first, we consider the clauses defining predicate `select/3`. Using the number of resolutions as the time complexity metric, the difference equations representing the time complexity for the clauses are

$$\begin{aligned}
\mathsf{T}_{\mathtt{select}}(head[2]) &= 1,\\
\mathsf{T}_{\mathtt{select}}(head[2]) &= \mathsf{T}_{\mathtt{select}}(\underline{body_1[2]}) + 1 = \mathsf{T}_{\mathtt{select}}(head[2]-1) + 1.
\end{aligned}$$

Summing these two equations, we obtain

$$\mathsf{T}_{\texttt{select}}(head[2]) = \mathsf{T}_{\texttt{select}}(head[2]-1) + 2.$$

This equation can be solved, with the boundary condition $\mathsf{T}_{\texttt{select}}(0) = 0$, from the implicit failure, to yield

$$\mathsf{T}_{\texttt{select}} \equiv \lambda x.2x.$$

This is then applied to the clauses defining predicate `perm/2`. The difference equations representing the time complexity for the clauses are

$$\begin{aligned}
&\mathsf{T}_{\texttt{perm}}(0) = 1,\\
&\mathsf{T}_{\texttt{perm}}(head[1]) = \underline{\mathsf{T}_{\texttt{select}}(\underline{body_1[2]})} + \underline{\mathsf{Sol}_{\texttt{select}}(\underline{body_1[2]})} \times \underline{\mathsf{T}_{\texttt{perm}}(\mathsf{Sz}^{(3)}_{\texttt{select}}(\underline{body_1[2]}))}\\
&\qquad\qquad\qquad = head[1] \times \mathsf{T}_{\texttt{perm}}(head[1]-1) + 2 \times head[1] + 1.
\end{aligned}$$

These equations can then be solved to obtain the time complexity

$$\mathsf{T}_{\texttt{perm}} \equiv \lambda x.\sum_{i=1}^{x}(3x!/i!) + 3x! - 2.$$

B 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 (procedure calls).

1. Naive Reverse: A very simple recursive program with two recursive predicates:

Input:

```
:- 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:

$\mathsf{Sz}^{(2)}_{\texttt{nrev}} \equiv \lambda x.x;$

$\mathsf{Rel}_{\texttt{nrev}} \equiv \infty;$

$\mathsf{Sol}_{\texttt{nrev}} \equiv \lambda x.1;$

$\mathsf{T}_{\texttt{nrev}} \equiv \lambda x.0.5x^2 + 1.5x + 1.$

$\mathsf{Sz}^{(3)}_{\mathtt{append}} \equiv \lambda\langle x, y\rangle.x + y;$
$\mathsf{Rel}_{\mathtt{append}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{append}} \equiv \lambda\langle x, y\rangle.1;$
$\mathsf{T}_{\mathtt{append}} \equiv \lambda\langle x, y\rangle.x + 1.$

Total Analysis Time: 0.20 secs.

2. Fibonacci: A simple program illustrating double recursion:

Input:

```
:- mode(fib/2,[+,-]).
:- measure(fib/2,[int,int]).
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:
$\mathsf{Sz}^{(2)}_{\mathtt{fib}} \equiv \lambda x.0.447 \times 1.618^x - 0.447 \times (-0.618)^x;$
$\mathsf{Rel}_{\mathtt{fib}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{fib}} \equiv \lambda x.1;$
$\mathsf{T}_{\mathtt{fib}} \equiv \lambda x.1.447 \times 1.618^x + 0.552 \times (-0.618)^x - 1.$

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:
$\mathsf{Sz}^{(2)}_{\mathtt{flatten}} \equiv \lambda x.x;$
$\mathsf{Rel}_{\mathtt{flatten}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{flatten}} \equiv \lambda x.1;$
$\mathsf{T}_{\mathtt{flatten}} \equiv \lambda x.0.5x^2 + x + 0.5.$

Total Analysis Time: 0.21 secs.

4. Towers of Hanoi:

Input:

```
:-  mode(hanoi/5,[+,+,+,+,-]).
:-  measure(hanoi/5,[int,void,void,void,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:

$\mathsf{Sz}^{(5)}_{\texttt{hanoi}} \equiv \lambda x.2^x - 1;$

$\mathsf{Rel}_{\texttt{hanoi}} \equiv \infty;$

$\mathsf{Sol}_{\texttt{hanoi}} \equiv \lambda x.1;$

$\mathsf{T}_{\texttt{hanoi}} \equiv \lambda x.x2^x + 2^{x-1} - 2.$

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 `part`/4 are not independent, and as a result gives a rather pessimistic estimate of the time complexity of `qsort`/2.

Input:

```
:- mode(qsort/2,[+,-]).
:- measure(qsort/2,[length,length]).
qsort([],[]).
qsort([First|L1],L2) :-
     part(First,L1,Ls,Lg),
     qsort(Ls,Ls2), qsort(Lg,Lg2),
     append(Ls2,[First|Lg2],L2).

:- 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).
```

Cost Expressions Inferred:

$\mathsf{Sz}^{(2)}_{\texttt{qsort}} \equiv \lambda x.2^x - 1;$

$\mathsf{Rel}_{\texttt{qsort}} \equiv \infty;$

$\mathsf{Sol}_{\texttt{qsort}} \equiv \lambda x.1;$

$\mathsf{T}_{\texttt{qsort}} \equiv \lambda x. \sum_{i=1}^{x}(i2^{x-i}) + x2^{x-1} + 2^{x+1} - 1.$

$\mathsf{Sz}^{(3)}_{\mathtt{part}} \equiv \lambda x.x;$
$\mathsf{Sz}^{(4)}_{\mathtt{part}} \equiv \lambda x.x;$
$\mathsf{Rel}_{\mathtt{part}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{part}} \equiv \lambda x.1;$
$\mathsf{T}_{\mathtt{part}} \equiv \lambda x.x + 1.$

Total Analysis Time: 0.55 secs.

6. N-Queens: A nondeterministic predicate that can generate multiple solutions via backtracking:

Input:

```
:- mode(safe/2,[+,-]).
:- measure(safe/2,[int,length]).
safe(N,Queens) :- extend(N,N,Queens).

:- mode(extend/3,[+,+,-]).
:- measure(extend/3,[int,int,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,[void,length]).
consistent(_,[]).
consistent(Q,[Q1|Rest]) :- noattack(Q,Q1), consistent(Q,Rest).

:- mode(noattack/2,[+,+]).
:- measure(noattack/2,[void,void]).
noattack(q(X1,Y1),q(X2,Y2)) :-
     Y1 =\= Y2, X is X1-X2, Y is Y1-Y2, Z is Y2-Y1, X =\= Y, X =\= Z.

:- mode(choose/2,[+,-]).
:- measure(choose/2,[int,int]).
choose(N,N) :- N > 0.
choose(N,M) :- N > 0, N1 is N-1, choose(N1,M).
```

Cost Expressions Inferred:

$\mathsf{Sz}^{(2)}_{\mathtt{safe}} \equiv \lambda x.x;$
$\mathsf{Rel}_{\mathtt{safe}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{safe}} \equiv \lambda x.x^x;$
$\mathsf{T}_{\mathtt{safe}} \equiv \lambda x.\sum_{i=1}^{x}(2ix^i) + x^{x+1}/(x-1) - x/(x-1) + x + 2.$

$\mathsf{Sz}^{(3)}_{\mathtt{extend}} \equiv \lambda\langle x, y\rangle.x;$
$\mathsf{Rel}_{\mathtt{extend}} \equiv \infty;$

$\mathsf{Sol}_{\texttt{extend}} \equiv \lambda\langle x,y\rangle.y^x$;
$\mathsf{T}_{\texttt{extend}} \equiv \lambda\langle x,y\rangle.\sum_{i=1}^{x}(2iy^i) + y^{x+1}/(y-1) - y/(y-1) + x + 1.$

$\mathsf{Rel}_{\texttt{consistent}} \equiv \infty$;
$\mathsf{Sol}_{\texttt{consistent}} \equiv \lambda x.1$;
$\mathsf{T}_{\texttt{consistent}} \equiv \lambda x.2x + 1.$

$\mathsf{Rel}_{\texttt{noattack}} \equiv \infty$;
$\mathsf{Sol}_{\texttt{noattack}} \equiv \lambda x.1$;
$\mathsf{T}_{\texttt{noattack}} \equiv \lambda x.1.$

$\mathsf{Sz}^{(2)}_{\texttt{choose}} \equiv \lambda x.x$;
$\mathsf{Rel}_{\texttt{choose}} \equiv \infty$;
$\mathsf{Sol}_{\texttt{choose}} \equiv \lambda x.x$;
$\mathsf{T}_{\texttt{choose}} \equiv \lambda x.2x.$

Total Analysis Time: 0.83 secs

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

Cost Expressions Inferred:

$\mathsf{Sz}^{(2)}_{\texttt{perm}} \equiv \lambda x.x$;
$\mathsf{Rel}_{\texttt{perm}} \equiv \infty$;
$\mathsf{Sol}_{\texttt{perm}} \equiv \lambda x.x!$;
$\mathsf{T}_{\texttt{perm}} \equiv \lambda x.\sum_{i=1}^{x}(3x!/i!) + 3x! - 2.$

$\mathsf{Sz}^{(1)}_{\texttt{select}} \equiv \lambda x.\bot$;
$\mathsf{Sz}^{(3)}_{\texttt{select}} \equiv \lambda x.x - 1$;
$\mathsf{Rel}_{\texttt{select}} \equiv \infty$;
$\mathsf{Sol}_{\texttt{select}} \equiv \lambda x.x$;
$\mathsf{T}_{\texttt{select}} \equiv \lambda x.2x.$

Total Analysis Time: 0.33 secs

8. Eight-Queens: A very different program from the n-queens program shown above, this illustrates how linear arithmetic constraints are handled. This program is only the "test" portion of a generate-and-test program: for this reason, all arguments of `queen/8` are inputs, the number of solutions per input is 1, and the time complexity for `queen/8` is much smaller than its relation size.

Input:

```
:- mode(queen/8,[+,+,+,+,+,+,+,+]).
:- measure(queen/8,[int,int,int,int,int,int,int,int]).
:- domain(queen/8,[1-8,1-8,1-8,1-8,1-8,1-8,1-8,1-8]).
queen(X1,X2,X3,X4,X5,X6,X7,X8) :- safe([X1,X2,X3,X4,X5,X6,X7,X8]).

:- mode(safe/1,[+]).
:- measure(safe/1,[length]).
safe([]).
safe([X|L]) :- noattacks(L,X,1), safe(L).

:- mode(noattacks/3,[+,+,+]).
:- measure(noattacks/3,[length,int,void]).
noattacks([],_,_).
noattacks([Y|L],X,D) :- noattack(X,Y,D), D1 is D+1, noattacks(L,X,D1).

:- mode(noattack/3,[+,+,+]).
:- measure(noattack/3,[int,int,void]).
noattack(X,Y,D) :- X =\= Y, Y-X =\= D, Y-X =\= -D.
```

Cost Expressions Inferred:

$\mathsf{Rel}_{\mathtt{queen}} \equiv 46312;$
$\mathsf{Sol}_{\mathtt{queen}} \equiv \lambda\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\rangle.1;$
$\mathsf{T}_{\mathtt{queen}} \equiv \lambda\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\rangle.83.$

$\mathsf{Rel}_{\mathtt{safe}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{safe}} \equiv \lambda x.1;$
$\mathsf{T}_{\mathtt{safe}} \equiv \lambda x.x^2 + x + 1.$

$\mathsf{Rel}_{\mathtt{noattacks}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{noattacks}} \equiv \lambda\langle x, y\rangle.1;$
$\mathsf{T}_{\mathtt{noattacks}} \equiv \lambda\langle x, y\rangle.2x + 1.$

$\mathsf{Rel}_{\mathtt{noattack}} \equiv \infty;$
$\mathsf{Sol}_{\mathtt{noattack}} \equiv \lambda\langle x, y\rangle.1;$
$\mathsf{T}_{\mathtt{noattack}} \equiv \lambda\langle x, y\rangle.1.$

Total Analysis Time: 5.32 secs.

9. 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. As for the previous example, this program is only the "test" component of a generate-and-test program.

Input:

```
:- mode(color/5,[+,+,+,+,+]).
:- measure(color/5,[int,int,int,int,int]).
:- domain(color/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]]).
color(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,[int,int,int]).
:- domain(c/3,[[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]]).
c(X,Y,Z) :- X =\= Y, X =\= Z.
```

Cost Expressions Inferred:

$\mathsf{Rel}_{\texttt{color}} \equiv 540;$
$\mathsf{Sol}_{\texttt{color}} \equiv \lambda\langle x_1, x_2, x_3, x_4, x_5\rangle.1;$
$\mathsf{T}_{\texttt{color}} \equiv \lambda\langle x_1, x_2, x_3, x_4, x_5\rangle.2.$

$\mathsf{Rel}_{\texttt{c}} \equiv 80;$
$\mathsf{Sol}_{\texttt{c}} \equiv \lambda\langle x_1, x_2, x_3\rangle.1;$
$\mathsf{T}_{\texttt{c}} \equiv \lambda\langle x_1, x_2, x_3\rangle.1.$

Total Analysis Time: 0.21 secs.

10. Precedence Scheduling: A program that illustrates the handling of arithmetic constraints. This program only tests whether the inputs given satisfy the precedence constraints given.

Input:

```
:- mode(schedule/7,[+,+,+,+,+,+,+]).
:- measure(schedule/7,[int,int,int,int,int,int,int]).
:- domain(schedule/7,[0-10,0-10,0-10,0-10,0-10,0-10,0-10]).
schedule(A,B,C,D,E,F,G) :-
     B >= A+1, C >= A+1, D >= A+1, E >= B+5,
     E >= C+3, F >= D+5, F >= E+2, G >= F+1.
```

Cost Expressions Inferred:

$\mathsf{Rel}_{\texttt{schedule}} \equiv 71;$
$\mathsf{Sol}_{\texttt{schedule}} \equiv \lambda\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle.1;$
$\mathsf{T}_{\texttt{schedule}} \equiv \lambda\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7\rangle.1.$

Total Analysis Time: 1.42 secs.