University of California
Santa Cruz

Automated Termination Analysis for Logic Programs

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer and Information Sciences

by

Kirack Sohn

December 1993

The dissertation of Kirack Sohn is approved:

Allen Van Gelder

Phokion G. Kolaitis

Al Kelly

Dean of Graduate Studies and Research

Contents

List of Figures

Automated Termination Analysis for Logic Programs

Kirack Sohn

ABSTRACT

The question of whether logic programs with function symbols terminate in a top-down (Prolog-like) execution is considered. Automated termination analysis is an essential tool for generating a suitable control in modern deductive database systems, such as LDL and NAIL!.

We describe a method of identifying a nonnegative linear combination of bound argument sizes, which (if found) strictly decreases in a top-down execution of logic programs. Testing a termination condition is transformed to a feasibility problem of linear inequalities using duality theory of linear programming. For nontrivial termination proofs, we often need to know the relationship among argument sizes of a predicate. We formalize the relationship by a fixpoint of "recursive transformation" mimicking immediate consequence operator. Since the transformation sometimes fails to finitely converge, we provide some practical techniques to resolve this problem. We also need to indicate which arguments are bound to ground terms during goal reduction. A method for deriving such binding information is described in the framework of abstract interpretation. Positive propositional formula are used to represent groundness dependency among arguments of a predicate.

This methodology can handle nonlinear recursion, mutual recursion, and cases in which no specific argument is certain to decrease. Several programs that could not be shown to terminate by earlier published methods are handled successfully.

Keywords: logic programs, deductive databases, termination, interargument constraints, groundness analysis

Acknowledgements

First of all, I would like to thank Allen Van Gelder, who is a wonderful advisor and a source of inspiration. His unfailing sharpness of thought was a strong support. He taught and influenced me in very many ways: a balance between theory and practice, how to tackle problems, technical writing and presentation skills. More importantly, he supplied me a model for a scientist.

My sincere thanks also go to Phokion Kolaitis, who provided me with a firm basis of logic in computer science. I also thank David Haussler, who served for my oral exam, and Al Kelly, who read my dissertation, and all professors in CIS/CE at UCSC, who taught me computer sciences. I thank Lynne Sheehan for her help and advice. It has been pleasant to ask her help and discuss problems. My colleagues, H. Yoon, T.J. Park, Kjell Post, Yumi Tsuji, discussed my work with me and made insightful observations. They made it enjoyable to stay here at UCSC. Thanks, guys!

The most important support came to me from my family: my wife, Hyunjoo and our wonderful kid, Dong-Gyu, who have kept my mood high and shared the burden of my study. Finally, my deepest appreciation goes to my parents in Korea for their incessant encouragement and patience.

1. Introduction

Logic programming was first introduced in a seminal article of Kowalski in 1974 [Kow74]. It has been successfully used as a tool for several areas including compiler writing, expert system design, natural language processing, hardware design, and knowledge-base design for two decades. The power of logic programming stems from two facts: its formalism is so simple and its semantics is based on clean mathematical logic.

Although it is more relevant in logic programming than in procedural programming, termination analysis was a relatively neglected subject in logic programming areas in contrast to the endeavor made for procedural languages for a long time. The last few years have nevertheless seen a variety of proposed methods. D. De Schreye and S. Decorte studied and collected in their reference list a total of 53 papers published on this topic since 1988 [DSD]. This proliferation is mainly motivated by the practical needs for termination analysis, such as in the area of control generation and program verification. A major concern in this case is automation of the termination proof process [Nai83, UVG88, APP+89, BS89b, DSVB90, Plü90a, Plü90b, Sag91, SVG91]. Another approach concerns the characterization of terminating logic programs. It aims at the treatment of negation as finite failure or the better understanding of decidability issues [AP90, AB91, Dev90]. These rather theoretical works usually provides manually verifiable criteria for termination.

Undecidability of the halting problem, a classical result of theoretical computer science, states that it is undecidable to determine whether or not any program terminates. The proposed method of this thesis only analyzes a sufficient condition; i.e., a program may terminate without our method detecting that fact.

This introductory chapter is organized as follows. In Section 1.1, we describe why termination analysis has practical importance, in particular with regard to its application to deductive databases. In Section 1.2, we briefly describe how the analysis works in general and what treatments are needed in case of termination analysis for logic programs. In Section 1.3, we give the organization of the thesis, serving as a short introduction to each

chapter.

1.1 Why Termination Analysis

There are a number of situations where termination analysis is useful and necessary. One obvious contribution of termination analysis is to the development of reliable software. Motivations for termination analysis in logic programming, however, arise from more or less different purposes.

The well-known Kowalski's formula [Kow79] says a logic program can be viewed as Algorithm = Logic + Control. Since the control component of a logic program is independent of its logic component, various control schemes tailored for the applications can be developed. Standard Prolog interpreters, equipped with depth-first search strategy are one example of control scheme. Unfortunately, this control scheme is unfair; that is, we are not guaranteed to always find a success branch which is logically derivable due to a non-terminating computation. To complement such a shortcoming, termination analysis is useful. With the negation as finite failure rule, the requirement of having a finite computation tree associated with a negated atom is also closely related to termination analysis.

Let us now turn to the application of logic to databases, often called deductive databases. Recently, many prototypes (e.g. Aditi, CORAL, EKS, LDL, LOGRES, LOLA, NAIL-GLUE, RDL, XSB) have been demonstrated in research environments, and a number of applications have been developed using these systems. They seem to be promising alternatives for next generation database systems.

A deductive database is divided into two components: an *extensional* database (EDB), which consists of a set of database facts, and an *intensional* database (IDB), which consists of a set of rules defining how additional relations are computed. IDBs may be evaluated either *top-down* or *bottom-up*. Bottom-up evaluation is often said to be more efficient for logic programs with finite domains (Datalog); however, there is no corresponding claim for general logic programs with function symbols. One of main problems in this

regard is to generate suitable evaluation strategy. *Capture rules* were introduced to decide which evaluation strategy is efficiently applicable to a logic program provided with a goal [UVG85, Ull85, MUVG86]. A capture rule is a statement of the form: "if the rules satisfy such-and-such conditions, then a good evaluation method is such-and-such." A minimal requirement to apply top-down execution method is the guarantee of termination for any query of interest. Ullman and Van Gelder were the first who studied termination analysis in deductive database environments with a strong emphasis on automation of the analysis [UVG88]. Later, the method was enhanced by Plümer [Plü90a, Plü90b] and Van Gelder and Sohn [SVG91]. Brodsky and Sagiv, who studied termination of Datalog programs, were also main contributors along this line [BS89b, Sag91].

1.2 How It Works

In this section, we shall describe the basic structure of termination proof for logic programs. Every termination proof is tantamount to showing the *well-foundedness* (no infinitely decreasing sequence) of a computation path. Consider a simple `while` loop in C.

s_1: `while (x > 0)`

s_2: `x = x - 1;`

The sources of infinite computation in procedural programs are loops. In this example, we have a loop control construct `while`. The valid domain for `x` at s_1 is positive. Suppose the initial value of `x` is positive, say N, otherwise the execution will exit from the `while` loop by the test `x > 0` at s_1. Since the positive value satisfies the test, the value of `x` decrements by 1 at s_2. A simple induction shows that the value of `x` changes over the execution as follows:

$$N, N-1, \ldots, N-K$$

where $N - K > 0$ and $N - K - 1 \leq 0$. The sequence is not infinitely decreasing, showing termination of `while` loop.

Let us take one more example illustrating termination detection technique.

s_3: `while (x < N)`

s_4: `x = x + 1;`

The technique used in the first example is not directly applicable here. Let us associate a function $f(x) = N - x$ with the `while` loop so that we consider the value of $f(x)$ instead of x. Now the domain of $f(x)$ at s_3 is positive and $f(x)$ decreases by 1 over the execution of `while` loop, since x increases by 1. Then the sequence of the values of $f(x)$ over the execution is finite, showing the termination. Essentially, what we should do to prove termination is to find such a function f that its range is nonnegative and its value decreases over execution, although the above two examples are too simple to describe detailed proof techniques.

The idea used in the above procedural programs is universal in termination proof for any language. However, proofs are quite different in technical levels due to the differences in the structure and the attributes of underlying languages. The following is a logic program similar to the first C program. Note there are no loop control structures in logic programs since recursion is the only way to iterate computation.

r_1: `p(0).`

r_2: `p(s(X)) :- p(X).`

The term `s(X)` denotes the successor to `X`, so it can be viewed as `X + 1`. First of all, we can not say whether the program terminates or not without taking a call (or query) to the procedure into account. The reason is logical variables are totally different from procedural ones, which are always bound. Logical variables do not have any values initially but they may be bound to values after the procedures are called. The value is permanent once it is bound (called "single assignment"). Thus, we need to make sure the call binds the variable `X`. For example, a call `p(s(s(0)))`, which binds the logical variable `X`, terminates. But `p(U)` does not terminate. What happens is the call `p(U)` does not bind `X` so the new call `p(X)` repeats the previous call. More difficult problems of this kind arise from unification and partially bound structures.

Another technical difficulty arises from nondeterministic behavior of logic programs. Unlike procedural programs, there is conceptually no single thread of computation in the

```
r1:  perm([], []).
r2:  perm(P, [X|L]) :-
r2.1:         append(E, [X|F], P),
r2.2:         append(E, F, P1),
r2.3:         perm(P1, L).
r3:  append([], L, L).
r4:  append([X|L1], L2, [X|L3]) :-
r4.1:         append(L1, L2, L3).
```

Figure 1.1: A logic program: `permutation`.

execution of logic programs.

Termination analysis can be affected by the underlying mechanism of handling nondeterminism such as backtracking, rule-goal tree expansion. One alternative is to consider all the computation threads for termination simultaneously independently of the mechanisms used. This type of termination is called "universal termination", taken in most published research.

To understand the problems with respect to deriving level mapping function, let us consider a practical logic program in Figure 1.1, which in turn will be used as an example program in this thesis. The readers who are not familiar with the syntax and semantics of logic programs are advised to skip or get back after reading Chapter 2. In Figure 1.1, `perm` succeeds when one argument is a permutation of list elements in the other argument, and `append` succeeds when its third argument is a concatenation of the first two.

We want to prove termination of a query `perm` with the first argument bound. As a matter of fact, we need to prove two subgoals `append` in $r_{2.1}$ and $r_{2.2}$ before the recursive subgoal `perm` in $r_{2.3}$, assuming standard Prolog computation rule selecting leftmost subgoal. The proof in either case of `append` goals is direct because one bound argument in the recursive subgoal is a proper subterm of the bound argument in the same position of the head in r_4. More precisely, a proper subterm ordering is not infinitely decreasing so a sequence

of `append` goals is not infinitely decreasing too because the first (or third) arguments of the goals are ordered by proper subterm ordering. However, the termination of `perm` cannot be proved similarly because there is no direct relationship between `P` and `P1`.

Our approach to this problem is to infer the relationship among arguments in the subgoals, `append` in this case, so as to relate `P` and `P1`. Analysis of `append` rules supplies the fact that the list length of the third argument of `append` is equal to the sum of the length of the first two. (List length is the number of elements in a list.) Omitting some arithmetic, we can conclude the list length of `P1` is less than the list length of `P`, hence proving termination of the query `perm` with the first argument bound. To achieve nontrivial termination proofs, it is essential to derive the relationship among argument sizes. Note two nonrecursive rules r_1 and r_3 play no part in termination analysis.

1.3 Outline of the Thesis

In Chapter 2, we describe basic notations and concepts of logic programming used in this thesis. It also serves as a brief introduction to logic programming and deductive databases.

We assume logic programs are executed in a top-down, left-to-right (Prolog-like) fashion. Internal data are organized using *structures* in Prolog context, which are almost always processed through recursive rules. As in earlier methods, termination of *recursion on structures* can be achieved by showing a *well-foundedness* of goal-reduction graph.

In Chapter 3, we describe termination proof procedure based on argument sizes. This chapter result from collaborative works with A. Van Gelder. We begin this chapter by giving the basic concepts for termination analysis, such as predicate dependency graphs, bound-free adornments, rule-goal graphs, well-founded ordering, and size abstraction of terms. We then describe how to formalize termination condition and transform it to testable form using linear programming theory.

We view argument sizes of derivable facts involving an n-ary predicate as points in the nonnegative orthant of R^n. Our approach is to find a nonnegative linear combination of bound argument sizes of recursive goals which decreases by at least some positive constant

in each recursive call. Existence of such a nonnegative linear combination proves the termination of a recursive goal, since argument sizes are constrained to nonnegativity, so is its nonnegative combination. Our method applies to nonlinear and mutual recursion with a slight extension. We also discuss limitations of our method.

Local variables are logical variables which do not appear in the head, but appear somewhere else in the rule. It is often the case that there is no direct relationship among the argument sizes in a head and those in a recursive subgoal due to the occurrences of local variables. Termination analysis for a certain rule may require the constraints among argument sizes of subgoals before a chosen recursive subgoal.

In Chapter 4, we will describe how to derive those constraints, so-called "interargument constraints" of a predicate. Interargument constraints are essentially a set of constraints which every derivable fact with respect to a predicate satisfies. Research on this topic has recently been studied as a separate task [BS89a, VG91, BS91]. In this thesis, the argument sizes of derivable facts with respect to an n-ary predicate are viewed as a set of points in R^n, which are approximated by their convex hull. Such constraints are formalized by a fixpoint of "recursive transformation" mimicking immediate consequence operator used to define the fixpoint semantics. However, the transformation do not necessarily converge in finitely many iterations. Approximating polycones to their affine hulls provides useful interargument constraints in many practical programs, guaranteeing convergence.

For a class of linear recursive logic programs satisfying so-called "translativeness" property, precise interargument constraints can be obtained via the analysis of structures of recursive transformations.

We need to find extreme points of polycones in a certain representation to verify a fixpoint. Finding extreme points is a time-consuming process. We investigate an efficient method to find them by exploiting information on adjacency of extreme points.

One of the most attractive features of logic programs is that arguments may be used bidirectionally, as input or output at run-time. Groundness analysis is a dataflow analysis to infer whether the arguments of a call are instantiated to ground terms before the call is

made or after the call is completed.

In Chapter 5, we formalize an efficient and precise groundness analysis for logic programs as an instance of an abstract interpretation. Aliased variables are variables which share the same object. They are often a hindrance to precise groundness analysis. Though relational groundness analysis can resolve globally the problems due to aliased variables, it has been considered to be impractical since tables representing the relationship are usually very big, hence hard to handle. In our formalism, the groundness relationships are represented in the form of boolean OR constraints. Since in practice such constraints are almost always in simple computable forms, our method runs efficiently. The abstract domain is exemplified with bottom-up abstract interpretation finding success patterns of a predicate (groundness information after a call is completed).

This chapter is rather independent of the main research thread. In general, groundness information is used by a compiler to effect various optimizations. However, to apply our termination analysis to general logic programs where safety assumption does not hold, groundness analysis is necessary.

Chapter 6 discusses further research directions and concludes the thesis.

2. Logic Programming Preliminaries

Since logic programming was introduced in Kowalski's seminal paper in 1974 [Kow74], logic has been successfully used as a programming language for two decades in various areas of computer sciences, not to mention artificial intelligence and database. Its power stems from simple formalism and rigorous mathematical framework. This chapter introduces the basic concepts of logic programming, which will in turn be used throughout the thesis. Since this chapter serves only a short introduction to logic programming enough to follow the thesis, readers may consult the book by Lloyd [Llo84] for details.

2.1 Logic Programs

We first give an inductive definition of terms. Constants or variables are terms. If f is an n-ary function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. If p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom*. A literal is either an atom or a negated atom. A negated atom is a negative literal; one that is not negated is a positive literal. A clause is a disjunction of literals. A Horn clause is a clause with at most one positive literal. A Horn clause is thus either

1. A single positive literal, e.g., `p(a,b)`, which we regard as a fact,
2. One or more negative literals, with no positive literal, which we regard as a goal, or
3. A positive literal and one more negative literals, which is a rule.

A Horn clause of group (3) in the form of: $\neg q_1 \vee \cdots \vee \neg q_n \vee p$ is logically equivalent to $(q_1 \wedge \cdots \wedge q_n) \rightarrow p$. The latter is natural expression of an inference rule, which can be understood as "If $q_1, \ldots, q_n$ are true, then p is true." Following Prolog syntax, we shall use the notation:

$$p \text{ :- } q_1, \ldots, q_n.$$

for $(q_1 \wedge \cdots \wedge q_n) \rightarrow p$. The atom p is called the head of the rule and $q_1, \ldots, q_n$ is called the body. A logic program (or just a program) is a finite set of rules and facts.

Variables appearing only in the body may be viewed as existentially quantified in the body, while other variables are universally over the entire rule. For example, a logical rule `path` with variables `X`, `Y`, and `Z`:

```
path(X, Y) :- path(X, Z), path(Z, Y).
```

can be understood as "For all `X` and `Y`, there is a path from `X` to `Y` if there exists `Z` such that there is a path from `X` to `Z` and there is a path from `Z` to `Y`." The predicate path defines essentially transitive closure relation.

Following Prolog syntax, Horn clause of group (2) in the form of: $\neg q_1 \vee \cdots \vee \neg q_n$ is denoted

$$:\text{-}\ q_1, \cdots, q_n.$$

It is called a *goal* clause, that is, a clause with no head. Each q_i $(i = 1, \ldots, n)$ is called a *subgoal* of the goal clause.

Technically, an *empty clause*, that is, a clause with empty head and empty body is denoted $\square$. This clause is understood as a contradiction.

In examples, we shall use standard Edinburgh-style Prolog syntax [CM81, SS86]; Variables are denoted by a character string starting with uppercase letters while constants, function symbols, and predicate symbols are denoted by a character string starting with lower case letters. The syntax `[H|T]` (equivalent to `'.'(H,T)`) denotes a list whose head is `H` and tail is `T`. Null (or empty) list is denoted by a constant "`[]`" (read as "nil"). `[a, b, c]` is a list of three elements. `[X, Y | U]` is a list with at least two elements.

In Prolog terminology, a structure is viewed as an uninterpreted function symbol with terms as its arguments. Structures are the only object to organize data local to rules. Such data are almost always processed through recursion (called "recursion on structure"). The set of all rules with the predicate p in the head of the rules is called the *procedure* (or definition) of p. A *ground* term (clause) is one without variables.

Example 2.1: Consider a simple Prolog program defining the transitive closure relation.

r_1: `path(X, Y) :- connected(X, Y).`

r_2: `path(X, Y) :- connected(X, Z), path(Z, Y).`

r_3: `connected(sfo, ny).`

r_4: `connected(ny, paris).`

The fact `connected(sfo, ny)` can be read as "there is a connection from `sfo` to `ny` ." The rule r_1 and r_2 say "for all `X` and `Y`, there is a path from `X` to `Y` if there is a connection from `X` to `Y` or if there exists `Z` such that there is a connection from `X` to `Z` there is a path from `Z` to `Y`". The predicate `connected` defines EDB relation and the predicate `path` defines IDB relation. The rules r_1, r_2 are the procedure of the predicate `path`. □

2.2 SLD Resolution

In this section, we introduce such notions as unification and SLD-resolution, which are central to logic programming.

A *substitution* θ is a finite mapping from variables to terms, and is written as a set of variable/term pairs:

$$\{x_1/t_1, \ldots, x_n/t_n\}.$$

When the substitution is applied to a syntactic object e, it reads informally: "all occurrences of variables $x_1, \ldots, x_n$ in e are replaced by (or are *bound* to) terms $t_1, \ldots, t_n$, respectively." The resulting syntactic object is $e\theta$. A pair x_i/t_i is called a *binding*. Note the notation implies the variables $x_1, \ldots, x_n$ are distinct. If a substitution θ is a one-to-one and onto mapping (i.e. permutation) from its domain and itself, it is called a *renaming* substitution. A *variant* of an expression is obtained by applying a renaming substitution to the expression. If all $t_1, \ldots, t_n$ are ground, then θ is called a ground substitution or instantiation. For a (ground) substitution θ, $e\theta$ is called a (ground) instance of e.

Example 2.2: Atom `p(U,V,Z)` is a variant of `p(X,Y,Z)`, since `p(U,V,Z)` can be obtained by applying a renaming substitution {`X/U,Y/V,U/X,V/Y`} to `p(X,Y,Z)`. Atom `p(a,b,a)` is a ground instance of `p(X,Y,X)`. □

Substitutions can be composed. Given substitutions $\theta = \{x_1/u_1, \ldots, x_m/u_m\}$ and $\sigma = \{y_1/v_1, \ldots, y_n/v_n\}$, the *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set

$$\{x_1/u_1\sigma, \ldots, x_m/u_m\sigma, y_1/v_1, \ldots, y_n/v_n\}$$

by removing those pairs $x_i/u_i\sigma$ for which $x_1 \equiv u_1\sigma$ and removing any pairs y_j/v_j for which $y_j \in \{x_1, \ldots, x_m\}$. A substitution θ is said to be more general than a substitution η if we have $\eta = \theta\sigma$ for some substitution σ. A substitution θ is a unifier of expressions A and B if $A\theta$ and $B\theta$ are syntactically identical. A and B are unifiable if there exists a unifier. A unifier θ of A and B is called a *most general unifier* (or shortly *mgu*) if it is more general than any other unifier of A and B. If A and B are unifiable, then there exists a most general unifier mgu(A, B), which is unique up to variable renaming.

Example 2.3: Suppose we have two atoms `p(X,Y)` and `p(a,U)` and a substitution $\eta =$ {`X/a`, `Y/b`, `U/b`}. They are unifiable since `p(X,Y)` $\theta \equiv$ `p(a,U)` θ. A substitution $\theta =$ {`X/a`, `Y/U`} is more general than θ since there is a substitution $\sigma =$ {`Y/b`,`U/b`} such that $\eta = \theta\sigma$. Indeed, θ is the most general unifier. □

Let G be the goal `:-`$A_1, \ldots, A_m, \ldots, A_n$ and C be the clause A`:-`$B_1, \ldots, B_q$. G' is derived from G and C using mgu θ if the following holds:

1. A_m is an atom, called the selected atom, in G
2. θ is an mgu of A_m and A.
3. G' is the goal `:-` $(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$

An SLD-derivation consists of a (finite or infinite) sequence $G_0 = G, G_1, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of program clauses and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that

1. C_{i+1} has no variables in common with $G_0, \ldots, G_i$ and
2. each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

A single derivation step is called SLD-resolution.

A *computation rule* uniquely determines which atom (called *selected* atom) is selected for every goal in a derivation. The standard Prolog computation rule is to always select the leftmost atom in a goal.

An SLD-derivation is finite if, for some goal G_i in the derivation, there is no next goal. There are two cases. The SLD-derivation is successful if G_i is an empty goal. In this case, it is called an SLD-refutation for the initial goal G. The second case occurs when the derivation is finitely failed. A derivation fails finitely if for some G_i no head of (the variant of) a clause unifies with the selected atom of G_i. The totality of SLD-derivations constructed starting from G under a certain computation rule forms a search space called an SLD-tree. Let P be a program, G a goal and R a computation rule. The SLD-tree for $P \cup \{G\}$ via R is a tree of goals defined as follows:

1. The root is G.
2. Let G' be a node in the tree and A a selected atom of G' under R. The node G' has exactly one descendant for every clause C of P such that A unifies with the head of the variant C' of C. This descendant is derived from G' and C' using an mgu of A and the head of C'.

SLD-resolution is a refinement of the resolution inference rule given by Robinson. An inference rule is correct or sound if only valid formulas are inferred. It is called complete if all valid formulas can be inferred. Resolution is sound and complete for logical formulas in clausal form. Soundness of SLD-resolution is directly implied by soundness of resolution. While SLD-resolution is incomplete for clauses in general, it is complete for Horn-clauses independently of computation rules. An SLD-derivation is *fair* if every atom that appears in the derivation is chosen at some step. The success set of a definite program P is the set of all $A \in B_P$ such that the queried program (P, A) has an SLD-refutation. A success set corresponds to the least Herbrand model of model-theoretic semantics.

Example 2.4: Continuing with Example 2.1, Figure 2.1 shows a finite SLD tree for initial goal `path(sfo, D)` constructed by standard Prolog computation rule (select leftmost atom). while Figure 2.2 shows an infinite SLD tree for the same initial goal constructed by selecting rightmost goal. In Figure 2.1 and Figure 2.2, `p` and `c` stand for `path` and `connected`, respectively. Selected atoms are underlined, and used clauses and performed substitutions are indicated. Used rules are appropriately renamed. This example shows the choice of

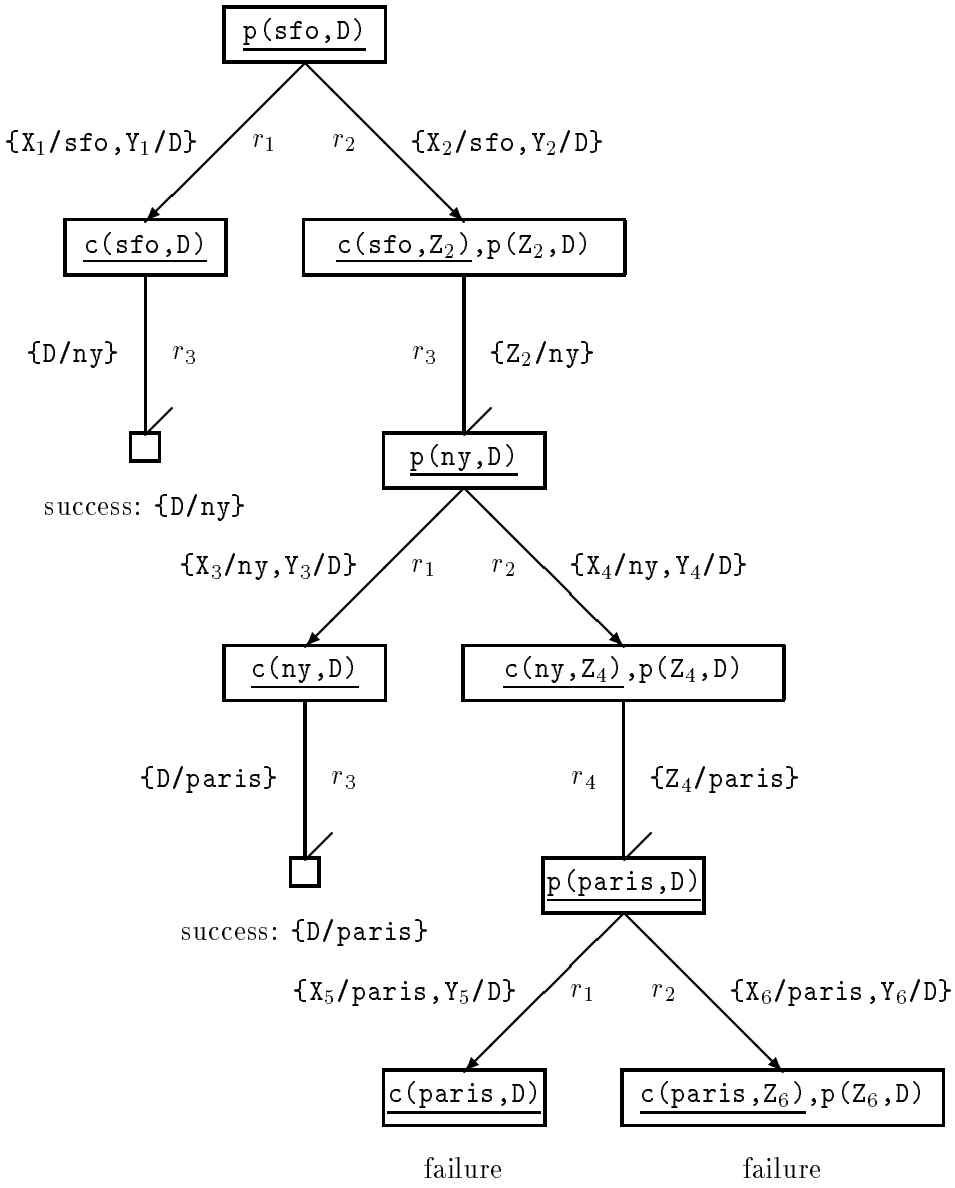


Figure 2.1: An SLD tree via the computation rule selecting the left-most subgoal.

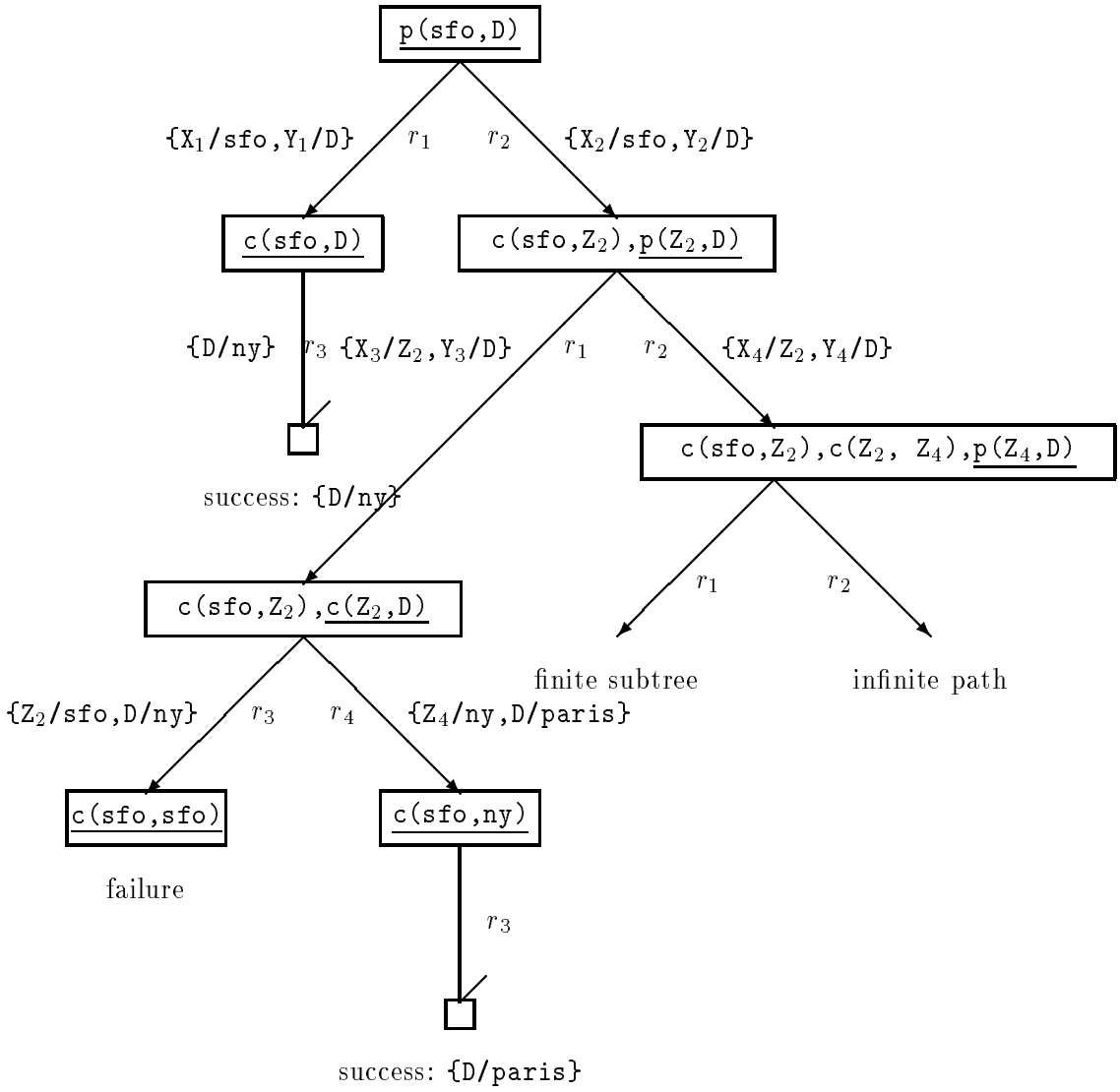


Figure 2.2: An SLD tree via the computation rule selecting the right-most subgoal.

computation rule has a great bearing on the size and structure of the corresponding SLD-tree. However, in both SLD trees, we find the same success set {`path(sfo, ny)`, `path(sfo, paris)`} by the independence of computation rule. □

We now turn to the problem of searching SLD-trees to find success branches. Two well-known search strategies are depth-first and breadth-first. Depth-first search strategy is employed in most Prolog implementations since it can be implemented very efficiently. However, the efficiency is achieved on the sacrifice of fairness of SLD-resolution. As shown in Figure 2.2, once we get into the infinite branch, the other success branches will never be explored. In this thesis, SLD-resolution equipped with leftmost computation rule and depth-first search strategy like standard Prolog system is assumed, unless explicitly stated otherwise.

With Horn clauses we can express what is true but not what is false. That is, SLD-resolution can not deduce negative information. There are some examples where it is natural to require that also negative information can be deduced. One possibility to do that is to conclude that a fact $\neg A$ is true if A is not proved from a program P. This rule is usually called "closed world assumption." Unfortunately, it is not an effective reasoning method, since the set of negative facts derived by closed world assumption is not necessarily recursively enumerable. A way out of this problem is to adopt some more restrictive form of unprovability, namely, *finite failure*. An SLD-tree is finitely failed if it is finite and every branch is failure. So we conclude $\neg A$ if the SLD tree associated with a ground atom A is finitely failed. SLD-resolution together with negation as finite failure is called SLDNF-resolution. The algorithm in Figure 2.3 describes how SLDNF-refutation works.

2.3 Fixpoint Semantics

The set of ground terms which can be constructed from the function symbols occurring in a program P is called Herbrand Universe U_P of P. The set of ground atomic formulas which can be constructed from the predicate and function symbols occurring in P is called the Herbrand Base B_P of P.

```
function SLDNF(query: a set of literals {L_1,...,L_m}): boolean
begin
    if query is empty then return TRUE
    else
    begin
        select a literal L_i from query such that
        L_i is a positive literal R(t_1,...,t_n) or a negative ground literal ¬P.
        if L_i is positive then
        begin
            for all data base clause R(t'_1,...,t'_n) ← L'_1,...,L'_p
                if L_i and R(t'_1,...,t'_n) unify then
                begin
                    compute m.g.u. θ.
                    if SLDNF({L_1,...,L_{i-1}, L'_1,...,L'_p,L_{i+1},...,L_m}θ) then return TRUE
                end
            return FALSE
        end
        else /* L_i is negative ground */
            if SLDNF({P}) then return FALSE
                /* if the return value is TRUE, an SLDNF-refutation is found for ← P,
                    else there is a finitely failed SLDNF-tree */
            else return SLDNF({L_1,...,L_{i-1},L_{i+1},...,L_m})
    end
end
```

Figure 2.3: SLDNF-Refutation Procedure.

The meaning (model) of a logic program can be obtained by a fixpoint of a certain function T_P called "immediate consequence operator" with respect to a program P:

$$T_P(I_k) = I_{k+1}$$

This function, which is a total mapping from the powerset of Herbrand Base to itself, is defined as follows:

$$T_P(I) = \{head \mid (head \leftarrow body) \in G(P) \text{ and } body \text{ is true in } I\}$$

where $G(P)$ is a set of ground instantiations of the rules in P.

T_P is monotone so it reaches a fixpoint by Tarski's classical fixpoint theorem. Starting with empty interpretation $\emptyset$, T_P arrives at a least fixpoint; however, depending on starting interpretations, there exist numerous fixpoints. The transformation may need infinite iterations before reaching a fixpoint. Details can be found in [Llo84, Hog90].

Basically, our algorithm to infer interargument constraints simulates the immediate consequence operator T_P.

2.4 Deductive Databases

One interpretation of logic is the database interpretation. Here a logic program is regarded as a database. We thus obtain a very natural and powerful generalization of relational databases, which correspond to logic programs solely of ground unit clauses. The concept of logic as a uniform language for data, programs, queries, views and integrity constraints has great theoretical and practical potential.

In the deductive database environment, facts are usually stored in the database and the predicate is said to define an extensional database (EDB) relation. One computed by logical rules is called an intensional database (IDB) relation. A Datalog program is a logic program which contains no function symbols of arity > 0.

Unlike the approach taken in logic as a programming language, in database context, we are interested in finding all facts implied by the logical rule. Therefore, to have operations on Datalog programs make sense, we need to make sure the relation with respect to a

predicate is finite. One simple way to solve the problem is to put syntactic restrictions on rules so that no infinite relations are created. The source of infiniteness is a variable that appears only in the head of a rule. For example, consider the following logic program:

r_1: `loves(X, Y) :- subject(X).`

r_2: `subject(Jesus).`

The `subject` is an EDB relation, and the intended meaning of `loves(X, Y)` is "`X` loves `Y`." The rule r_1 defines an infinite relation since the second argument Y of `love` is universally quantified and does not appear in the subgoal. That is, the relation can be interpreted as "`Jesus` loves whatever is in the universe."

A rule is defined to be *safe* if all variables in the head also appear in the body of the rule. If the relation of subgoals are finite, then the relation of the predicate of the head is also finite, since all the values for the variables in the head come from the set of values in the variables in the subgoals, which is finite. The rule r_1 in the above program is not safe since Y does not appear in the body.

Logical rules can be evaluated top-down or bottom-up. Top-down evaluation is believed to be more efficient for logical rule with function symbols. Our research focuses on whether or not top-down evaluation terminates. Since we are interested in finding all solutions of a query in database environment, we need to guarantee all the branches of the SLD-tree are finite. This type of termination is called *universal termination*. Note the search strategy is irrelevant, since we examine the whole SLD-tree with respect to a given query. However, the computation rule is important, as shown in Example 2.4. Many deductive database implementations employ dynamic subgoal ordering. Although we assume a left-to-right computation rule, our termination analysis technique can also be applied to an evaluation strategy with subgoal ordering.

3. Termination Detection

3.1 Basic Concepts

3.1.1 Predicate Dependency Graphs

For the effective analysis of logic programs, we often need to know the way predicates in a logic program depend on each other. To do so, we construct a digraph whose nodes are predicates. An arc from node p to node q is drawn if p is the predicate of the head of a certain rule and q is the predicate of its subgoal. Intuitively, q supports the derivation, or solution, of p. A logic program is recursive if its predicate dependency graph has one or more cycles.

From the digraph, we can identify the strongly connected components (SCCs), and the partial order induced upon them. A *recursive subgoal* is one whose predicate is in the same SCC as the head of the rule. An SCC with more than one predicate is said to have *mutual* recursion. If each rule in an SCC has at most one recursive subgoal, then the recursion in this SCC is said to be *linear*. In fact, most recursions in logic programs are linear.

3.1.2 Modular Termination Analysis

In general, modularity reduces the task of termination analysis significantly. To do so, we shall analyze one SCC at a time starting from the lowest level of the partial order. Successful termination analysis lets us march to upper levels. Usually the analysis of an SCC requires the information on the predicates in lower SCCs, such as "interargument constraints".

Example 3.1: Consider a rather unmotivated example below:

r_1: `p(f(X)) :- q(X).`

r_2: `q(f(X)) :- r(X).`

r_3: `r(f(X)) :- p(X), s(X).`

r_4: `s(f(X)) :- s(X).`

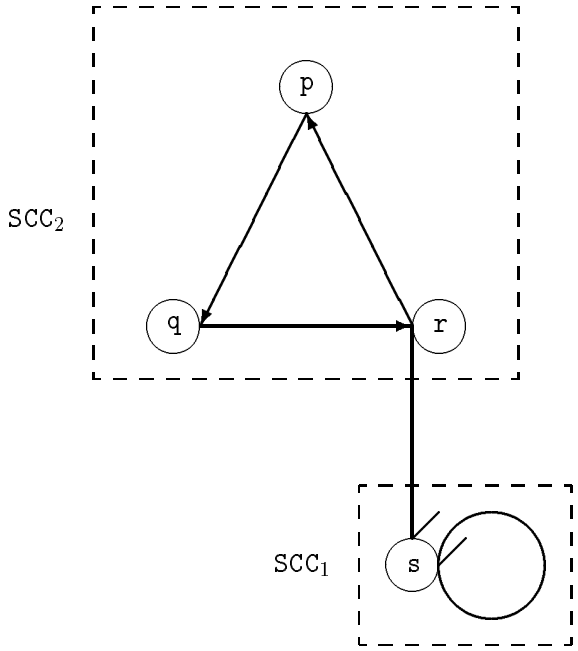


Figure 3.1: A predicate dependency graph.

r_5: `s(0).`

Let us draw a predicate dependency graph for this program. We have four predicates `p, q, r, s` in the program so we draw four nodes. In r_1, `p` depends on `q` so we draw an arc from `p` to `q`. By doing the same thing for other rules, we have the predicate dependency graph for the above logical rules, as shown in Figure 3.1. Using algorithms to find SCCs, which can be found in many algorithm textbooks, for example [CLR90], we identify two SCCs, $\texttt{SCC}_1$ consisting of the predicate `s`, and $\texttt{SCC}_2$ consisting of the predicates `p, q, r`. $\texttt{SCC}_1$ is linear recursive, while $\texttt{SCC}_2$ is mutually recursive. Since $\texttt{SCC}_1$ is in the lower level than $\texttt{SCC}_2$ in the partial order induced on SCCs, we do termination analysis of $\texttt{SCC}_1$ before $\texttt{SCC}_2$. If it is shown to terminate, we move to $\texttt{SCC}_2$, otherwise we stop there by reporting

"we can not show termination." □

3.1.3 Bound-Free Adornments

Unlike procedural or functional programs, logic programs are bidirectional, that is, arguments to a procedure may be used as input, output, or both. This bidirectional use of arguments makes it difficult to analyze logic programs for termination (possibly various data-flow analyses, too). In termination analysis, we are mainly interested in how input data structures are processed during recursive calls. Therefore we need to indicate which arguments are used as input. Since there is no precise dichotomy of input/output for arguments, bound arguments (containing no variables) can be viewed as input. We shall use the same notations as in [Ull89] for bound-free adornments and rule-goal graphs in the following section.

An argument is said to be *bound* (or *ground*) if all the variables in the argument are bound, otherwise *free*. [1] Whenever we consider a goal in the execution of logic program, say

$$p(t_1, \ldots, t_n)$$

that goal is associated with a binding relation for some subset of p's arguments so as to indicate which arguments are bound and which are not by an adornment, or binding pattern. The adornment is a string of b's and f's of length n for n-ary predicate p. If i-th symbol of the adornment is b, then the i-th argument of p is *bound*. If the i-th symbol of the argument is f, then the i-th argument is *free*. A class of goals whose binding pattern is α is denoted p^{α}.

Suppose we have a goal `append` whose first and second arguments are bound and the third free. Then its adorned predicate is denoted $\texttt{append}^{bbf}$. Of course, we may have a goal `append` with a different binding pattern, say, $\texttt{append}^{ffb}$.

[1] In logic programming world, the terminology "ground" is usually used instead of "bound" used in deductive database world.

We also need to indicate the bound/free status of variables in a rule to construct a rule-goal graph, which is introduced in the next section. Assuming that all the variables in a subgoal become bound after the subgoal is processed successfully, we have two simple rules to indicate which variables are bound and which free:

1. A variable that appears in the bound argument of the head is bound before processing any subgoals.
2. After processing a subgoal G_i during rule evaluation, a variable that appears anywhere in G_i or was bound before processing G_i is bound.

Note that all the variables in a rule are bound after processing the last subgoal. Since we concern several different situations during rule evaluation, we indicate them using subscripted rule number. When r_j is the rule considered, the situation, before processing any subgoals, is denoted $r_{j.0}$, while the situation after processing the ith subgoal is $r_{j.i}$. An adornment for a rule indicates which variable are bound and which are free. The notation we use for a *rule adornment* is a superscript of the form $[X_1, \ldots, X_m | Y_1, \ldots, Y_n]$, where X's are bound and the Y's are free.

Example 3.2: Consider the recursive path rule r_1.

$$r_1 : \texttt{path(X, Y) :- arc(X, Z), path(Z, Y)}$$

Suppose `path` is called with the binding for the first argument. Then in the rule before consideration of any subgoals, only X is bound. We represent this fact by the adorned rule $r_{1.0}^{[X|Y,Z]}$. The first subgoal `arc(X,Z)` provides a binding for `Z`, so the situation after considering the first subgoal is represented by $r_{1.1}^{[X,Z|Y]}$.

3.1.4 Rule-Goal Graphs

We can represent the patterns of binding that occur in top-down evaluation by a finite structure called a rule/goal graph. Suppose we are given a set of Horn-clause rules and a query goal. If p is the predicate of the query, and α is the adornment that has b whenever the query specifies a value for the corresponding argument and has f whenever no value is

specified, then we begin construction of the rule/goal graph for this query with the node p^{α}.

We then consider each node in the rule/goal graph and expand it according to the following rule. As we expand, we add goal nodes, which are adorned predicates, and rule nodes which are nodes representing a rule and some number of the subgoals for that rule. Using the notation described above, we have r_0, with an adornment, to represent rule r before considering any subgoals, and r_i, with an adornment, to represent rule r after considering its first i subgoals.

1. A goal node with EDB predicate has no successors.
2. A goal node that is an IDB predicate p with an adornment α has a successor rule node $r_{j.0}$ for each rule r_j whose head predicate is p.
3. Consider a rule node $r_{j.i}$ and suppose $q(t_1, \ldots, t_k)$ is the $i+1$st subgoal of r_j. As successors of the rule node $r_{j.i}$ we draw a goal node q^{β} with an appropriate adornment β and a rule node $r_{j.i+1}$ unless the $i+1$st subgoal is the last. If the goal node q^{β} already exists, it is simply connected as a successor.

Rule nodes are also adorned in the way we described in the previous section. Figure 3.2 shows a rule-goal graph constructed by a query goal $\texttt{perm}^{bf}$ and a logic program `perm` in Figure 1.1.

The construction of rule-goal graphs was described under the assumption that all the variables in a subgoal become bound after the subgoal is processed successfully. This assumption is not so realistic. The construction may be imprecise since we do not consider the problems with respect to aliased variables. These problems can be resolved by the groundness analysis presented in Chapter 5. The complexity of rule-goal graphs was studied in [UV88]. Theoretically, its size is exponential of input programs, but is known to be practically linear.

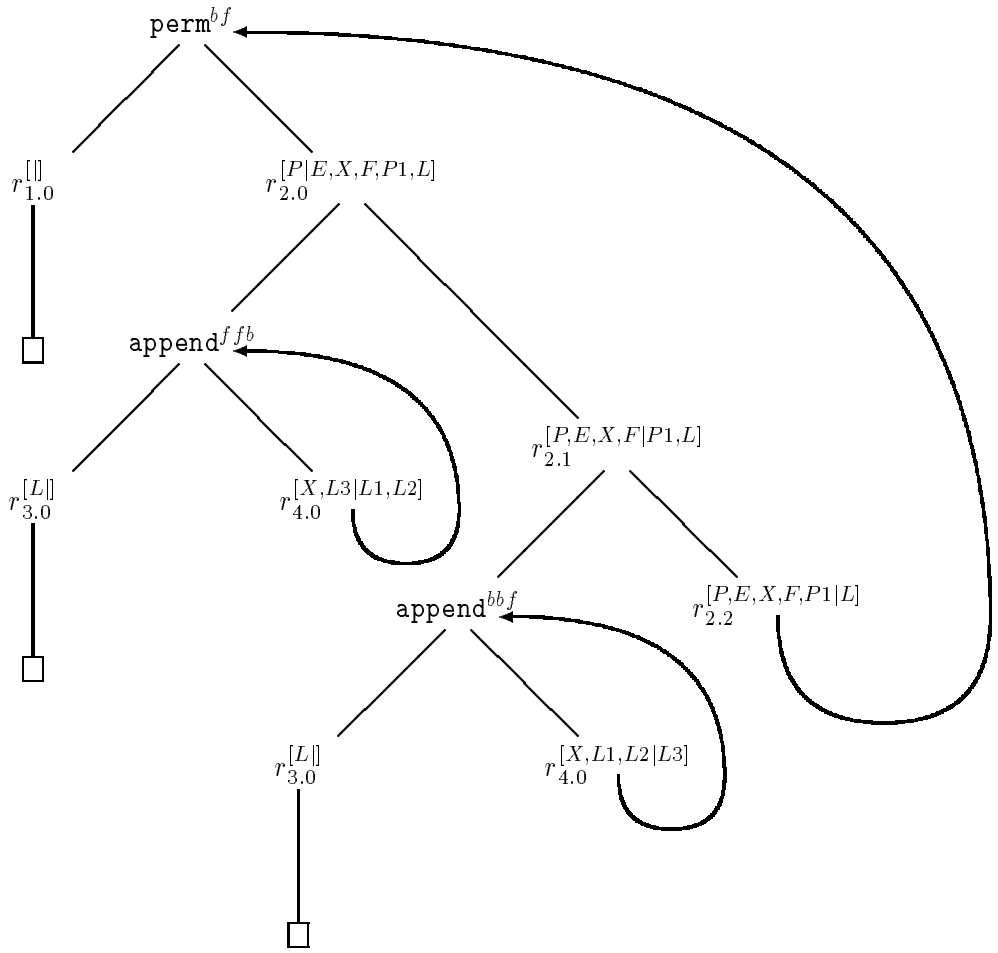


Figure 3.2: A rule-goal graph for the adorned query perm^{bf}.

3.1.5 Adorned Programs

In the previous section, we described how to construct rule-goal graphs. For termination analysis, we will consider *adorned programs* which are direct translation of rule-goal graphs.

Example 3.3: Consider again a logic program `perm` in Figure 1.1. Its rule-goal starting from a query goal $\texttt{perm}^{b}f$ is constructed in a way described in the previous section and is illustrated in Figure 3.2. Annotating the predicates of a program with the binding patterns in its rule-goal graph, we have a specialized version of the program conforming to an query goal. The following adorned program is given by the rule-goal graph in Figure 3.2.

r_1: $\texttt{perm}^{bf}$`([], []).`

r_2: $\texttt{perm}^{bf}$`(P, [X|L]) :-`
`append`^{ffb}`(E, [X|F], P),`
`append`^{bbf}`(E, F, P1),`
`perm`^{bf}`(P1, L).`

r_3: $\texttt{append}^{bbf}$`([], L, L).`

r_4: $\texttt{append}^{bbf}$`([X|L1], L2, [X|L3]) :-`
`append`^{bbf}`(L1, L2, L3).`

r_5: $\texttt{append}^{ffb}$`([], L, L).`

r_6: $\texttt{append}^{ffb}$`([X|L1], L2, [X|L3]) :-`
`append`^{ffb}`(L1, L2, L3).`

Procedure `perm` is always called with the first argument bound and the second free. Procedure `append` is called in two different patterns, namely, $\texttt{append}^{bbf}$ and $\texttt{append}^{ffb}$, So we have two versions of `append` procedure. These two versions are considered as different procedures and separate termination proofs will be tried. The specialized version of a program annotated with the binding patterns will be called a *adorned program*. Each procedure is associated with a unique single binding pattern. □

3.1.6 Subterm Ordering

We first give some basic concepts on ordering.

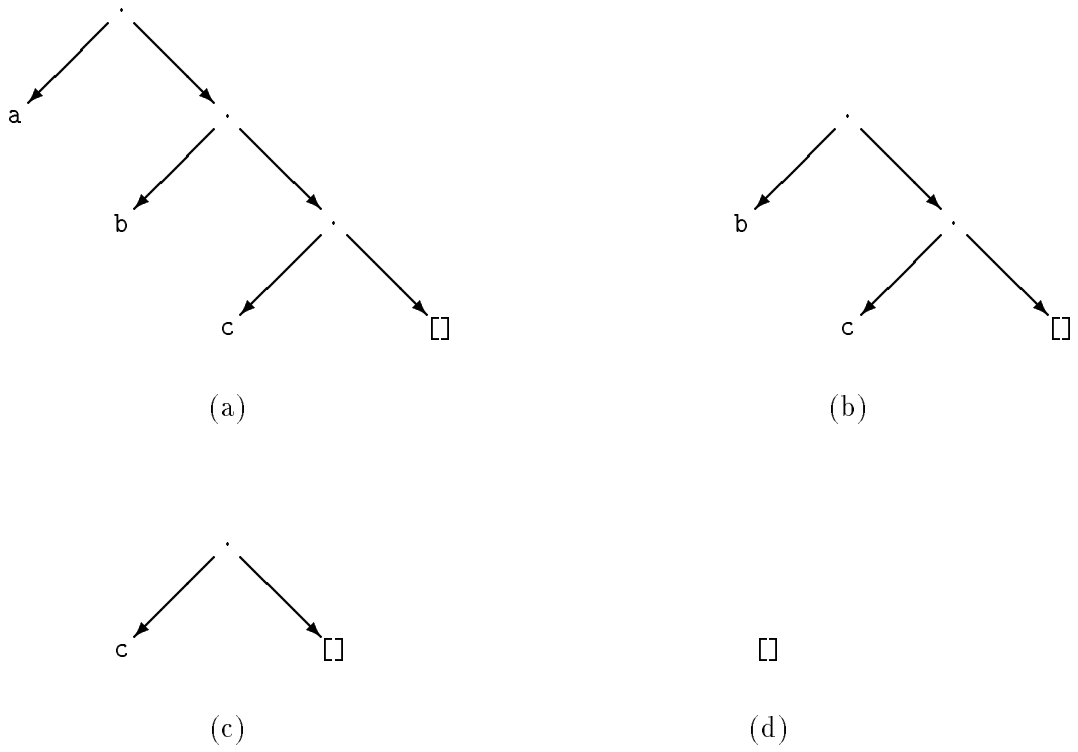


Figure 3.3: The first argument of `append` goal over execution. (a) at initial call. Termsize = 6. (b) after one recursive call. Termsize = 4. (c) after two recursive calls. Termsize = 2. (d) after three recursive calls. Termsize = 0.

Definition 3.1: A *partially ordered set* $(P, \geq)$ is a reflexive, antisymmetric, and transitive binary relation $\geq$ on a set P. A *strictly partially ordered set* $(P, >)$ is an antisymmetric, and transitive binary relation $>$ on a set P. A partially ordered set P is *totally ordered* if for any two elements a and b in P, either $a \geq b$ or $b \geq a$. A strict partial ordering $>$ on a set P is *well-founded* if for any element s_1 in P, there is no infinitely decreasing sequence $s_1 > s_2 > s_3 > \cdots$ of elements $s_1, s_2, s_3, \ldots$ of P. For example, the relation $>$ on the set of natural numbers $\mathbf{N}$ is well-founded while it is not well-founded on the set of integers $\mathbf{Z}$. □

A logic program terminates if and only if the trace (proof tree or SLD tree) of the execution of the program is finite. For example, try to prove termination of a query

append([a,b,c], L1, L2). As shown in Figure 3.3, the first argument reduces to nil during the recursive call of the initial query, so the call append([], U, V) does not unify with the recursive rule. So the trace is finite. However, one does not want to construct a trace explicitly because the construction may be a nonterminating task and it is impossible to handle every specific input. Instead, we consider a class of queries classified by binding patterns. Therefore, one may show a trace must be finite by syntactic properties of clauses that are possibly called. We usually exploit the *well-foundedness* of an ordering over goals. More precisely, we show that the order associated with the states in the trace is well-founded, and between every two consecutive states, the order decreases. As an example of well-founded ordering, let us take (proper) subterm ordering on ground terms.

Example 3.4: Continuing with Example 3.3, consider a query $\texttt{append}^{bbf}$. Suppose $A >_{subterm} B$ if B is a subterm of A and $\texttt{append}(A_1, A_2, A_3) >_{append} \texttt{append}(B_1, B_2, B_3)$ if $A_1 >_{subterm} B_1$. Clearly, subterm ordering is well-founded since there are only finite number of proper subterms for a ground term.

In adorned procedure r_3, r_4 for append, the first argument L1 of the recursive subgoal $\texttt{append}^{bbf}$(L1, L2, L3) is a subterm of the first argument [X|L1] of the head $\texttt{append}^{bbf}$([X|L1], L2, [X|L3]). That is, [X|L1] $>_{subterm}$ L1. By the definition of $>_{append}$,

$$\texttt{append([X|L1], L2, [X|L3])} >_{append} \texttt{append(L1, L2, L3)}.$$

For every two consecutive goals of the SLD-tree, if $\texttt{append}^{bbf}$([X|L1], L2, [X|L3]) is a goal, then $\texttt{append}^{bbf}$(L1, L2, L3) is its immediate descendant by the unification. Therefore, the path from the initial query $\texttt{append}^{bbf}$ to empty goal is finite, hence proving termination of $\texttt{append}^{bbf}$. □

Subterm ordering is enough to detect termination for many simple examples. However, it is often the case that there is no syntactic relationship between arguments in head and those in recursive subgoals since arguments are processed through subgoals before the recursive ones (See Example 3.7). It is difficult and unnecessary to relate terms by subterm ordering. By this reason, we will introduce size abstraction of terms in the following section.

3.1.7 Abstractions of Terms

In a logical rule, the body of the rule may contain variables that do not appear in the head. We usually refers to them as *local variables*. A subterm ordering is not adequate to detect termination of logic programs containing local variables since there is no syntactic relationship between two terms. In this section we describe size abstractions of terms to give ordering on ground terms.

Size abstraction is a mapping from terms to natural numbers. Ullman and Van Gelder [UVG88] were the first to introduce an abstraction function for terms, motivated by the fact that subterm ordering is incapable of handling problems due to local variables. They used listsize abstraction as a basis for the well-founded ordering in a top-down termination analysis, which is defined in the following.

Definition 3.2: List size is defined as follows:

$$listsize(t) = \begin{cases} 1 + listsize(t_n) & \text{if } t = f(t_1, ..., t_n) \\ t & \text{if } t \text{ is a } \textit{variable} \\ 0 & \text{if } t \text{ is a } \textit{constant} \end{cases}$$

Listsize is informally the number of edges in the rightmost path in the tree representation of a ground term. For terms containing logical variables, a real variable X constrained to nonnegativity is associated with each logical variable `X`. For instance, the listsize size of `f(a,g(X),X)` is $1 + X$. □

Listsize abstraction is capable of detecting termination of structural recursions relying on rightmost paths such as lists. However it often fails when tree-like data structures are used.

Example 3.5: Consider a program visiting a binary tree in depth-first manner.

```
traverse(nil).
traverse(t(L, R)) :- traverse(L), traverse(R).
```

`t(L,R)` can be thought as a binary tree whose left subtree is `L` and right subtree is `R`. Suppose that `traverse` is called with its argument bound. By constructing a rule-goal graph from

`traverse`^{b}, we know `traverse` is always called with its argument bound. Therefore we can measure its listsize and compare their sizes to analyze termination. In order to show termination of `traverse`^{b}, we have to show every pair of head and recursive subgoal has well-founded ordering. Consider the head and the first subgoal. The listsize of the head argument is $1+R$ and the listsize of the argument in the second subgoal is R, so $1+R > R$. That proves the termination of the cycle from the head and the first subgoal. Now consider the cycle from the head and the second subgoal. the listsize of the head argument is $1+R$ and the listsize of the argument of the second subgoal is L, hence no relationship between two argument sizes. So it fails to prove termination of `traverse`^{b}. □

In spite of this shortcoming shown in the above example, the basic reason why [UVG88] used listsize is that their inference algorithm for interargument constraints can only handle the relationship between two logical variables, each variable representing a listsize of an argument. This restriction was lifted by Plümer who used a "linear norm" as abstraction function. Later, Sohn and Van Gelder also provided a method resolving such problems using structural term size. We will describe the method in detail in the following section.

Definition 3.3: Structural term size is defined as follows:

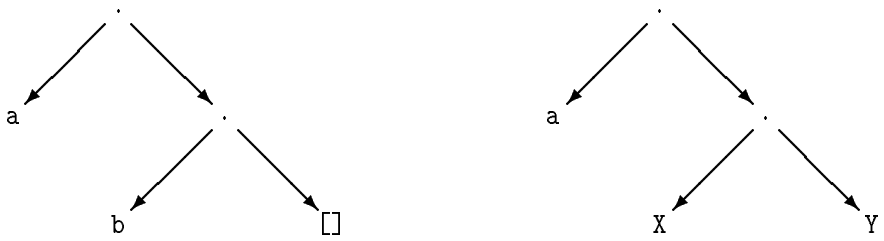
$$termsize(t) = \begin{cases} n + \sum_{i=1}^{n} termsize(t_i) & \text{if } t = f(t_1, ..., t_n) \\ t & \text{if } t \text{ is a } \textit{variable} \\ 0 & \text{if } t \text{ is a } \textit{constant} \end{cases}$$

It is informally the number of edges in the tree representation of a ground term. For terms containing logical variables, a real variable X constrained to nonnegativity is associated with each logical variable `X`. Note that the structural term size of such terms is a linear polynomial of those real variables whose coefficients are positive integers. More formally,

□

This measure is called "structural" in that every portion of a term structure contributes to the size measure. Figure 3.4 shows the listsize and termsize of two terms.

Example 3.6: Continuing with Example 3.5, termsizes of `t(L, R)`, `L` and `R` are $2+L+R$, L, and R, respectively. The termsize of recursive subgoal is less than that of head since



(a) (b)

Figure 3.4: (a) tree representation of `[a,b]`, termsize = 4 and listsize = 2. (b) tree representation of `[a,X|Y]`, termsize = $4 + X + Y$ and listsize = $2 + Y$.

$2 + L + R > L$ and $2 + L + R > R$. So the goals over two cycles are well-founded, showing termination of `traverse`^{b}.

3.2 Related Work

Most termination detection methods attempted to prove that the goal reduction graph is well-founded. In order to show termination, we sometimes need to know the relationship among argument sizes of certain predicates. We describe several approaches developed in earlier work.

Naish's method [Nai83] was based on the measure of "proper subterm", which gives a partial order on logical terms. His idea was that if there exists a subset of bound arguments for each predicate such that no bound argument increases and some bound argument decreases in any recursive rule of the predicate, the recursive subgoals are well-founded. A typical procedure for his method was one that merges two lists, maintaining order. Depending on which rule applies at a particular step, either the first argument decreases, or the second one does. He gave an algorithm determining whether some subset of

the bound arguments of each predicate existed such that each recursive call was guaranteed to reduce one or more elements of the subset without changing others. This exponential time algorithm was made semi-polynomial by Sagiv and Ullman [SU84].

Ullman and Van Gelder [UVG88] proposed a method to test top-down termination with a view of testing the applicability of top-down capture rules in a knowledge-base system. They introduced a measure of terms called "list size", which corresponds to the number of edges in rightmost path in a tree representation of terms. This measure corresponds to the length of a list; however, it is less natural for general structural terms such as binary trees, leading to failure to detect termination. They gave an algorithm for identifying and testing a restricted class of rules, those satisfying so-called "uniqueness" property, which guarantees that the set of inequality constraints between two arguments is unique. Termination of such rules could be tested in polynomial time in the number of predicate symbols and the number of rules. The idea of using interargument constraints (in the form of inequality between two argument positions) was first introduced. Candidates of interargument inequalities sufficient for termination were generated, and then their validity was tested.

Brodsky and Sagiv studied inference of interargument constraints called "monotonicity constraints" [BS89a] and termination detection [BS89b] in logic programs without function symbols (Datalog programs) as two separate tasks.

A monotonicity constraint is a statement that one argument of a predicate is greater than another in all derivable (or given, for EDB) facts for that predicate. Such constraints were the backbone of their termination detection method called "argument mapping". Argument mappings can be viewed as monotonicity constraints between an argument in a head and one in a subgoal, and composing argument mappings as unifications. Although monotonicity constraints were not powerful enough to characterize the property of derivable facts, their method could detect termination due to certain unification issues, such as "occur check". Their exponential-time algorithm could test *necessary* and sufficient condition for termination on finite database relations, on the assumption that monotonicity constraints were externally supplied. They also gave a sufficient condition for termination on infinite

database relations (Logic programs with function symbols can be simulated by Datalog with infinite database relations). They recently proposed a method to infer inequality constraints between two arguments in programs with function symbols [BS91].

Plümer described some extensions for rules that did not satisfy the "uniqueness" property [Plü90a, Plü90b]. His main contribution was that certain constraints involving more than two argument positions could be used. However, he required an "admissibility" property on the rules. Moreover, mutual recursion presented problems for his method. He claimed that every mutual recursion could be eliminated by "unfolding" technique, which does not seem to terminate. His algorithm was exponential-time, since he had to guess one inequality per predicate and the number of possible guesses was exponential in the number of arguments in a predicate[2].

Van Gelder [VG91] investigated the problem of deriving constraints among argument sizes. His method was aimed at finding all constraints in the form of polyhedral convex cones. Thus those constraints can be viewed at least as inequalities among linear combinations of any number of argument positions. His method works bottom-up; starting from a base polyhedral cone corresponding to base cases of recursive rules, "recursive transformations" (similar to the immediate consequence operator T_P) are applied to input polycone until a fixpoint is reached. The recursive transformation generally needs infinitely many iterations for reaching a fixpoint. His method showed the nature of interargument constraints and how they could be derived naturally.

3.3 Argument Size Constraints

We begin by processing each recursive rule in the SCC separately. If the rule has several recursive subgoals, each is processed separately. Say a rule with head p and a recursive subgoal q has been chosen. We obtain linear equations

[2]Plümer assumed there were bounds on clause structures; hence claiming that the complexity is linear.

$$\begin{aligned} x &= a + A\lambda \\ y &= b + B\lambda \qquad x, y, \lambda \geq 0 \\ 0 &= c + C\lambda \end{aligned} \tag{3.1}$$

where x is the vector of *bound* argument sizes in p, y is the vector of *bound* argument sizes in the recursive subgoal q, and λ is the vector of other variables that appear in constraints, including those representing the sizes of logical variables and unbound arguments.

a, A, b and B are derived directly from the rule under consideration. c and C are derived from *interargument constraints* on (possibly, p, q, and) subgoals that precede q in the rule body. The constraints are obtained by the method in Chapter 4 (or supplied by other external means such as [VG91, BS91]).

Example 3.7: Consider the adorned logic program `perm` [3]:

r_1: `perm`^{bf}`([], []).`

r_2: `perm`^{bf}`(P, [X|L]) :-`
`append`^{ffb}`(E, [X|F], P),`
`append`^{bbf}`(E, F, P1),`
`perm`^{bf}`(P1, L).`

The problem is that no order relationships among pairs of arguments show that $P1 < P$. That is, inference systems can derive $P > E$ and $P > F$, but neither E nor F is $\geq P1$. Considering the recursive rule r_2, which is called with the first argument bound and the second free, we show how Eq. 3.1 is set up for this rule.

Let λ be the vector $(P, X, L, E, F, P1)^{\mathrm{T}}$. We have $x_1 = P$, so a and A are:

$$a = \begin{bmatrix} 0 \end{bmatrix}$$
$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

[3] This example was studied by Plümer [Plü90a]. `perm`^{bf} cannot be shown to terminate by any of the previous methods cited. Plümer suggests that unfolding transformations on `append` can lead to a set of rewritten rules that his method can handle; but no algorithm is given.

Similarly, $y_1 = P1$, so

$$b = \begin{bmatrix} 0 \end{bmatrix}$$
$$B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Two `append` subgoals precede the recursive subgoal in the body. We assume that analysis of the `append` procedure has supplied the *imported constraint*:

$$0 = \texttt{append}_1 + \texttt{append}_2 - \texttt{append}_3$$

Applying this to each `append` subgoal, and noting that the termsize of `[X | F]` is $2+X+F$, we get

$$0 = E + (2 + X + F) - P \qquad 0 = E + F - P1$$

Putting this in the array format of Eq. 3.1, gives

$$c = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$
$$C = \begin{bmatrix} -1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 \end{bmatrix}$$

Discussion of this rule continues in Example 3.8. □

3.4 Derivation of Termination Condition

For each predicate p_i in the SCC, we designate a nonnegative vector α_i with arity equal to the number of bound arguments in p_i. If x_i is the vector of these bound arguments, then the inner product, $\alpha_i^{\mathrm{T}} x_i$, is the function that should decrease during recursion. Figure 3.5 illustrates the situation pictorially. Suppose we have two recursive subgoals whose sizes are denoted y' and y''. α is a nonnegative vector. Since $\alpha^{\mathrm{T}} x > \alpha^{\mathrm{T}} y'$ and $\alpha^{\mathrm{T}} x > \alpha^{\mathrm{T}} y''$, nonnegative linear combination of goal size decreases regardless of which recursive subgoal we follow. Finally, the nonnegative linear combination will approach to zero along the vector α over the recursive calls. That is, our goal here is to find a value for the vectors α that guarantee this function decreases.

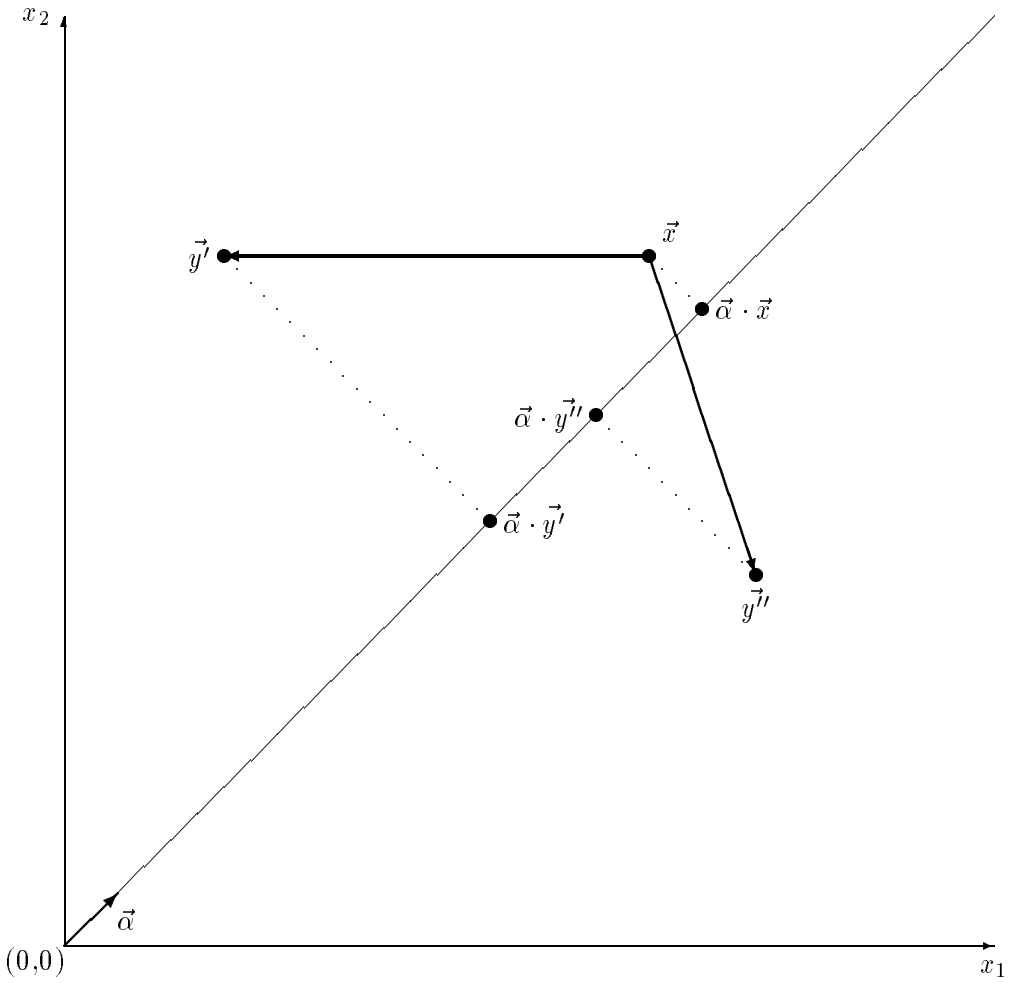


Figure 3.5: Nonnegative linear combination

Again, let p_i be the predicate in the head of the rule and let p_j be the predicate of the chosen recursive subgoal that led to Eq. 3.1. To avoid excessive subscripting we shall refer to x, α, y, and β, rather than to x_i, α_i, x_j, and α_j.

We want to choose α and β to ensure that for all x, y, and λ that satisfy the constraints Eq. 3.1 we have

$$\alpha^{\mathrm{T}} x \geq \beta^{\mathrm{T}} y + \delta_{ij} \tag{3.2}$$

where δ_{ij} is chosen to be 0 or 1 as described in Section 3.6 for mutual recursion, but is

simply 1 if $i = j$.

For the moment, regard α and β as constants. The question then is whether

$$\forall xy\lambda \text{ (Eq. 3.1 implies Eq. 3.2)} \tag{3.3}$$

As is well known, this can be solved as a linear programming problem in which the objective is

$$\textbf{minimize: } \theta = (\alpha^{\mathrm{T}} x - \beta^{\mathrm{T}} y) \tag{3.4}$$

subject to Eq. 3.1. Letting θ^* be the minimum of the objective function, the implication Eq. 3.3 holds if and only if $\theta^* \geq \delta_{ij}$.

Now consider the dual of the above minimization problem [PS82, Sch86]. We use for dual variables u with the same arity as x, v with the same arity as y, and w with the same arity as c. That is, one dual variable is associated with each constraint in the primal; because these constraints are equalities, u, v, and w have unrestricted ranges. The dual is given by

$$\begin{aligned} &\textbf{maximize:} \\ &(u^{\mathrm{T}}, v^{\mathrm{T}}, w^{\mathrm{T}}) \begin{bmatrix} a \\ b \\ c \end{bmatrix} \\ &\textbf{subject to:} \\ &(u^{\mathrm{T}}, v^{\mathrm{T}}, w^{\mathrm{T}}) \begin{bmatrix} I & 0 & -A \\ 0 & I & -B \\ 0 & 0 & -C \end{bmatrix} \leq (\alpha^{\mathrm{T}}, -\beta^{\mathrm{T}}, 0) \end{aligned} \tag{3.5}$$

By duality theory this has the same optimum, θ^*, as the primal, and so Eq. 3.3 is true if and only if the dual constraints of Eq. 3.5 remain satisfiable with the addition of

$$(u^{\mathrm{T}}, v^{\mathrm{T}}, w^{\mathrm{T}}) \begin{bmatrix} a \\ b \\ c \end{bmatrix} \geq \delta_{ij} \tag{3.6}$$

The key observation is that α and β appear linearly in the dual constraints, so we can regard them as variables and remain with a linear system. Lassez made essentially the same observation, but in a more circumlocutory fashion [Las90b]. We incorporate the final constraints,

$$\alpha \geq 0 \qquad \beta \geq 0 \tag{3.7}$$

Combining into one matrix, transposing, and reversing signs as appropriate gives

$$\begin{bmatrix} -I & 0 & 0 & I & 0 \\ 0 & -I & 0 & 0 & -I \\ A^{\mathrm{T}} & B^{\mathrm{T}} & C^{\mathrm{T}} & 0 & 0 \\ a^{\mathrm{T}} & b^{\mathrm{T}} & c^{\mathrm{T}} & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ \alpha \\ \beta \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ \delta_{ij} \\ 0 \\ 0 \end{bmatrix} \tag{3.8}$$

This set of constraints is very amenable to reduction by Fourier-Motzkin elimination [Sch86, LHM89, Las90b]. In this technique a variable is eliminated by "cancelling" all positive occurrences with all negative occurrences, pairwise, creating new rows (with 0 in that variable's column). Then all rows containing a nonzero coefficient for that variable can be eliminated, preserving satisfiability (See Appendix A for details).

In fact, from the origin of a, A, b, and B in Eq. 3.1, we know that all entries of these vectors and matrices are nonnegative. We can use this fact to perform Fourier-Motzkin elimination on u and v in Eq. 3.8, giving

$$\begin{bmatrix} C^{\mathrm{T}} & A^{\mathrm{T}} & -B^{\mathrm{T}} \\ c^{\mathrm{T}} & a^{\mathrm{T}} & -b^{\mathrm{T}} \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} w \\ \alpha \\ \beta \end{bmatrix} \geq \begin{bmatrix} 0 \\ \delta_{ij} \\ 0 \\ 0 \end{bmatrix} \tag{3.9}$$

To claim a theoretical polynomial time bound, we stop with Eq. 3.8 and give the undistinguished variables w unique names so they do not clash with undistinguished variables obtained from other rule-subgoal analyses.

In practice, to conclude the processing of this rule-subgoal combination, we perform Fourier-Motzkin elimination on w as well, leaving constraints that only involve the distinguished variables α_i $(= \alpha)$ and α_j $(= \beta)$. If the subgoal predicate is the same as the head, then $\alpha = \beta$ and constraints are appropriately simplified. Although, δ_{ij} has not been specified yet if $i \neq j$, we still regard it as a constant.

Example 3.8: Continuing with Example 3.7, the constraints corresponding to Eq. 3.9 are

$$\begin{bmatrix} -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \alpha \\ \beta \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \delta_{11} \\ 0 \\ 0 \end{bmatrix}$$

Eliminating w_1 and w_2 reduces them to

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \delta_{11} \\ 0 \\ 0 \end{bmatrix}$$

Finally, requiring $\alpha = \beta$ and setting $\delta_{11} = 1$ gives the single constraint $2\alpha \geq 1$.

Because this is the only rule and only recursive subgoal, termination can be demonstrated using $\alpha = 1/2$. A Prolog session of termination proof for this example can be found in Appendix C.1. □

In general, each combination of rule and recursive subgoal in the SCC is processed to produce constraints on the α_i's. A solution that satisfies all constraints simultaneously (with δ_{ij} chosen appropriately; see Section 3.6), proves top-down termination for the SCC.

Again, to claim the theoretical, but largely imaginary, polynomial time bound, we observe that the size of all the constraints combined is polynomial in the size of the input (where the imported feasibility constraints that led to matrices c and C are counted as part of the input). The final constraints represent a feasibility problem in linear programming.

In practice, Fourier-Motzkin elimination is simple and adequate.

3.5 Multiple Bound Arguments

In recursive procedures with several bound arguments, some previously developed methods require searching through subsets of bound arguments and/or paths in the dependency graph [Nai83, Plü90a]. One of our main motivations in using linear techniques was to short-circuit this complicated and expensive task. Our method seeks a nonnegative linear combination of the bound arguments that suffices for all cases.

Example 3.9: The following procedure merges two input lists [VG91]. We assume the first and second arguments are bound and the third is free.

```
r1:  merge^bbf([], Ys, Ys).
r2:  merge^bbf(Xs, [], Xs).
r3:  merge^bbf([X | Xs], [Y | Ys], [X | Zs]) :-
                X =< Y,
                merge^bbf([Y | Ys], Xs, Zs).
r4:  merge^bbf([X | Xs], [Y | Ys], [Y | Zs]) :-
                Y =< X,
                merge^bbf(Ys, [X | Xs], Zs).
```

Note that there is no explicit relationship between the size of a bound argument in the head and the size of the corresponding bound argument in the recursive subgoal. The first two rules are nonrecursive. Now we process the third rule.

Let λ be the vector $(X, Xs, Y, Ys, Zs)^{\mathrm{T}}$. We derive a, A, b, and B from Eq. 3.1.

$$
\begin{aligned}
a &= \begin{bmatrix} 2 \\ 2 \end{bmatrix} \\
A &= \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \\
b &= \begin{bmatrix} 2 \\ 0 \end{bmatrix} \\
B &= \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}
\end{aligned}
$$

The matrices c and C are empty because the subgoal `X =< Y` does not supply any contribution.

By substituting these into Eq. 3.9, letting $\alpha = (\alpha_1, \alpha_2)$, $\beta = \alpha$, $\delta_{ij} = 1$, and then eliminating redundant rows, we have the following constraints.

$$
\begin{bmatrix} 1 & 0 \\ 1 & -1 \\ -1 & 1 \\ 0 & 2 \\ 0 & 1 \end{bmatrix}
\begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix}
\geq
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}
$$

By symmetry, the constraints for the fourth rule can be obtained by interchanging the first and second columns in the above matrix.

Now combining two sets of constraints and simplifying them reduces them to $\alpha_1 = \alpha_2 \geq 1/2$. The solution implies the sum of two bound arguments always decreases in every recursive call. □

3.6 Extension to Mutual Recursion

One useful feature of our methodology is that rules with mutual recursion and nonlinear recursion do not require much extension to the techniques already illustrated.

3.6.1 Mutual Recursion

Mutual recursion, wherein the SCC contains more than one predicate, arises naturally in user-written rules, and can also be introduced by predicate splitting rule transformations whose purpose is to make every subgoal unifiable with the head of every rule for the same-named predicate.

In the case of mutual recursion the set of δ_{ij} must be chosen so that, regarding them as edge weights, every cycle in the dependency graph has positive weight. Rather than specify this condition as a set of (possibly exponentially many) additional linear constraints, we do the following:

1. For $i \neq j$, set $\delta_{ij} = 0$ when required by existing constraints in the dual; that is, when a dual constraint in Eq. 3.9 has δ_{ij} as the "constant" and has only zeros in c^{T} and a^{T}.
2. Set all other $\delta_{ij} = 1$.
3. Compute the min-plus closure (by Floyd's algorithm) of the resulting edge-weighted graph; check that there are no zero-weight cycles. A zero-weight cycle is strong evidence of nontermination, and the algorithm reports it if found and halts.

Assuming all δ_{ij} have been chosen so that there are no zero-weight cycles, the constraints from all rule-subgoal combinations are combined and tested for feasibility, as illustrated in Example 3.10 below.

Although no natural examples are known, it is possible that no feasible solution exists with nonnegative δ_{ij}, but some feasible solution does exist that includes some negative δ_{ij}, yet has only positive *cycles*. Here we sketch the method of finding feasible solutions to systems of constraints like those in Section 3.6, but without imposing an arbitrary constraint that all $\delta_{ij} \geq 0$. Intuitively, this allows for the possibility that the critical bound subgoals get larger before getting smaller, in such a way that they are smaller by the time a cycle around the dependency graph has been completed. We are aware of no natural examples of such rules.

We need to add constraints that guarantee for each simple cycle in the dependency graph that the sum of the δ's around that cycle is positive. There could be exponentially many

such cycles, in principle. The idea of path constraints was suggested by C. H. Papadimitriou. For each triple of predicates p_i, p_j, and p_k in the same SCC, with $k \neq i$ and $k \neq j$, add the constraint $\pi_{ij} \leq \pi_{ik} + \pi_{jk}$. The new variables π_{ij} represent shortest paths. Their "base cases" are $\pi_{ij} \leq \delta_{ij}$. Positive cycles are enforced by $\pi_{ii} \geq 1$. Again, the polynomial time bound may be claimed by reference to linear programming theory; in practice, our program quietly runs Fourier-Motzkin elimination on the π_{ij} reducing the path inequalities to involve only δ_{ij} and constants. These are added to other constraints as discussed in Section 3.4.

3.6.2 Nonlinear Recursion

Nonlinear recursion occurs frequently in divide-and-conquer algorithms, and many other applications. The main point is that, when the second and subsequent recursive subgoals are being analyzed for constraints on α_i and α_j, the earlier recursive subgoals must contribute their interargument feasibility constraints to the effort. Therefore, interargument feasibility constraints must be produced for the entire SCC before its termination analysis begins.

Example 3.10: Issues of both mutual and nonlinear recursion are illustrated with the following rules, which specify an arithmetic expression parser. The first argument is the list to be parsed, and is assumed bound. The second argument is the unparsed suffix (or continuation), and is free.

```
e(L, T) :- t(L,  ['+' | C] ), e(C, T).
e(L, T) :- t(L, T).

t(L, T) :- n(L,  ['*' | C] ), t(C, T).
t(L, T) :- n(L, T).

n(['(' | A] , T) :- e(A,  [')' | T]).
n([L | T], T) :- z(L).
```

The only nonrecursive rule is the last; `z` is in a lower SCC. We suppose no knowledge about `z`.

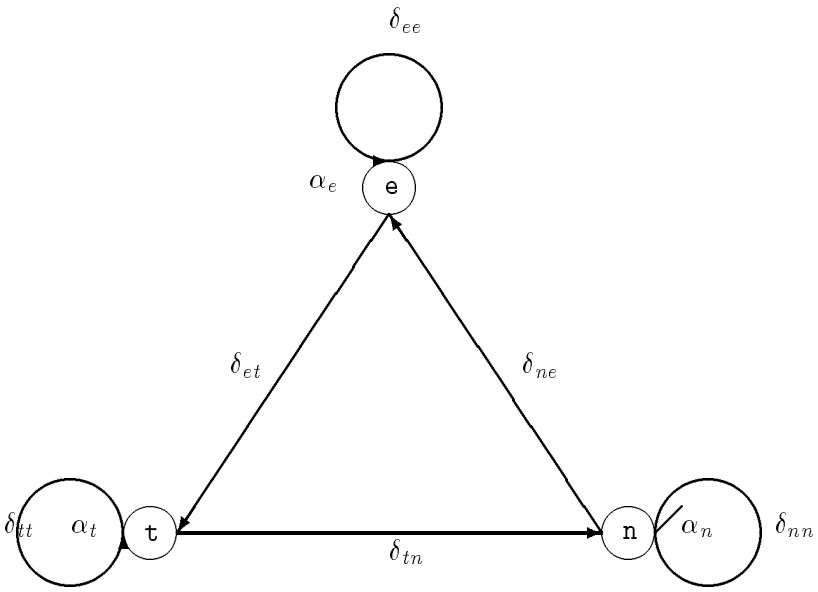


Figure 3.6: Extension to mutual recursion.

This example was studied by Plümer [Plü90a], who eliminated mutual recursion by pushing e, n, and t into the third argument of a new predicate `parse(L, T, P)`. Intuitively, such a syntactic change should not make a substantial difference. He had to make further *ad hoc* assumptions to handle `parse`.

The predicate dependency graph is drawn in Figure 3.6. To apply our method to mutual recursion, we have to make sure every simple cycle has a positive weight. That is,

$$\begin{aligned}
\delta_{ee} &> 0 \\
\delta_{tt} &> 0 \\
\delta_{nn} &> 0 \\
\delta_{et} + \delta_{tn} + \delta_{ne} &> 0
\end{aligned}$$

We shall only discuss in detail the derivation of constraints for the first rule and its e subgoal. Recall that the form is

$$x = a + A\lambda \qquad y = b + B\lambda \qquad 0 = c + C\lambda \qquad x, y, \lambda \geq 0$$

The first point is that the `t` subgoal precedes the `e` subgoal, so should supply feasibility constraints (values for c and C in Eq. 3.1), *even though it is in the same SCC*. It is not immediately obvious what these constraints are, but they can be found by the Van Gelder's methods [VG91], and are:

$$0 = 2 - \mathtt{t}_1 + \mathtt{t}_2 + \mu$$

that is, $\mathtt{t}_1 \geq 2 + \mathtt{t}_2$ (μ is a "slack variable").

Let λ be the vector $(L, T, C, \mu)^{\mathrm{T}}$. The above equation generates

$$\begin{aligned} c &= \begin{bmatrix} 4 \end{bmatrix} \\ C &= \begin{bmatrix} -1 & 0 & 1 & 1 \end{bmatrix} \end{aligned}$$

We also have

$$\begin{aligned} a &= \begin{bmatrix} 0 \end{bmatrix} \\ A &= \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \\ b &= \begin{bmatrix} 0 \end{bmatrix} \\ B &= \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \end{aligned}$$

Let us identify e with α, t with β, and n with γ. Then the constraints corresponding to Eq. 3.9 are (note that $(A^{\mathrm{T}} - B^{\mathrm{T}})$ and $(a^{\mathrm{T}} - b^{\mathrm{T}})$ form the second column):

$$\begin{bmatrix} -1 & 1 \\ 0 & 0 \\ 1 & -1 \\ 1 & 0 \\ 4 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ \alpha \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \delta_{ee} \\ 0 \end{bmatrix}$$

Elimination of the nondistinguished w_1 boils this down to

$$4\alpha \geq \delta_{ee}$$

The development for the third rule with `t` as the recursive subgoal is similar, yielding

$$4\beta \geq \delta_{tt}$$

The first rule with `t` as the recursive subgoal and the third rule with `n` as the recursive subgoal produce the same constraints as the second and fourth rules, which are discussed next.

The second and fourth rules are straightforward (yielding $\alpha \geq \beta \geq \gamma$), but the important point is that their constraints force δ_{et} and δ_{tn} to be 0.

Finally, the fifth rule gives

$$\begin{bmatrix} -1 & 1 \\ 0 & 0 \\ 0 & 2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \gamma \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ \delta_{ne} \\ 0 \\ 0 \end{bmatrix}$$

which does *not* force δ_{ne} to be 0 (because $a = [2]$). These constraints reduce to: $\gamma \geq \alpha$ and $2\gamma \geq \delta_{ne}$.

Besides self-loops, the dependency graph has edges (e, t), (t, n), and (n, e). As mentioned, δ_{et} and δ_{tn} were required to be 0. However, assigning the "weight" $\delta_{ne} = 1$ results in no zero-weight cycles, so the termination test is able to continue. Of course, δ_{ee} and δ_{tt} are 1. Now straightforward computation leads to

$$\alpha = \beta = \gamma \geq 1/2$$

so termination is proved. A Prolog session of termination proof for this example can be found in Appendix C.2. □

3.7 Limitations

A set of rules may terminate in top-down execution due to the inability of certain terms to unify, rather than because of term size reduction. Sometimes the syntactic transformations described later clarify such situations before termination detection begins. The argument mapping method [BS89b, BS91] pays closer attention to unification issues than we do, and has the remarkable property of detecting certain cases where unification fails due to the "occurs check".

r_1: `q(X,Y) :- e(X,Y).`

r_2: `q(X,f(f(X))) :- p(X,f(f(X))), q(X,f(X)).`

r_3: `q(X,f(f(Y))) :- p(X,f(Y)).`

r_4: `p(X,Y) :- e(X,Y).`

r_5: `p(X,f(Y)) :- r(X,f(Y)), p(X,Y).`

r_6: `r(X,Y) :- e(X,Y).`

r_7: `r(X,f(Y)) :- q(X,Y), r(X,Y).`

r_8: `r(f(X),f(X)) :- t(f(X),f(X)).`

r_9: `t(X,Y) :- e(X,Y).`

r_{10}: `t(f(X),f(Y)) :- q(f(X),f(Y)), t(X,Y).`

There are several cycles in predicate dependency graphs. Consider the cycle $q \rightarrow p \rightarrow r \rightarrow t \rightarrow q$ produced by r_2, r_5, r_8, r_{10}. No argument sizes diminish through the cycle so our method fails to detect the termination of the cycle. Suppose initial call to q unifies with `q(X, f(f(X)))`. Following the cycle, two new goals `p(X, f(f(X)))` by r_2 and `r(X, f(f(X)))` by r_5 are generated. The goal `r(X, f(f(X)))` fails to unify with the head `r(f(X), f(X))` of r_8 with occur-check. Sagiv's method can detect termination of such situation [Sag91][4].

[4]This example was given by Sagiv in personal communication

Usually, the problem regarding occur-check at one step of resolution can be resolved by predicate splitting. For example, consider a recursive rule:

```
p(X, X) :- p(X, f(X)).
```

The subgoal does not unify with the head so we assign different predicate name, say,

```
p0(X, X) :- p1(X, f(X)).
```

However, predicate splitting is not powerful enough to capture the occur check problem due to several steps of resolution as in Sagiv's example.

Also, it is possible that bound arguments really do shrink during recursion, but the interargument constraints to prove that fact are not derivable by known methods. For instance, the feasible region may not be accurately characterized by a finite set of linear constraints. Consider a `mergesort` example.

m_1: `mergesort([],[]).`

m_2: `mergesort([X],[X]).`

m_3:
```
mergesort([X,Y|Rs],Zs) :-
            split([X,Y|Rs],Us,Vs),
            mergesort(Us,Usort),
            mergesort(Vs,Vsort),
            merge(Usort,Vsort,Zs).
```

s_1: `split([],[],[]).`

s_2: `split([X|Xs],[X|Ys],Zs) :- split(Xs,Zs,Ys).`

`merge` is a usual procedure to merge two sorted lists. Our inference algorithm for interargument constraints supplies a linear constraint:

$$\texttt{split}_1 = \texttt{split}_2 + \texttt{split}_3.$$

Applying this constraint to the first subgoal in m_3, we have a linear constraint among variables:

$$4 + X + Y + Rs = Us + Vs.$$

This does not imply the head argument size ($4 + X + Y + Rs$) is greater than the recursive subgoal argument size (Us or Vs), hence failure to prove termination. There are two possible remedies for this problem.

One approach is to get stronger interargument constraints. Since `split` divides the input list (first argument) into two lists (second and third arguments) evenly, if the first argument has more than two elements ($\texttt{split}_1 > 4$) then the first argument size is greater than the other two ($\texttt{split}_1 > \texttt{split}_2$ and $\texttt{split}_1 > \texttt{split}_3$). The `split` subgoal in m_3 satisfies the premise. So we can conclude that termsize(`[X,Y|Rs]`) > termsize(`Us`) and termsize(`[X,Y|Rs]`) > termsize(`Vs`). This proves termination of `mergesort`^{bf}. However, it is noticed many published methods to infer disjunctive constraints are practically unusable owing to their higher complexity.

The other approach, introduced by [UVG88], is to apply predicate splitting so that termination behavior is better exposed. It is simple to apply the technique and adequate to resolve the problem. The first subgoal `split` in m_3 does not unify with s_1. The relation `split` can be split into three relations, `split`, `split1` and `split2`. The heads of rules with which `split([X,Y|Rs],Us,Vs)` does not unify are renamed to `split1`; those with which `split([X,Y|Rs],Us,Vs)` does unify are renamed to `split2`. Rules for `split` are added: `split(X,Y,Z) :- split1(X,Y,Z)` and `split(X,Y,Z) :- split2(X,Y,Z)`. Finally, `split` subgoals in the program are specialized to `split1` or `split2` if possible. The above example transforms into:

```
m1: mergesort([],[]).
m2: mergesort([X],[X]).
m3: mergesort([X,Y|Rs],Zs) :-
              split2([X,Y|Rs],Us,Vs),
              mergesort(Us,Usort),
              mergesort(Vs,Vsort),
              merge(Usort,Vsort,Zs).
```

s_1: `split1([],[],[]).`

s_2: `split2([X|Xs],[X|Ys],Zs) :- split(Xs,Zs,Ys).`

s_3: `split(Xs,Ys,Zs) :- split1(Xs,Ys,Zs).`

s_4: `split(Xs,Ys,Zs) :- split2(Xs,Ys,Zs).`

Repeated application of predicate splitting terminates, essentially because rules are simply partitioned, and no substitutions are done.

Unfolding is just an application of resolution. If predicate p has k rules $p(\vec{X}_i) \leftarrow B_i(\vec{X}_i)$, then one rule in which p is a subgoal, say $q(\vec{Y}) \leftarrow \ldots, p(\vec{Z}), \ldots$, can be replaced by k rules,

$$q(\vec{Y}\theta_i) \leftarrow \ldots, B_i(\vec{X}\theta_i), \ldots$$

where the θ_i are the respective most general unifiers of $\vec{X}_i$ and $\vec{Z}$.

Safe unfolding is a special case that applies when no rule for p has p as a subgoal. In this case all p subgoals are replaced by unfolding, and p is thereby removed from that SCC of the dependency graph. Continuing the above example, safe unfolding applies to `split`, then to `split1`, and gives:

m_1: `mergesort([],[]).`

m_2: `mergesort([X],[X]).`

m_3:
```
mergesort([X,Y|Rs],Zs) :-
            split2([X,Y|Rs],Us,Vs),
            mergesort(Us,Usort),
            mergesort(Vs,Vsort),
            merge(Usort,Vsort,Zs).
```

s_1: `split2([X],[X],[]).`

s_2: `split2([X|Xs],[X|Ys],Zs) :- split2(Xs,Zs,Ys).`

s_3: `split(Xs,Ys,Zs) :- split2(Xs,Ys,Zs).`

s_4: `split([],[],[]).`

If `split` is not referenced elsewhere, their rules may be discarded. Repeated application of safe unfolding must terminate because SCCs shrink upon each application; hence the term *safe*. So many practical cases of mutual recursion can be eliminated by this transformation that there is sometimes the mistaken perception that all can be.

In the above program, we still notice the first subgoal `split2` in m_3 does not unify with s_1. Applying one more predicate splitting and unfolding gives:

```
m1: mergesort([],[]).

m2: mergesort([X],[X]).

m3: mergesort([X,Y|Rs],Zs) :-
              split([X,Y|Rs],Us,Vs),
              mergesort(Us,Usort),
              mergesort(Vs,Vsort),
              merge(Usort,Vsort,Zs).


s1: split([X1,X2],[X1],[X2]).

s2: split([X|Xs],[X|Ys],Zs) :- split(Xs,Zs,Ys).
```

Inferring interargument constraints for the `split` predicate in the above program:

$$\begin{aligned} \texttt{split}_1 &= 4 + \lambda_1 + \lambda_2 \\ \texttt{split}_2 &= 2 + \lambda_1 \\ \texttt{split}_3 &= 2 + \lambda_2 \end{aligned}$$

These equations imply that termsize(`[X,Y|Rs]`) $\geq 2 +$ termsize(`Us`) and termsize(`[X,Y|Rs]`) $\geq 2 +$ termsize(`Vs`). That shows termination of $\texttt{mergesort}^{bf}$.

3.8 Extension to CLP(R)

In this section, we briefly discuss the extension of our method to CLP(R) programs. Further research on this area will be investigated later.

We cannot apply our termination detection method directly to CLP(R) programs The reason is variables in CLP(R) are real variables so they are not constrained to nonnegativity.

Therefore, the assumption that nonnegative combination of bound arguments (αx in Eq. 3.2) are positive does not hold.

We need to know the domain where recursive calls are satisfied. Consider the `while` example in CLP(R):

```
while(X,Y) :-
      X =< Y,
      while(X+1, Y+2).
while(X,Y) :- X > Y.
```

Note `X` and `Y` are real variables. When both arguments of a goal are bound, the goal terminates, because the second argument grows faster than the first, finally violating the condition `X =< Y`.

Let x_1 and x_2 be the first argument and the second of `while` predicate, respectively. In order for a goal to unify with `while` rule recursively, the goal must satisfy the constraint $x_1 \leq x_2$, because of the subgoal `X =< Y` in the recursive rule. In addition to Eq. 3.3 as termination condition, we have to ensure that $x_1 \leq x_2$ implies $\alpha x \geq 0$.

Let $R(x)$ be the constraint set associated with a predicate so that a goal unifies recursively. The following constitutes termination condition for one rule.

$$\forall x y \lambda \ (R(x) \text{ implies } \alpha x \geq 0 \ \wedge \ \text{Eq. 3.1 implies Eq. 3.2}) \tag{3.10}$$

In case of the above `while` example, the constraint that α must be (1,-1) is derived. How to derive $R(x)$ in general will be a further research topic.

3.9 Discussion

We have presented a methodology for termination detection that uses duality theory of linear programming, and is quite general and straightforward.

This method can be fully automated and is easy to implement. We implemented the method in Prolog. The complete system consists of 2770 lines of Prolog code. Some performance measures are listed in Table 3.1, and the Prolog sessions for two test inputs,

input program	`perm`	`quicksort`	`mergesort`	`parser`
cputime in milliseconds	310	400	549	1319
ratio	3.44	2.22	2.20	9.42

Table 3.1: Performance measures of termination detection

`perm` and `parser` are shown in Appendix C. The programs were tested on SUN4 sparc station using Sicstus Prolog 2.1. The second row of Table 3.1 indicates the analysis time and the third row indicates the ratio of analysis time to compilation time:

$$\text{ratio} = (\text{analysis time}) \ / \ (\text{compilation time}).$$

The results show the analysis is quite slow compared to the compilation of the programs. Our broad conclusion is the termination detection together with the inference of inter-argument constraints (See Section 4.6) can be available as an user option of Prolog compilers. The program `perm` and `parser` are typical linear and mutually programs respectively, with which the power of our method is shown. We have not tested our implementation with substantial size of programs, because our program take only pure logic program (without negation, cut, or builtin predicates) as an input. However, we conjecture that the running time for large input is only linear of running times for our test inputs, since a practical logic program is divided into small strongly connected components in its predicate dependency graph.

4. Inference of Interargument Constraints

4.1 Introduction

Our termination detection method described in the previous chapter was built on the basis of interargument constraints for predicates imported from lower SCCs. In this chapter, we will describe how to infer such constraints.

Local variables are those appearing only in the body of a rule.

In many programs (see Example 3.7), it is often the case that there are no direct relationships among the argument sizes in a head and those in a recursive subgoal due to existence of local variables. So termination proof usually requires the relationship among argument sizes of subgoals in order to relate the argument sizes in the head and the recursive subgoals.

Example 4.1: Consider a recursive rule:

```
p([a|X]) :- p(X).
```

The term size of the head argument is bigger than that of the subgoal argument size, because there are no local variables. In this case, termination proof is direct. Consider another recursive rule:

```
p(X) :- a(X, Y), p(Y).
```

Let x, y denote the size of `X` and the size of `Y`, respectively. and a_i the i-th argument size of `a`. `X` and `Y` are not related to each other in this form, since `Y` is a local variable. By the subgoal `a`, `X` is processed into Y somehow. If it is true that $a_1 > a_2$, then we can infer $x > y$. $a_1 > a_2$ is called an interargument constraint for `a`. It can be inferred by taking the semantics of the definition for `a` into account. □

We also believe the relationships among argument sizes are useful for showing safety of queries in deductive databases, for code improvement using better memory allocation strategies, and for deciding granularity of tasks in the parallel execution of logic programs as well as for automatic termination proofs.

Interargument constraint (for short, IC) of a predicate is a set of constraints which every derivable fact with respect to the predicate satisfies. Methods to derive ICs have been studied recently in terms of Datalog [BS89a, BS91] or logical rules with function symbols [VG91]. In their methods IC is formalized by the least fixpoint of bottom-up inference operator similar to "immediate consequence operator". In [BS91] ICs are captured by a disjunctive union of inequalities between two argument sizes. They show undecidability results on interesting questions such as whether a specific procedure for computing ICs computes a finite set of constraints. Van Gelder studied a method to derive a single conjunctive set of constraints by taking the convex union of disjunctive sets of constraints as an input for bottom-up inference [VG91]. It is often the case that both methods fail to finitely converge.

In this chapter, IC with respect to n-ary predicate is captured by a single polyhedral convex set in the nonnegative orthant of R^n, called a *polycone*. We formalize the derivation of ICs in a way similar to [VG91]. This method is an instance of bottom-up abstract interpretation based on fixpoint semantics, so correctness is guaranteed. We introduce two new techniques to capture ICs in finite time, using *affine widening* and *translativeness* property.

Widening polycones to affine hulls accelerates the convergence of the transformation associated with the inference operator up to finitely many iterations, yet supplying useful ICs.

We characterize a class of linear recursive logic procedures satisfying translativeness property, for which precise ICs (corresponding to lfp of the transformation) are automatically given. Whether a procedure satisfies translativeness property can be tested by analyzing the relationship between argument sizes of the head and those of the recursive subgoal. Many practical programs satisfy the translativeness property, and ICs which cannot be derived by other methods can be captured.

We also investigate an efficient method to find extreme points in an affine image of a polycone, which is essential to verifying a fixpoint of a transformation. We believe this

method is also useful in the implementation of CLP(R).

Overall, the main contribution of this chapter is to provide an efficient method to derive precise ICs for practical logic programs.

Example 4.2: With a simple `append` procedure, we compare our results with others. Sagiv and Brodsky's method [BS91] cannot capture the IC we derive here since their IC is in the form of inequalities between two argument positions. Van Gelder's method [VG91] does not converge with this example; his heuristic that "sometimes" works gives the same IC. However, we provide an affine widening that always works. Let us consider a version of the usual `append` procedure abstracted by listsize (See Example 4.3 for rules).

$append(0, t, t) \leftarrow t \geq 0.$

$append(1 + u, v, 1 + w) \leftarrow (u, v, w) \geq \vec{0}, append(u, v, w).$

The relationship among argument sizes with respect to `append` is essentially an infinite set:

$$\{append(0, x, x),\ append(1, x, 1 + x),\ append(2, x, 2 + x), \ldots\}.$$

We approximate this set to the convex union of the set elements, which is $append(y, x, y+x)$. That implies that the size of the third argument is the sum of the size of the first and the size of the second. It is indeed a fixpoint of our transformation supplied with affine widening, which is derived in three steps of transformations.

The relationship can also be captured via the analysis of the structure of a transformation since `append` procedure satisfies translativeness property. Recursive procedures in logic programs usually handle structures as a backbone of recursion called *recursion on structures*; that is, some elements of a structure are processed in a recursive call and the rest is passed for the subsequent calls. Linear recursive procedures relying on "recursion on structures" technique usually are translative. □

Section 4.2 introduces basic concepts on term size abstraction and linear constraints.

In Section 4.3 we formalize transformations to derive ICs, and introduce affine widening to approximate the least fixpoint.

Section 4.4 introduces the *translativeness* property of a class of linear recursive logic programs. For such a class of programs, ICs are obtained with no iterations. Linear recursive programs relying on "recursion on structures" usually satisfy this property.

Section 4.5 concerns a method finding all extreme points of a polycone in parametric representation. We remove nonextreme points from affine images of extreme points of a polycone using the adjacency graph on extreme points.

Section 4.6 summaries the chapter.

4.2 Basic Concepts

Logic programs can be abstracted by the size of terms. Our method does not depend on any specific size definition. In examples we shall use *listsize* [UVG88] and *structural term size* [VG91, SVG91]. List size and structural term size are defined informally to be the number of edges in the rightmost path and the number of edges, resp., in the tree that represents the ground term. For terms containing logical variables, a real variable x constrained to nonnegativity is associated with each logical variable `X`. For instance, the listsize and structural term size of `f(a,g(X),X)` are $1 + x$ and $4 + 2x$, resp. A rule is abstracted by replacing terms by their sizes. We shall call the resulting rules and programs *abstract rules* and *abstract programs*, resp.

Example 4.3: Consider usual `append` procedure.

```
a([], T, T).
a([X|U], V, [X|W]) :- a(U,V,W).
```

Replacing terms by their structural term sizes reduces to the following CLP(R)-like procedure.

$a(0, t, t) \leftarrow t \geq 0.$

$a(2 + x + u, v, 2 + x + w) \leftarrow (x, u, v, w) \geq \vec{0}, a(u, v, w).$

□

The i-th argument size of predicate p is denoted by p_i; for example, $a_1 = 0, a_2 = v, a_3 = v$ or $\vec{a} = (0, v, v)$ in the base rule of Example 4.3. $\vec{a}$ denotes a vector of a_i's.

A *polyhedral convex set* in R^n is the intersection of finitely many closed half-spaces. A *polycone* is a polyhedral convex set in the nonnegative orthant of R^n. IC with respect to an n-ary predicate p is captured in the form of a polycone in R^n. A polycone in R^n is usually represented by an affine image of another polycone in R^m; that is, represented in the form of:

$$\begin{gathered} \vec{p} = \vec{a} + A\vec{x} \\ C(\vec{x}) \end{gathered} \tag{4.1}$$

where $\vec{p}$ is the vector of variables representing argument sizes, $\vec{x}$ is the vector of *parameters*, $\vec{a}$ is a vector constant, A is a matrix, and $C(\vec{x})$ is a set of constraints in parameter space, containing nonnegativity constraints for parameters. Eq. 4.1 is called *parametric representation* of a polycone. We sometimes use the same symbol, for example Δ, to denote a polycone or its parametric representation; however, its meaning should be clear by the context. An empty polycone is denoted by $\emptyset$, whose parametric representation is inconsistent.

The parametric representation of a polycone can be generated by a tuple of a set of points and a set of rays called a *generator* The polycone is a vector sum of a convex hull of points $\vec{x_1}, \ldots, \vec{x_n}$ and a cone of rays $\vec{y_1}, \ldots, \vec{y_m}$; that is,

$$\begin{gathered} \vec{p} = \sum \lambda_i \vec{x_i} + \sum \mu_j \vec{y_j} \\ \sum \lambda_i = 1 \\ \lambda_i \geq 0, i = 1, \ldots, n \\ \mu_j \geq 0, j = 1, \ldots, m \end{gathered} \tag{4.2}$$

We shall say that a polycone in Eq. 4.2 is *generated* by points $\vec{x_1}, \ldots, \vec{x_n}$ and rays $\vec{y_1}, \ldots, \vec{y_m}$. A unique minimal generator called a *frame* is obtained by eliminating nonextreme points and rays from the generator. Equivalence of two polycones can be tested by comparing their frames. A generator and a frame of a polycone Δ are denoted by gen(Δ) and frame(Δ), resp. For example, let Δ be a polycone corresponding to Eq. 4.2, gen(Δ) is ($\{\vec{x_1}, \ldots, \vec{x_n}\}$, $\{\vec{y_1}, \ldots, \vec{y_m}\}$).

The *convex union* $\Delta_1 \sqcup \Delta_2$ of two polycones Δ_1 and Δ_2 is defined as the closure of the set of points that are convex combinations of any two points of union of two polycones. $\sqcup$ is commutative and associative. A convex union of n polycones $\Delta_1, \ldots, \Delta_n$ is denoted by $\sqcup\{\Delta_1, \ldots, \Delta_n\}$. The generator of $\sqcup\{\Delta_1 \ldots \Delta_n\}$ is a tuple of the union of sets of points and the union of sets of rays in the generators of Δ_i's, so its parametric representation can be generated by points and rays in $\text{gen}(\Delta_1), \ldots, \text{gen}(\Delta_n)$.

The *affine hull* $\text{aff}(\Delta)$ of a polycone Δ is the smallest subspace containing Δ. If $\text{gen}(\Delta) = (\{\vec{x_1}, \ldots, \vec{x_n}\}, \{\vec{y_1}, \ldots, \vec{y_m}\})$, then

$$\text{aff}(\Delta) \equiv \begin{cases} \vec{p} = \sum \lambda_i \vec{x_i} + \sum \mu_j \vec{y_j} \\ \sum \lambda_i = 1 \end{cases} \tag{4.3}$$

See [Sch86, Roc70] for details on polyhedral convex sets.

Substitution $[x_1/e_1, \ldots, x_n/e_n]$ is defined in an usual way; that is, a variable x_i is replaced by a linear arithmetic term e_i. Using vector notation, it is denoted by $[\vec{x}/\vec{e}]$.

A predicate dependency graph is a digraph with nodes of predicates and arcs from p to q where p is a head predicate of a certain rule and q is its subgoal predicate. Intuitively, q supports the derivation, or solution, of p. We identify strongly connected components (SCC) of this digraph, and a directed acyclic graph (DAG) whose nodes are SCCs. In the actual derivation of ICs, we analyze each SCC at a time starting from the leaves in the DAG. The analysis of an SCC is supported by ICs from the lower SCCs.

4.3 Transformations Corresponding to Logic Programs

Suppose there are k predicates $p^1, \ldots, p^k$ in a program P, and let $a(p^i)$ denote the arity of p^i. Let $\Delta_{p^1}, \ldots, \Delta_{p^k}$ be parametric representations of polycones in $R^{a(p^1)}, \ldots, R^{a(p^k)}$ respectively, and $\Delta = (\Delta_{p^1}, \ldots, \Delta_{p^k})$. Let $\mathcal{U}_{p^i}$ be the set of all convex sets in the positive orthant of $R^{a(p^i)}$ and $\mathcal{U} = \mathcal{U}_{p^1} \times \cdots \times \mathcal{U}_{p^k}$. $\mathcal{U}$ forms a complete lattice equipped with componentwise subset ordering $\sqsubseteq$. We now introduce a transformation corresponding to one logical rule.

Definition 4.1: (natural transformation) Suppose the i-th non-base abstract rule whose head predicate is p is in the form of:

$$p(\vec{e}) \leftarrow \vec{\lambda} \geq \vec{0}, q(\vec{f}), \ldots r(\vec{g}). \tag{4.4}$$

where $\vec{\lambda}$ is a vector of variables appearing in the rule. $\vec{e} = (e_1, \ldots, e_{a(p)})$, $\vec{f} = (f_1, \ldots, f_{a(q)})$, $\ldots$ $\vec{g} = (g_1, \ldots, g_{a(r)})$ are vectors of linear arithmetic terms. Assume that every polycone is in parametric representation. The *natural transformation* $\Psi_{<p,i>} : \mathcal{U} \rightarrow \mathcal{U}_p$ corresponding to the i-th rule of predicate p is defined by:

$$\Psi_{<p,i>}(\Delta) \equiv \begin{cases} \vec{p} = \vec{e} \\ \Delta_q[\vec{q}/\vec{f}] \\ \vdots \\ \Delta_r[\vec{r}/\vec{g}] \\ \vec{\lambda} \geq \vec{0} \end{cases} \tag{4.5}$$

For i-th base abstract rule in the form of:

$$p(\vec{e}) \leftarrow \vec{\lambda} \geq \vec{0}. \tag{4.6}$$

we have the following *base* polycone:

$$B_{<p,i>} \equiv \begin{cases} \vec{p} = \vec{e} \\ \vec{\lambda} \geq \vec{0} \end{cases} \tag{4.7}$$

□

Example 4.4: Continuing with Example 4.3, suppose we have a parametric representation of a polycone: $\Delta = (\{\vec{a} = (0, t, t), t \geq 0\})$. Then $\Psi_{<a,1>}(\Delta) = \{\vec{a} = (2 + x + u, v, 2 + x + w)$, $u = 0$, $v = t$, $w = t$ $x \geq 0$, $u \geq 0$, $v \geq 0$, $w \geq 0$, $t \geq 0\}$. For the base rule, we have a polycone: $B_{<a,1>} = (\{\vec{a} = (0, t, t), t \geq 0\})$. □

We extend this formalism to the whole program P in a natural way.

Definition 4.2: (recursive transformation) A *recursive transformation* $T_P : \mathcal{U} \rightarrow \mathcal{U}$ associated with a program P is a direct product of $T_{p^1}, \ldots, T_{p^k}$. T_{p^i} is defined by a convex union

of l base polycones and m natural transformations associated with the rules whose head predicate is p^i.

$$\begin{aligned} T_P(\Delta) &= (T_{p^1}(\Delta), \ldots, T_{p^k}(\Delta)) \\ T_{p^i}(\Delta) &= \sqcup\{B_{<p^i,1>}, \ldots, B_{<p^i,l>}, \Psi_{<p^i,1>}(\Delta), \ldots, \Psi_{<p^i,m>}(\Delta)\} \end{aligned} \tag{4.8}$$

□

Natural transformation and recursive transformation are similar to those in [VG91], but more general.

Theorem 4.1: A recursive transformation T_P is monotone.

Proof: The operations involved in T_P are essentially intersection, projection, and convex union of polycones. These operations preserve monotonicity. □

Theorem 4.2: There exists the least fixpoint associated with T_P.

Proof: It follows from the fact that T_P is monotone and $\mathcal{U}$ forms a complete lattice. □

Our formalism is an instance of abstract interpretation, hence correctness is guaranteed. A fixpoint can be verified by comparing the frames of polycones in Δ and $T(\Delta)$ componentwise.

Example 4.5: Continuing with Example 4.3, we now compute the least fixpoint of recursive transformation. Since there is only one predicate `a` in a program, Let B, Ψ, T denote $B_{<a,1>}, \Psi_{<a,1>}, T_P = T_a$, resp. Note that $\emptyset$ denotes an empty polycone. In parametric representation we omit nonnegativity constraints of variables. How to find extreme points and rays will be covered in Section 4.5.

1. (base polycone) $B = \{\vec{a} = (0, t, t)\}$, frame$(B) = (\{(0,0,0)\}, \{(0,1,1)\})$
2. (natural transformation) $\Psi(\emptyset) = \emptyset$
3. (recursive transformation; convex union of B and $\emptyset$) frame$(T(\emptyset)) = (\{(0,0,0)\}, \{(0,1,1)\})$
4. (verify a fixpoint) frame$(\emptyset) \neq$ frame$(T(\emptyset))$,
   so generate the parametric representation of $T(\emptyset)$
5. (natural transformation) $\Psi(T(\emptyset)) = \{\vec{a} = (2+x+u, v, 2+x+w), u = 0, v = t, w = t\}$,
   frame$(\Psi(T(\emptyset)) = (\{(0,0,0)\}, \{(0,1,1), (1,0,1)\})$

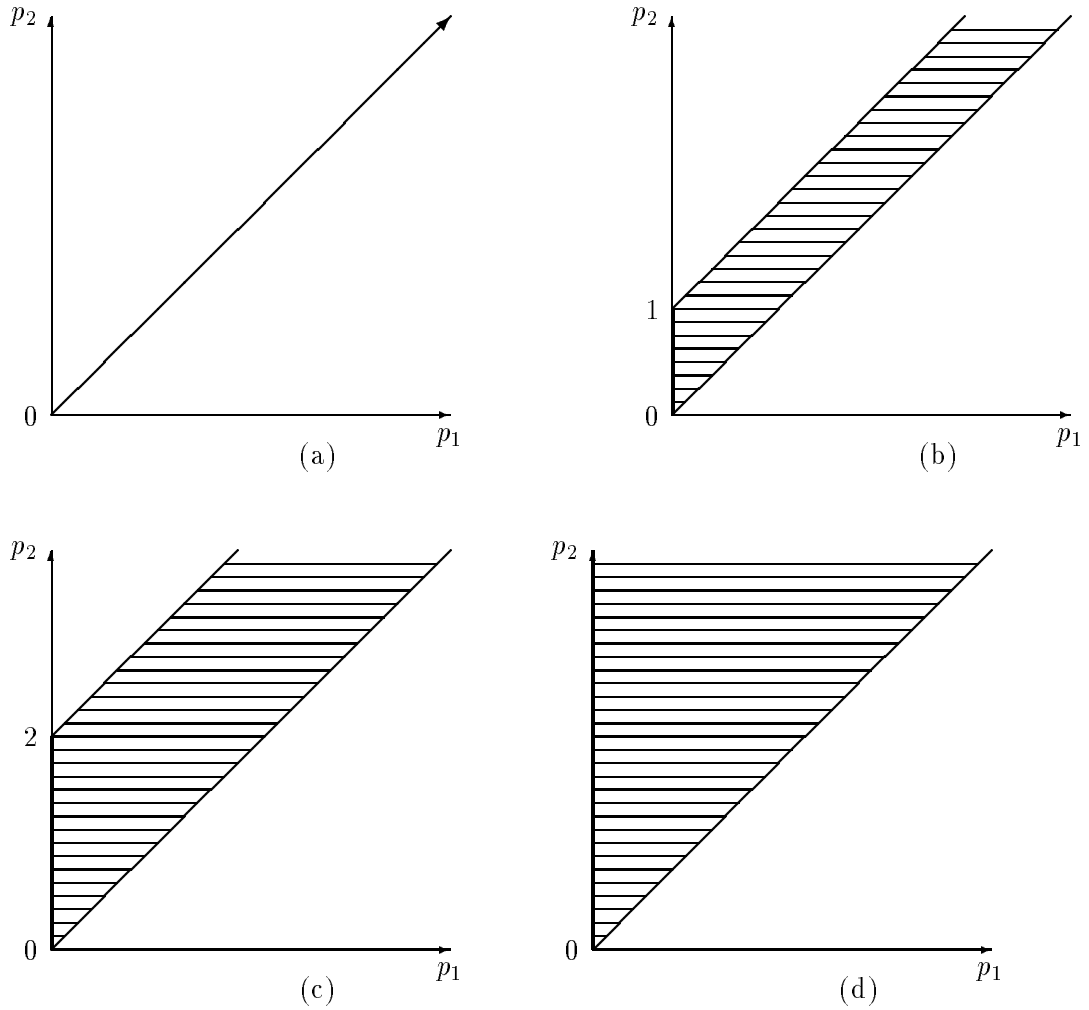


Figure 4.1: Polycones after recursive transformations

6. (recursive transformation; convex union of B and $\Psi(T(\emptyset))$)
   frame$(T^2(\emptyset)) = (\{(0,0,0)\}, \{(0,1,1),(1,0,1)\})$
7. (verify a fixpoint) frame$(T^1(\emptyset)) \neq$ frame$(T^2(\emptyset))$,
   so generate the parametric representation of $T^2(\emptyset)$

With one more transformation, we reach the least fixpoint. IC generated by the frame of lfp is $\{\vec{a} = (u, v, u+v)\}$. In `append` procedure, its third argument size is equal to the sum of its first and second. □

Unfortunately, recursive transformations associated with many practical programs often fail to converge finitely, in particular if we use listsize abstraction.

Example 4.6: Consider an abstract procedure below:

$p(t,t) \leftarrow t \geq 0.$

$p(u,1+v) \leftarrow (u,v) \geq \vec{0}, p(u,v).$

In Figure 4.1, (a) shows the polycone from the base case rule. (b) and (c) shows the polycones after one and two recursive transformations, respectively. (d) shows the polycone after infinite transformations, which is the fixpoint of the transformation. □

Whether a recursive transformation converges finitely is believed to be undecidable[1]. In logic programs data structures in input arguments are usually transferred to output arguments even though they may be broken into smaller ones, or combined into larger ones, or the values of elements are processed. In terms of argument sizes, the relationship among them may be approximated by a set of equalities, that represents an affine subspace. Requiring the least fixpoint is too much in general; however, postfixpoints [2] are also a correct upper approximation of least fixpoint by Tarski's fixpoint theorem: $T(x) \leq x$ implies $\mathrm{lfp}T \leq x$. The idea of using widening has already been used in the abstract interpretation of programs in procedural languages [CH78].

We now describe what we call *affine widening*. To simplify notations, let us omit subscripts concerning predicate name and suppose there is only one predicate in a program. A_i is an affine subspace w.r.t. the predicate. If A_i is a postfixpoint of T, we are done with a correct IC. Otherwise, we widen $T(A_i)$ to its affine hull A_{i+1}. Suppose A_l is such a postfixpoint that it is found for the first time in the computation of $A_0 = \emptyset, A_1, \ldots$.

Theorem 4.3: The sequence $A_0, A_1, \ldots, A_l$ is finite.

---

[1] Brodsky and Sagiv studied inference of disjunction of inequalities between two argument positions. With a transformation similar to our recursive transformation, they proved the question on convergence is undecidable[BS91]

[2] If $x \geq T(x)$ ($x \leq T(x)$), then x is a postfixpoint (prefixpoint) of T.

Proof: It is enough to show the dimension of A_{i+1} is greater than that of A_i for $i = 0, \ldots, l-1$ since A_i's are affine subspaces in R^n where n is the arity of a predicate associated with the recursive transformation T. $A_i \subset T(A_i)$ since A_i is not a postfixpoint of T and $T(A_i) \subseteq A_{i+1}$ since A_{i+1} is an affine hull of $T(A_i)$. So A_{i+1} is an affine subspace properly including A_i; that is, the dimension of A_{i+1} is greater than that of A_i. □

Although affine widening sometimes widens to "no constraints", it provides yet useful IC with many practical programs.

Example 4.7: Let us consider `append` procedure with a constraint that all list elements are integers.

```
a([], T, T).
a([X|U], V, [X|W]) :- integer(X), a(U,V,W).
```

The purpose of introducing the constraint `integer(X)` is to set the termsize of `X` to zero. The abstracted version of the above `append` procedure is as follows:

$a(0, t, t) \leftarrow t \geq 0.$

$a(2 + x + u, v, 2 + x + w) \leftarrow (x, u, v, w) \geq \vec{0}, x = 0, a(u, v, w).$

Using recursive transformation without affine widening, the transformation does not converge finitely since it expands the input convex set inch by inch at every iteration. We now try with affine widening. $A_1 = T(\emptyset) = \{a_1 = 0, a_2 = a_3\}$ is an affine subspace, which is one-dimensional. $\text{frame}(A_1 \sqcup T(A_1)) = (\{(0,0,0),(1,0,1)\}, \{(0,1,1)\})$, so its affine hull is $A_2 = \{a_3 = a_1 + a_2\}$, which is two-dimensional. With one more iteration, we find that A_2 is a postfixpoint of T (fortunately, a fixpoint). □

4.4 Translativeness Property

Let $\vec{p}$ be a vector of argument sizes in a head and $\vec{p'}$ a vector of argument sizes in a recursive subgoal. All nonrecursive subgoals are replaced by their ICs. Then an abstract rule can be viewed as a polycone in R^{2n}, so this polycone can be generated by its extreme points and rays.

Example 4.8: Consider the naive `reverse` procedure.

```
r([],[]).
r([X|U],W) :- r(U,V), a(V,[X],W).
```

Applying listsize abstraction to the procedure, we have:

$r(0,0)$.

$r(1+u,w) \leftarrow (u,v,w) \geq \vec{0},\ r(u,v),\ a(v,1,w)$.

The analysis for `a` supplies an IC: $a_1 + a_2 = a_3$. Replacing $a(v,1,w)$ by $(a_1 + a_2 = a_3)[a_1/v, a_2/1, a_3/w]$ reduces to the following abstract procedure.

$r(0,0)$.

$r(1+u,w) \leftarrow (u,v,w) \geq \vec{0},\ r(u,v),\ v+1 = w$.

Let $\vec{r}$ and $\vec{r'}$ denote the argument sizes of head and recursive subgoal, resp. So we have the following parametric representation for r_1, r_2, r'_1, r'_2.

$$r_1 = 1+u,\ r_2 = w,\ r'_1 = u,\ r'_2 = v,\ v+1 = w,\ u \geq 0,\ v \geq 0,\ w \geq 0$$

The parametric representation generated by the extreme points and rays is:

$$\begin{array}{rcrcr} r_1 & = & 1 & +u & \\ r_2 & = & 1 & & +v \\ r'_1 & = & 0 & +u & \\ r'_2 & = & 0 & & +v \\ u & \geq & 0 & & \\ v & \geq & 0 & & \end{array}$$

□

Definition 4.3: (translativeness) A natural transformation Ψ associated with a linear recursive rule is viewed as a polycone in R^{2n} when all nonrecursive subgoal are replaced by their ICs. Suppose the parametric representation generated by extreme points and rays of the polycone in R^{2n} is in the form of :

$$\Psi \equiv \begin{cases} \begin{pmatrix} \vec{p} \\ \vec{p'} \end{pmatrix} = \begin{pmatrix} A \\ 0 \end{pmatrix} \vec{\lambda} + \begin{pmatrix} B \\ 0 \end{pmatrix} \vec{\mu} + \begin{pmatrix} P_1 \\ P_2 \end{pmatrix} \vec{\rho} \\ \sum \lambda_i = 1 \\ \lambda_i \geq 0, \mu_j \geq 0, \rho_k \geq 0 \end{cases} \tag{4.9}$$

where P_1 and P_2 are permutations. Substituting $\rho = P_2^{-1} y$ boils down to:

$$\Psi \equiv \begin{cases} \vec{p} = A\vec{\lambda} + B\vec{\mu} + P\vec{p'} \\ \sum \lambda_i = 1 \\ \lambda_i \geq 0, \mu_j \geq 0 \end{cases} \tag{4.10}$$

where P is $P_1 P_2^{-1}$. If a natural transformation Ψ can be reduced to Eq. 4.10, Ψ is *P-translative*.

Let Δ_0 denote a polycone represented by:

$$\Delta_0 \equiv \begin{cases} \vec{p} = A\vec{\lambda} + B\vec{\mu} \\ \sum \lambda_i = 1 \\ \lambda_i \geq 0, \mu_j \geq 0 \end{cases} \tag{4.11}$$

$\Psi(\Delta)$ is indeed a vector sum of Δ_0 and $P\Delta$:

$$\Psi(\Delta) = \Delta_0 + P\Delta$$

□

Now we introduce some useful operations on polycones.

Definition 4.4: Let $\Delta, \Delta_1, \Delta_2$ be polycones in R^n. The vector sum, scalar multiplication, and linear transformation of polycones are defined as follows:

1. (vector sum) $\Delta_1 + \Delta_2 = \{x + y | x \in \Delta_1, y \in \Delta_2\}$
2. (scalar multiplication) $\lambda\Delta = \{\lambda x | x \in \Delta\}$
3. (linear transformation) $A\Delta = \{Ax | x \in \Delta\}$ where A is a matrix.

□

Lemma 4.1: The following equalities on polycones hold.

1. $A(\Delta_1 + \Delta_2) = A\Delta_1 + A\Delta_2$
2. $A(\Delta_1 \sqcup \Delta_2) = A\Delta_1 \sqcup A\Delta_2$
3. $\Delta_1 + (\Delta_2 \sqcup \Delta_3) = (\Delta_1 + \Delta_2) \sqcup (\Delta_1 + \Delta_3)$

□

Suppose we have k base polycones and l P-translative natural transformations. Then we can simplify the recursive transformation T as follows:

$$\begin{aligned} T(\Delta) &= B_1 \sqcup \cdots \sqcup B_k \sqcup (\Delta_1 + P\Delta) \sqcup \cdots \sqcup (\Delta_l + P\Delta) \\ &= (B_1 \sqcup \cdots \sqcup B_k) \sqcup ((\Delta_1 \sqcup \cdots \sqcup \Delta_l) + P\Delta)) \\ &= \Delta_B \sqcup (\Delta_R + P\Delta) \end{aligned}$$

where $\Delta_B = B_1 \sqcup \cdots \sqcup B_k$ and $\Delta_R = \Delta_1 \sqcup \cdots \sqcup \Delta_l$.

We now describe how P-translative natural transformation can be unfolded to I-translative transformation using unfolding, where I is an identity matrix.

Definition 4.5: (Unfolding) Suppose T is P-translative transformation. $T^2(\Delta)$ is obtained by applying T to $T(\Delta)$.

$$\begin{aligned} T(T(\Delta)) &= \Delta_B \sqcup (\Delta_R + P(\Delta_B \sqcup (\Delta_R + P\Delta))) \\ &= \Delta_{B'} \sqcup (\Delta_{R'} + P^2\Delta) \end{aligned}$$

where $\Delta_{B'} = \Delta_B \sqcup (\Delta_R + P\Delta_B)$ and $\Delta_{R'} = \Delta_R + P\Delta_R$. □

Applying at most n unfoldings in which n is the arity of the associated predicate, we can get I-translative transformation since P is a permutation matrix.

Theorem 4.4: Let T be a translative transformation in the form of:

$$T(\Delta) = \Delta_B \sqcup (\Delta_R + \Delta)$$

and the generators of Δ_B and Δ_R are:

$$\begin{aligned} \text{gen}(\Delta_B) &= (\{u_1, \ldots, u_k\}, \{v_1, \ldots, v_l\}) \\ \text{gen}(\Delta_R) &= (\{x_1, \ldots, x_n\}, \{y_1, \ldots, y_m\}) \end{aligned}$$

then

$$\text{gen}(\mathsf{lfp}T) = (\{u_1, \ldots, u_k\}, \{v_1, \ldots, v_l, x_1, \ldots, x_n, y_1, \ldots, y_m\})$$

Proof:

$$\begin{aligned} T^{\infty}(\emptyset) &= \Delta_B \sqcup (\Delta_R + \Delta_B) \sqcup (2\Delta_R + \Delta_B) \sqcup \cdots \sqcup (k\Delta_R + \Delta_B) \sqcup \cdots \\ &= \Delta_B + (\mathbf{0} \sqcup \Delta_R \sqcup 2\Delta_R \sqcup \cdots \sqcup k\Delta_R \sqcup \cdots) \\ &= \Delta_B + \{\lambda x | x \in \Delta_R, \lambda \geq 0\} \end{aligned}$$

□

In practice, linearly recursive logic programs relying on "recursion on structures" technique usually satisfy translativeness property. Hence their tight interargument constraints can be found without any iterations.

Example 4.9: The following procedure is intended to divide its first argument into its second and third argument. In order to assure balanced division of the "input" list, the last two arguments are interchanged upon recursion.

```
d([], [], []).
d([X | U], [X | V], W) :- integer(X), d(U, W, V).
```

Applying termsize abstraction to the procedure, we have:

$d(0,0,0)$.

$d(2+u, 2+v, w) \leftarrow (u,v,w) \geq \vec{0},\ d(u,w,v)$.

Clearly, frame(Δ_B) = ($\{0,0,0\}$, $\{\}$). From the second rule, we get a P-translative transformation:

$$\Psi(\Delta) = \Delta_R + P\Delta$$

where

$$\Delta_R \equiv \{\vec{d} = (2,2,0)\}$$

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Applying an unfolding boils down to I-translative transformation:

$$T^2(\Delta) = \Delta_{B'} \sqcup (\Delta_{R'} + \Delta)$$

where frame($\Delta_{B'}$) = $(\{(0,0,0),(2,2,0)\},\{\})$ and frame($\Delta_{R'}$) = $(\{(4,2,2)\},\{\})$. By Theorem 4.2,

$$\mathsf{lfp}T \equiv \begin{cases} d_1 = \lambda_2 + 2\mu_1 \\ d_2 = \lambda_2 + \mu_1 \\ d_3 = \mu_1 \\ \lambda_1 + \lambda_2 = 2 \\ \lambda_1 \geq 0, \lambda_2 \geq 0, \mu_1 \geq 0 \end{cases}$$

It implies that the first argument size is the sum of the second and the third, and the first argument size is greater than the second or the third if the first is positive, which is important information when handling termination proofs. It is noted that Van Gelder's recursive transformation does not converge with this example and his heuristic does not work [VG91]. Sagiv and Brodsky's method can infer only inequalities between two argument positions [BS91]. Recursive transformation with affine widening described in the previous section gives an IC: $d_1 = d_2 + d_3$, which is less precise than the above. □

4.5 Extreme Points of a Polycone

Parametric representation of a polycone in R^n can be viewed as a set of constraints Δ_λ in R^m and an affine transformation Φ from R^m to R^n.

$$\Phi \equiv \begin{cases} \vec{x} = \vec{a} + A\vec{\lambda} \\ \Delta_{\vec{\lambda}} \end{cases} \tag{4.12}$$

Lemma 4.2: The extreme points in $\Phi(\Delta)$ are affine images of the extreme points of Δ:

$$\mathrm{ep}(\Phi(\Delta)) \subseteq \{\vec{a} + A\vec{p} \mid \vec{p} \in \mathrm{ep}(\Delta)\}$$

where $\mathrm{ep}(S)$ is the set of extreme points in S.

The extreme rays in $\Phi(\Delta)$ are images of linear transformation of the extreme rays of Δ:

$$\mathrm{er}(\Phi(\Delta)) \subseteq \{A\vec{r} \mid \vec{r} \in \mathrm{er}(\Delta)\}$$

where $\mathrm{er}(S)$ is the set of extreme rays in S. □

Δ is a polycone, so there are finite number of extreme points and rays. As there is an algorithm to find all extreme points and rays of a polycone in [MR80, VG91], we can effectively compute them (See Appendix B.)

The cost of computing all extreme points and rays depends on the number of parameters and constraints. Before we compute extreme points and rays, we should be able to eliminate redundant parameters and constraints unless the cost of elimination is expensive. Parametric representation of a polycone usually has the constraints in the form of equalities. Gaussian elimination procedure reduces equalities to inequalities, eliminating at most n parameters where n is the number of equalities. By exploiting the fact that polycone is constrained to nonnegativity, we can eliminate redundant inequalities and parameters:

1. $a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b$ where $a_i \geq 0, b \leq 0$ is redundant.
2. $a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b$ where $a_i \leq 0, b = 0$ forces x_i with nonzero coefficient to be zero.
3. $a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b$ where $a_i \leq 0, b > 0$ makes the polycone inconsistent

An implicit equality corresponds to the elimination of one parameter and one constraint. The cost of finding an implicit equality is equivalent to that of running a linear program. General techniques for redundancy elimination of linear constraints can be found in [KLTZ83, LHM89].

To verify a fixpoint of a recursive transformation T, we need to extract extreme points and rays from points and rays transformed from the constraint set. Suppose we have k points $\vec{p_1}, \ldots, \vec{p_k}$ and l rays $\vec{r_1}, \ldots, \vec{r_l}$. A ray $\vec{r_i}$ is nonextreme if it is nonnegative linear combination of the other rays. In other words, a ray $\vec{r_i}$ is nonextreme if the following linear system is solvable:

$$\vec{r_i} = \sum_{j \neq i} \mu_j \vec{r_j}, \ \mu_j \geq 0 \tag{4.13}$$

Similarly, a point $\vec{p_i}$ is nonextreme if the following system is solvable.

$$\vec{p_i} = \sum_{j \neq i} \lambda_j \vec{p_j} + \sum \mu_j \vec{r_j}, \ \sum \lambda_j = 1, \ \lambda_j \geq 0, \ \mu_j \geq 0 \tag{4.14}$$

Thus we can eliminate nonextreme points and rays by testing the solvability of a certain linear system. Solvability of a linear system can be tested theoretically in polynomial time by Khachiyan's ellipsoid method or Karmarkar's method [Sch86]. It also can be tested by exponential time algorithm such as Fourier-Motzkin elimination or the first phase of simplex method. Those polynomial time algorithms have been reported to be impractical unless the input size is substantially large. The exponential-time algorithm will work if we can maintain small input sizes.

Fourier-Motzkin elimination stops upon finding an inconsistent inequality while the first phase of Simplex algorithm stops upon finding a feasible solution. Thus in case many nonextreme points (or rays) are present, the first phase of Simplex algorithm must be used. In any case maintaining small input sizes is a key to using exponential-time algorithms successfully.

As we deal with polytopes in the following, we will use "vertex" instead of "extreme point". Affine images of vertices in a polytope may contain a high degree of redundancy. It is the case that the number of neighbors adjacent to a vertex is usually far less than that of vertices in polytopes. Theorem 4.5 says it is enough to show that an affine image v_0 is a convex combination of only its neighbors on a graph induced by affine transformation, in order to show that v_0 is not a vertex. (We are working on the extension to general polyhedral convex sets.) Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of vertices and $E = \{e_1, e_2, \ldots, e_m\}$ the set of edges in Δ. Thus $G = (V, E)$ represents the adjacency graph on vertices of a polytope. This graph can be constructed using a variant of Simplex algorithm which finds all basic feasible solutions rather than an optimal solution. A graph $G_\Phi = (V_\Phi, E_\Phi)$ is one induced by an affine transformation Φ:

$$\begin{aligned} V_\Phi &= \{\Phi(v) \mid v \in V\} \\ E_\Phi &= \{(\Phi(v_1), \Phi(v_2)) \mid \Phi(v_1) \neq \Phi(v_2),\ (v_1, v_2) \in E\} \end{aligned} \tag{4.15}$$

Definition 4.6: [Grü67] Let $G = (V, E)$ be a graph, M a set of vertices of G, and v, u two vertices of G which do not belong to M. M *separates* v from u provided every path connecting v and u contains some vertex in M. □

Conjecture 4.1: Let v and u be two vertices in an adjacency graph G of a polytope P separated by $M \subset V$. Then the line segment connecting v and u passes through the convex hull of M. □

Note the following theorem is based on Conjecture 4.1.

Theorem 4.5: Suppose Conjecture 4.1 holds. Let $x_0 \in V_\Phi$ and $\{y_1, \ldots, y_k\}$ be the set of vertices adjacent to x_0. Then x_0 is a vertex in $\Phi(\Delta)$ iff x_0 is not a convex combination of y_i's.

Proof: (Sketch)

($\Rightarrow$). Trivial.

($\Leftarrow$). Let $\{z_1 \ldots z_l\}$ be the set of vertices not adjacent to x_0 in V_Φ (If there exist no such z_i's, proof is trivial).

Let $\{p_1, \ldots, p_m\}$ be the set of vertices in V which are transformed to x_0, and $\{q_1, \ldots, q_n\}$ the set of vertices in V which are transformed to $\{y_1, \ldots, y_k\}$, and $\{r_1, \ldots, r_o\}$ the set of vertices in V which are transformed to $\{z_1 \ldots z_n\}$.

Each line segment connecting p_i and r_j, $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, o\}$ must pass through ConvexHull$\{q_1, \ldots, q_n\}$ by Conjecture 4.1. Since affine transformation preserves the convexity, each line segment connecting x_0 and z_j, $j \in \{1, \ldots, l\}$ must pass through ConvexHull$\{y_1, \ldots, y_m\}$. Thus x_0 is a vertex. □

4.6 Summary

We formalized a method of deriving constraints among argument sizes and provided an affine widening operation to accelerate the convergence in finitely many steps, yet providing useful interargument constraints.

We defined a class of linear recursive logic programs in terms of translativeness property. For such a class, tight interargument constraints are automatically given via the analysis of the structure of transformations. In practice, most of logic programs are linear recursive and rely on "recursion on structure" technique, hence satisfying translativeness property.

input program	append	merge	perm	mergesort	parser
cputime in milliseconds	239	499	650	1270	859
ratio	8.00	3.12	7.22	5.08	6.14

Table 4.1: Performance measures of inference of interargument constraints

The most costly operations in the procedure of handling linear constraints are the projections and finding all the extreme points and rays of polycones. We presented a method exploiting information on adjacency of extreme points, which outperforms methods using Fourier-Motzkin variable elimination.

We implemented the method in Prolog. The complete system consists of 2297 lines of Prolog code. Some performance measures are listed in Table 4.1. The input progrmas are shown in Appendix D. The ratio was defined in Section 3.9. Some discussion on the performance results also can be found in that section. The programs were tested on SUN4 sparc station.

We are working on the extension of translativeness to nonlinear recursion.

5. Relational Groundness Analysis

5.1 Introduction

Logic programming has been successfully used as a tool for several areas including compiler writing, expert system design, natural language processing, hardware design, and knowledge-base design. One of the most attractive features of Prolog is bidirectional use of arguments for input, output, or both; however, in practical programs most procedures do not make this sophisticated use of arguments. Using mode information allows compilers to produce more specific code which results in substantial speedup [Mel85]. With this regard, Warren introduced explicit mode declarations to help the compiler produce better code [War77]. But mode declarations must be verified since wrong annotation may introduce subtle errors in program executions.

Another approach is to infer mode declarations automatically via *abstract interpretation* [Mel87, BJCB87, MU87, DW88]. Abstract interpretation has been used as standard means for dataflow analysis since it was placed on a solid semantic basis by Cousot and Cousot [CC77] (See [CC92] for the theory and applications to logic programming).

Early work done by Mellish [Mel81] produced erroneous results since aliasing effects resulting from unification was not considered, which was corrected later [Mel87].

Mannila and Ukkonen [MU87] used simple two-valued abstract domains $\{ground, any\}$, hence $free$ mode [1] cannot be inferred ($ground, free, any$ are abstractions of ground terms, free variables, all (ground or nonground) terms, respectively). In addition, their method could not handle the problem with aliased variables accurately.

Bruynooghe *et al.* [BJCB87] suggested multi-passes algorithm (repeat previous call strategy) to resolve aliasing problem, which are considered to be costly if the strategy is applied globally.

[1] In their paper, *nonground* means possibly nonground, which means *any*

Debray and Warren [DW88] presented an algorithm to give a sound and efficient treatment of aliasing. However their method produces less precise mode information since they use conservative local analysis in which all unsafe instantiations are replaced by *any*.

In this chapter, we study relational abstract domains which describe possible groundness relationships among arguments. Like [MU87], we only analyze groundness, however, our method provides great accuracy with respect to groundness. It is also noted that success of correctness proofs like program termination relies on preciseness of groundness information [UVG88, Plü90b, SVG91].

Let us consider a call `p(A, f(A,B))` in which we have no instantiation information on `A` and `B`. Hence the goal is abstracted as $p(any, any)$ in the previous cited methods. But more careful look at the call gives us the possible combinations of call patterns $\{p(0,0), p(0,1), p(1,1)\}$ where 0 and 1 are *ground* and *nonground* respectively to keep notations simple. That is, if the second argument is *ground*, then the first has no other choice but *ground*. Suppose we have a clause `p(U, f(a,b)).` With the call pattern $p(any, any)$, the success pattern after the call to the clause is $p(any, ground)$ whereas with our call pattern, the success pattern is $p(ground, ground)$. Preciseness of success patterns affect the subsequent call patterns immediately. It should be mentioned that if we use a table of possible groundness relationships among arguments as shown above, we will end up with combinatorial blowup.

We now introduce simple boolean constraints to denote groundness relation. Roughly, a term can be abstracted as a logical disjunction of variables occurring in the term. For example, $p(a, a \vee b))$ is the abstraction of the goal `p(A, f(A,B))` where a and b are propositional variables corresponding to `A` and `B` and $\vee$ is boolean OR.

Using positive propositional formula (called domain PROP) for groundness analysis has been studied by Marriott and Søndergaard [MS89]. The domain was further studied by Cortesi *et al.* [CFW91]. Our abstract domain is simpler than PROP since we only use positive proposition formula forming so-called a "boolean cone". For such formula, we can easily find a minimal generator set which is a unique minimal representation. Therefore,

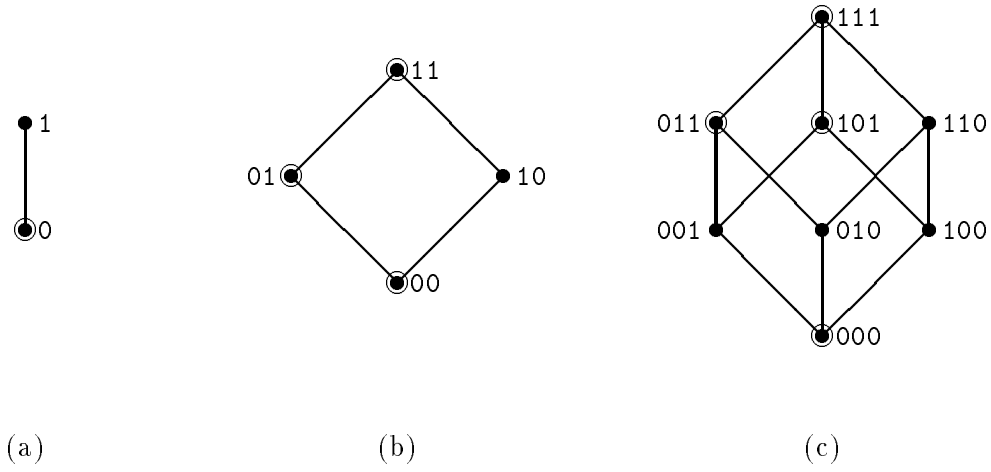


Figure 5.1: Diagram of ground analysis

testing if two formula are equivalent can be done efficiently. Main contribution of our study is to introduce novel concepts on boolean cones and develop operations to accomplish efficiently relational groundness analysis and to provide bottom-up groundness analysis using propositional formula.

In Section 5.2, we describe notations used through the chapter and introduce groundness abstraction of logic programs, and concepts, properties, and operations on boolean cones.

Section 5.3 describes some simplification procedures to remove redundancy in groundness constraints.

Section 5.4 presents bottom-up groundness analysis mimicking immediate consequence operator.

Section 5.5 summarizes the chapter.

The framework similar to our relational bottom-up analysis has been investigated on real arithmetic domain to derive constraints among argument sizes by Van Gelder [VG91].

5.2 Basic Concepts

5.2.1 Abstraction

We now describe abstraction of terms, atoms, and clauses. Logic programs can be abstracted by groundness relationships among terms. Informally speaking, groundness abstraction of a term carries the information that if all the variables in the term are ground, then the term is ground. We choose the boolean value 0 to denote *ground*. So $x \vee y$ is the groundness abstraction of `f(X,Y)`. That is, if x (`X`) and y (`Y`) are 0 (*ground*), then $x \vee y$ (`f(X,Y)`) is 0 (*ground*). Let $vars(t)$ denote a set of variables in a term t. We shall use lowercase letters for the boolean variables corresponding to logical variables.

Example 5.1: Figure 5.1 illustrates groundness relationships of three atoms: (a) `p(a)`, (b) `p(U, f(U,V))`, and (c) `p(U, V, f(U,V))`. The lattices (a), (b), and (c) in Figure 5.1 are possible relationships for unary, binary, and ternary predicate, respectively. Circled dots represent possible relationships for the three atoms listed above. For example, for the atom `p(U, f(U,V))`, both arguments can be either ground (00) or free (11), and the first argument can be ground and the second free (01). But it is impossible to have the first free and the second ground. □

For notational convenience, we shall use p_i for a boolean variable representing groundness abstraction of i-th argument of predicate p. Boolean terms are terms built from boolean variables and connective boolean OR $\vee$. Note that we do not use NOT and AND in boolean terms. We often use a vector notation to denote a vector of boolean terms. For example, $(a_1, a_2) \vee (b_1, b_2) \leftrightarrow (c_1, c_2)$ denotes: $a_1 \vee b_1 \leftrightarrow c_1$ and $a_2 \vee b_2 \leftrightarrow c_2$. Substitution $[x_1/e_1, \ldots, x_n/e_n]$ is defined in an usual way; that is, a boolean variable x_i is replaced by a boolean term e_i. Using vector notation, it is denoted by $[\vec{x}/\vec{e}]$.

Definition 5.1: Let t be a logical term. Then $\alpha(t)$ is a groundness-abstracted term.

$$\alpha(t) = \begin{cases} 0 & \text{if } vars(t) \text{ is empty} \\ \vee_{x \in vars(t)} x & \text{otherwise} \end{cases}$$

Let p be n-ary predicate. Then groundness-abstracted atom is as follows.

$$\alpha(p(t_1, t_2, \ldots, t_n)) = p(\alpha(t_1), \alpha(t_2), \ldots, \alpha(t_n))$$

Abstraction of clauses is obtained by applying abstraction to each atom occurring in the clauses. □

Example 5.2: Let us consider the usual `append` procedure.

```
a([], U, U).
a([X|U], V, [X|W]) :- a(U, V, W).
```

Applying groundness abstraction to each clause transforms it to groundness-abstracted procedure.

$a(0, u, u)$.

$a(x \vee u, v, x \vee w) \leftarrow a(u, v, w)$.

Since terms are boolean expressions, unification must be replaced by boolean constraint solving in the execution of the groundness-abstracted procedure. □

5.2.2 Relational Abstract Domains

We now introduce boolean cones and their constraint representation. We also examine some operations like OR-closure as least upper bound operation and equivalence of two boolean cones, which are useful in finding the fixpoint of a certain transformation concerning groundness relation.

Definition 5.2: Let $a, b \in \{0,1\}^n$. $C \subseteq \{0,1\}^n$ is a *boolean cone* if

1. $\mathbf{0} \in C$
2. If $a \in C$ and $b \in C$, then $a \vee b \in C$

□

Example 5.3: $\{(0,0,0),(1,1,0),(1,0,1),(1,1,1)\}$ is a boolean cone whereas $\{(0,0,0),(1,1,0),(1,0,1)\}$ is not. □

We use the word "boolean cones" since they have the properties similar to those of convex cones in R^n. We now examine some useful properties of boolean cones.

Lemma 5.1: Intersection of two boolean cones C_1 and C_2 is a boolean cone C_3.

Proof: Clearly **0** is in C_3. Suppose a and b are in C_3. Therefore a and b are in both C_1 and C_2. Since C_1 and C_2 are boolean cones, $a \vee b$ is in both C_1 and C_2. Then $a \vee b$ is also in C_3. □

Example 5.4: Let $C_1 = \{(0,0,0),(0,1,1),(1,0,1),(1,1,1)\}$ and $C_2 = \{(0,0,0),(1,1,0),(1,1,1)\}$. C_1 and C_2 are boolean cones. $C_1 \bigcap C_2 = \{(0,0,0),(1,1,1)\}$ is also a boolean cone. □

Lemma 5.2: The projection of boolean cone in $\{0,1\}^n$ onto $\{0,1\}^m$ where $m \leq n$ is also a boolean cone.

Proof: Trivial. □

We now introduce the concept similar to extreme rays of convex cones in R^n, which we call *generators*. Generator sets serve as the unique, minimal representation of boolean cones. Therefore testing equivalence of two cones reduces to comparing two generator sets. Boolean OR of two vectors is a vector of the results of componentwise OR.

Definition 5.3: Let C be a boolean cone C, and $\alpha \in C$. Then α is a *generator* of C if $\alpha \neq \mathbf{0}$ and cannot be the boolean OR of any other points in C. A *minimal generator set* $gen(C)$ of C is a set of all generators in C. A generator set is the set including a minimal generator set and the points which can be generated by the minimal generator set.

Example 5.5: $gen(\{(0,0,0),(0,1,1),(1,0,1),(1,1,1)\} = \{(0,1,1),(1,0,1)\}$ □

We can get a minimal generator set from a generator set by removing non-generators. A non-generator is a boolean OR of some generators. Testing if a point is a generator can be done as follows. Suppose we have a generator set $\{y_0, y_1, \ldots, y_n\}$ and we want to test if y_0 is a generator. If y_0 is not a generator, then there must be $a_1, \ldots, a_n$ such that $a_i \in \{0,1\}$ and

$$y_0 = a_1 \cdot y_1 \vee \cdots \vee a_n \cdot y_n.$$

If i-th component of y_0 is 0 and i-th component of y_j is 1, then a_j has no other choice but zero. For other a_j's, we can assign 1's. Then we can evaluate the above formula. If it is true, y_0 is a non-generator, otherwise a generator. This test is done in linear time in terms of the size of the generator set.

Lemma 5.3: There exists a unique minimal generator set $gen(C)$ for any boolean cone C.

Proof: Suppose G_1 and G_2 are two generator sets for a boolean cone C and $G_1 \neq G_2$. Therefore, there exists x such that $x \in G_1$ and $x \notin G_2$. Since G_2 is a generator set, x can be a boolean OR of some points $g_1, \ldots, g_k$ in G_2. Since $g_1, \ldots, g_k$ are members of C, they are in G_1 or boolean combinations of generators in G_1. That concludes x is a boolean combination of generators in G_1, which is a contradiction. □

Let $\mathcal{U}$ be the set of all boolean cones in $\{0,1\}^n$. Equipped with the subset ordering $\subseteq$, $\mathcal{U}$ forms a complete lattice with an empty set as the least element, $\{0,1\}^n$ as the greatest element, set intersection $\bigcap$ as greatest lower bound operation, OR-closure $\bigsqcup$ as least upper bound operation as defined below.

Definition 5.4: The *OR-closure* of n cones $C_1, C_2, \ldots, C_n$ is the set of points which are the boolean OR of any points in C_i's; it is denoted by $\bigsqcup\{C_1, C_2, \ldots, C_n\}$. □

Lemma 5.4: Let $C_1, C_2, \ldots, C_n$ be boolean cones. $\bigsqcup\{C_1, C_2, \ldots, C_n\}$ is the set generated by the union of the generator sets of $C_1, C_2, \ldots, C_n$.

Proof: It follows from the definition of a boolean cone and a generator set. □

Example 5.6: Let $C_1 = \{(0,0,0),(0,1,0),(0,1,1)\}$ and $C_2 = \{(0,0,0),(0,1,0),(1,1,0)\}$ and C be $\bigsqcup\{C_1, C_2\}$. Their generator sets are $gen(C_1) = \{(0,1,0),(0,1,1)\}$ and $gen(C_2) = \{(0,1,0),(1,1,0)\}$. Their union is, $gen(C) = \{(0,1,0),(0,1,1),(1,1,0)\}$. Hence $C = \{(0,0,0),(0,1,0),(0,1,1),(1,1,0),(1,1,1)\}$. □

5.2.3 Groundness Constraints

The success of relational groundness analysis relies on whether we have efficiently computable form of relations. Groundness relationships among arguments with respect to a predicate can be represented in the form of a set of boolean constraints.

Definition 5.5: Let $\vec{e} = (e_1, e_2, \ldots, e_n)$ be boolean terms and $c_1, c_2, \ldots, c_m$ boolean equalities. Let $\vec{p} = (p_1, \ldots, p_n)$ be a vector of boolean variables corresponding to groundness abstraction of arguments of predicate p.

$$G = \{\vec{p} \leftrightarrow \vec{e}, c_1, c_2, \ldots, c_m\}$$

is called a *groundness constraint* for a predicate p. If $m = 0$, it is called a *simplified groundness constraint* for p. □

Theorem 5.1: A groundness constraint defines a boolean cone.

Proof: It is enough to show that one conjunction of a groundness constraint defines a boolean cone, since the intersection and/or projection of boolean cones is also a boolean cone. Each conjuction G is in the following form:

$$x_1 \vee \cdots \vee x_n \leftrightarrow y_1 \vee \cdots \vee y_m.$$

The model of this formula G is the set of all the satisfiable assignments. Let $(v_1, \ldots, v_k)$ be a tuple of all the variables appearing in G. $(0, \ldots, 0)$ is clearly a satisfiable assignment. Let $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ be two satisfiable assignments of G. Then $(a_1 \vee b_1, \ldots, a_n \vee b_k)$ is also a satisfiable assignment, since $(a_1 \vee b_1, \ldots, a_k \vee b_k)$ has more 1's than $(a_1, \ldots, a_k)$ or $(b_1, \ldots, b_k)$. □

5.3 Simplifications and Normal Forms

The most costly operation is to test equivalence of two groundness constraints, which must be performed at each iteration of transformation concerning abstract interpretation. This operation is tantamount to finding generators of projection of cones, since generators are unique, minimal representation of cones. In most cones associated with practical programs, generators can be obtained by the following *simplification rules*.

- $x \vee y \vee \cdots \vee z \leftrightarrow 0$ is equivalent to $x \leftrightarrow 0, y \leftrightarrow 0, \ldots, z \leftrightarrow 0$.
- If $x \vee y \vee \cdots \vee z \leftrightarrow w$ is a constraint and w does not appear on the left-hand side substitute the left-hand side into every place where w appears. If w appears nowhere, delete the constraint.
- If x appears on the same side as y in every occurrence, x can be deleted.

A simplified groundness constraint corresponds to a set of points including generators, which reduces to a generator set by removing redundancy. This is explained in the following example.

Example 5.7: Let $C = \{\vec{p} \leftrightarrow (x \vee w, y \vee w, x \vee y \vee w)\}$ be a simplified groundness constraint. $(x \vee w, y \vee w, x \vee y \vee w) \leftrightarrow x \cdot (1,0,1) \vee y \cdot (0,1,1) \vee w \cdot (1,1,1)$ where $\cdot$ can be viewed as scalar multiplication and $\vee$ as componentwise boolean OR. Then $(1,0,1), (0,1,1)$, and $(1,1,1)$ are candidates for generators, and $(1,1,1)$ is redundant since $(1,1,1) = (1,0,1) \vee (0,1,1)$. So $gen(C) = \{(0,1,1), (1,0,1)\}$. This example also explains how to generate groundness constraints from generators. □

Let us turn our attention to groundness constraints which cannot be simplified by the above simplification rules. One way to find generators is to transform equalities with boolean terms to ones with real arithmetic terms. That is,

$$x_1 \vee x_2 \vee \ldots \vee x_n \leftrightarrow y_1 \vee y_2 \vee \ldots \vee y_m$$

can be transformed to

$$\begin{aligned}
&x_1 + x_2 + \ldots + x_n \geq y_1 \\
&x_1 + x_2 + \ldots + x_n \geq y_2 \\
&\vdots \\
&x_1 + x_2 + \ldots + x_n \geq y_m \\
&y_1 + y_2 + \ldots + y_m \geq x_1 \\
&y_1 + y_2 + \ldots + y_m \geq x_2 \\
&\vdots \\
&y_1 + y_2 + \ldots + y_m \geq x_n \\
&x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \in \{0,1\}
\end{aligned}$$

It is easy to prove that both have the same set of solutions, and extreme rays of the linear arithmetic systems are vectors of 0 or 1's after being normalized so that one of their positive components is 1.

Theorem 5.2: If x_0 is a generator of a boolean cone represented by a groundness constraint, then it is a normalized extreme ray of a convex cone represented by a linear arithmetic constraint transformed from the groundness constraint.

Proof: Suppose x_0 is a generator and it is not a normalized extreme ray. Since x_0 is not a normalized extreme ray, x_0 is a positive combination of some normalized extreme rays $y_1, \ldots, y_n$, that is,

$$x_0 = a_1 y_1 + \cdots + a_n y_n \tag{5.1}$$

where a_i is positive. Let us consider a sum of $y_1, \ldots, y_n$, that is,

$$x'_0 = y_1 + \cdots + y_n \tag{5.2}$$

The value $x'_{0i} > 0$ whenever $x_{0i} = 1$ and the value $x'_{0i} = 0$ whenever $x_{0i} = 0$. Therefore Eq. 5.1 implies Eq. 5.2. In fact, Eq. 5.2. is exactly how we compute

$$x_0 = y_1 \vee \cdots \vee y_n. \tag{5.3}$$

Since $y_1, \ldots, y_n$ are the points in the boolean cone, x_0 is not a generator. Contradiction. □

As there are well-known methods to find extreme rays of convex cones (same as finding extreme points of convex polytopes) [VG91, Las90a], we can find candidates for generators for the corresponding boolean cones by removing redundancy.

5.4 Bottom-Up Groundness Analysis

In the preceding sections, we described the relational abstract domain and some useful operations on the domain. We now give a bottom-up abstract interpretation to get success patterns in the form of groundness constraints.

Abstract interpretation consists of abstract domain and abstract operations which mimics base semantics faithfully. Hence bottom-up abstract interpretation mimics fixpoint semantics. Formal framework of bottom-up abstract interpretation can be found in [CC92, MS88].

Now we define a transformation mimicking immediate consequence operator.

Definition 5.6: Suppose we have k predicates whose names are $p, q, \ldots, r$ in an abstract program and p_i is i-th clause having the head predicate p and so forth. Let $P, Q, \ldots, R$ be groundness constraints associated with those predicates, and $\mathcal{U}_p$ the set of all boolean cones determined by the arity of the predicate p and so forth. Let $\mathcal{U} = (\mathcal{U}_p, \mathcal{U}_q, \ldots, \mathcal{U}_r)$. Let x be a vector of groundness constraints $(P, Q, \ldots, R) \in (\mathcal{U}_p, \mathcal{U}_q, \ldots, \mathcal{U}_r)$. *Recursive transformation* T is a mapping from $\mathcal{U}$ to $\mathcal{U}$ and T_p is a mapping from $\mathcal{U}$ to $\mathcal{U}_p$ as given below.

$$
\begin{aligned}
T(x) &= (T_p(x), T_q(x), \ldots, T_r(x)) \\
T_p(x) &= \bigsqcup\{T_{p^1}(x), T_{p^2}(x), \ldots, T_{p^m}(x)\} \\
&\vdots \\
T_{p^i}(x) &= \{\vec{p} \leftrightarrow \vec{a}, Q[\vec{q}/\vec{b}], \ldots, R[\vec{r}/\vec{c}]\}
\end{aligned}
$$

where there are m clauses having the head predicate p and the i-th abstract clause of the predicate p is in the form of

$$p(\vec{a}) \leftarrow q(\vec{b}), \ldots, r(\vec{c})$$

□

Theorem 5.3: There exists the least fixpoint of T and it can be reached in finite number of iterations.

Proof: By its definition, T_{p_i} is clearly monotone, so T is monotone. The domain $(\mathcal{U}_p, \mathcal{U}_q, \ldots, \mathcal{U}_r)$ forms a finite complete lattice, equipped with componentwise subset ordering. □

In practice, it is more efficient to process strongly connected components separately in predicate dependency graphs, starting from leaves in a tree induced by SCCs.

Example 5.8: Continuing with Example 5.2,

$$
\begin{aligned}
T^1 &= T(\emptyset) = \{\vec{a} \leftrightarrow (0, u, u)\} \\
T^2 &= T(T^1) = \bigsqcup\{T^1, \{\vec{a} \leftrightarrow (x \vee u, v, x \vee w, u \leftrightarrow 0, v \leftrightarrow u', w \leftrightarrow u'\}\} \\
&= \{\vec{a} \leftrightarrow (x, v, x \vee v\}
\end{aligned}
$$

input program	append	merge	perm	mergesort	parser
cputime in milliseconds	40	69	60	140	120
ratio	1.33	0.43	0.67	0.56	0.85

Table 5.1: Performance measures of inference of success patterns

Interested readers are invited to verify $T^3 = T^2$, which is the least fixpoint of T. Note that we only use simplification rules to reduce to simplified groundness constraints. The resulting fixpoint shows that in the success pattern of `append` procedure, if the first and second argument are ground, then the third is ground, and vice versa. □

5.5 Summary

We presented a novel relational groundness analysis in this chapter. To compute groundness relations efficiently, we used boolean constraints forming boolean cones. We introduced the concept of generators to facilitate the equivalence test of two sets of boolean constraints.

We implemented the method in Prolog. Some performance measures are listed in Table 5.1. The ratio was defined in Section 3.9. The input programs are shown in Appendix D. The programs were tested on SUN4 sparc station. The domain PROP was implemented by B. Le Charlier and P. Van Hentenryck [LCVH93] and by M. Codish and B. Demoen [CD93]. Our current implementation can take programs in pure Prolog as input. We are working on lifting this limitation. It will be interesting to compare our performance results with their results.

Including freeness analysis and developing efficient test of cone equivalence will be future research directions.

6. Conclusion

We have presented a methodology for termination analysis in logic programming environment. Our method is modular in that we examine one strongly connected component by one in the predicate dependency graph of a program. Nonnegative linear combination of bound argument sizes is used as level-mapping function. To test if this function decreases in well-founded domain, we have transformed the test condition into solvability problem of a linear system using duality theory of linear programming. This method is straightforward and general. Unlike other methods, our method could be extended to nonlinear and mutual recursion with only slight modification.

We often need to know the relationship among argument sizes of a predicate in a subgoal to relate head argument sizes and recursive subgoal argument sizes. We have formalized a method of deriving such relationship using transformation similar to immediate consequence operator. The transformation acts on abstract domain where only the size of a term is considered. This transformation, however, does not necessarily converge. Since it is difficult and unnecessary to find the least fixpoint, we have provided an affine widening approximation to accelerate the convergence of the transformation up to finitely many steps.

We have also defined a class of linear recursive logic programs in terms of translativeness property. For such a class, tight interargument constraints are automatically given via the analysis of the structure of transformations. In practice, most of logic programs are linear recursive and rely on "recursion on structure" technique, hence satisfying translativeness property.

Groundness analysis plays a very important role for global optimization of Prolog compiler. It is also required in order to apply our termination analysis method to Prolog programs rather than knowledge-base systems. Relational groundness analysis has been thought to be costly in published papers. We have suggested to use boolean constraints to represent groundness relationship among arguments of a predicate. This method works

quite effectively with the help of simplification procedures and gives more precise analysis since it resolves the problem due to aliased variables.

The prototype of the implementation is done in Prolog. The performance measures have showed termination analysis is so efficient and precise that it can be practically applied to large programs.

Appendix A. Fourier-Motzkin Elimination

Fourier-Motzkin elimination procedure is used to test if a system of linear constraints are solvable. In this technique a variable is eliminated by "cancelling" all positive occurrences with all negative occurrences, pairwise, creating new rows (with 0 in that variable's column). Then all rows containing a nonzero coefficient for that variable can be eliminated, preserving solvability. If there are only positive (or negative) occurrences of the variable, all the inequalities containing positive (or negative) occurrences are deleted. The reasoning is the variable can have arbitrary values depending on how other variables are constrained.

Consider the following linear constraints:

$$\begin{array}{rrrcl} 2x_1 & +x_2 & +x_3 & \geq & 1 \\ -x_1 & -x_2 & & \geq & 1 \\ x_1 & & -x_3 & \geq & 2 \\ -x_1 & & & \geq & 0 \end{array}$$

Using the matrix notations for denoting the system of inequalities:

$$\left[\begin{array}{rrr|r} 2 & 1 & 1 & 1 \\ -1 & -1 & 0 & 1 \\ 1 & 0 & -1 & 2 \\ -1 & 0 & 0 & 0 \end{array}\right]$$

The vertical line denotes $\geq$.

Try to eliminate the first column corresponding to x_1. Column 1 and 3 have positive coefficients while Column 2 and 4 have negative coefficients. So each pair of rows with positive rows and rows with negative rows, that is Column 1 and 2, Column 1 and 4, Column 3 and 1, Column 3 and 4, can be canceled with appropriate multiplication and addition operations so that the coefficient of x_1 is zero. For example, multiplying Row 2 by 2 and then adding Row 1 and Row 2 makes the coefficient zero. Deleting all the rows with nonzero coefficients gives a new linear system projected on $\{x_2, x_3\}$-space.

$$\left[\begin{array}{rr|l} -1 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & -1 & 3 \\ 0 & -1 & 2 \end{array}\right]$$

After eliminating x_2 and x_3 in the same way, we have the following matrix:

$$\left[\begin{array}{|c} 4 \end{array}\right]$$

Since vertical bar represents $\geq$, it is indeed $0 \geq 4$. It is contradictory, so the initial linear system is unsolvable. This process is inherently exponential-time since it may double the number of inequalities at each variable elimination. How big the number of inequalities grows depends on the order of variables we choose to eliminate. For example, if we choose x_2 to eliminate instead of x_1 in the initial linear system, the resulting number of inequalities after elimination will be 3. This naive Fourier-Motzkin elimination is useless in practice.

Heuristics is used in order to minimize the growth of the number of inequalities. Let P_i and N_i denote the number of positive and negative occurrences of x_i, respectively. $P_i \times N_i$ is the number of new inequalities introduced by eliminating x_i. $P_i + N_i$ is the number of inequalities deleted by eliminating x_i. Therefore, $P_i \times N_i - (P_i + N_i)$ is the difference in the number of inequalities after eliminating x_i. We choose x_i as a pivot variable such that $P_i \times N_i - (P_i + N_i)$ is minimal.

Without sophisticated redundancy check, Fourier-Motzkin elimination is almost useless for sizable input. For example, what follows is an artificially made system of linear inequalities.

$$\left[\begin{array}{cccccc|c} 1 & 1 & -1 & -1 & -1 & 8 & 1 \\ -4 & 2 & -1 & -2 & 1 & -1 & 9 \\ -6 & 2 & 0 & 0 & -1 & 1 & 1 \\ 0 & -1 & -2 & 2 & 0 & -1 & 0 \\ 1 & -1 & 1 & -6 & 0 & 0 & 1 \\ -1 & 2 & -1 & -1 & 1 & 0 & 1 \\ 1 & -1 & -4 & 1 & 0 & -6 & 0 \\ 6 & -2 & 1 & 1 & 0 & 0 & 1 \\ -8 & 4 & -1 & 1 & -1 & -2 & 0 \\ 1 & 1 & 0 & -1 & -2 & 0 & 0 \end{array}\right]$$

Initially, it has 10 inequalities. After eliminating one, two, three, and four variables, the number of inequalities is 24, 93, 1562, ∞, respectively.

However, the linear inequalities generated to test termination condition, have usually many zero coefficients. That's because terms contains only one or two variables. Consider the following CLP(R) program:

```
p(X, N, D) :-
        X = (Xpos - Xneg),
        D = (N - (Xpos - Xneg)),
        ((Xpos - Xneg) - N) >= 0.
p(X, N, D) :-
        X = (Xpos - Xneg),
        D = (N - (Xpos - Xneg)),
        D >= 1,
        (X1pos - X1neg) = (Xpos - Xneg) + 1,
        D1 = (N - (X1pos - X1neg)),
        X1 = (X1pos - X1neg),
        p(X1, N, D1).
```

The matrix corresponds to termination condition.

$$\left[\begin{array}{cccccccccc|c}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
-1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & -1 & 0 & -2 & -2 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right]$$

After eliminating one, two, three, and four variables, the number of inequalities is 14, 13, 12, 9, respectively. Our broad conclusion is that it is adequate and simple to use Fourier-Motzkin elimination for our termination analysis.

Appendix B. Finding All Extreme Points and Rays

In this chapter, we describe how to modify Simplex algorithm to find all extreme points and rays of a polycone. Simplex algorithm assumes the feasible region is bounded and visits basic feasible solutions (extreme points) in such a way that they maximize (or minimize) the object function.

In our setting, we need to find all basic feasible solutions (for short, BFS). Since BFSs are connected and we can move from BFS to BFS by pivoting operation, we can use exhaustive graph search algorithm such as depth-first search. The question is how we can identify extreme rays.

The simplex algorithm of linear programming can be modified to find the extreme points and directions of a polycone. Background on this algorithm can be found in Papadimitriou and Steiglitz [PS82, Ch. 2], and elsewhere; we review the essentials briefly. Assume we have a linear programming problem in a *standard form*, except for the objective function (which may be treated as 0). That is, we have a linear system

$$A\xi = b \qquad \xi \geq 0 \tag{6.2}$$

that describes the polycone, where A is an $m \times N$ matrix of full rank, and $m < N$. (This ξ includes x and the slack (independent) variables of the generalized Tucker representation; its arity is N.) Recall that a *basis* for the problem is a set of m linearly independent columns of A, designated $A_{\mathcal{B}(i)}$ for $1 \leq i \leq m$. B denotes the nonsingular matrix composed of the basis columns of A. Here $\mathcal{B}$ is an m-element subset of $\{1, \ldots, N\}$ in a fixed sequence.

An extreme point is a nonnegative vector P such that

$P_k = 0$ for $k \notin \mathcal{B}$.

$P_{\mathcal{B}(i)} = i$-th component of $B^{-1}b$.

It is well known that basic feasible solutions correspond to vertices (extreme points) of the polycone, and that they can be enumerated by pivoting from one basis to another.

Suppose a column, say A_j, in the simplex tableau is all nonpositive. An extreme ray is a nonnegative vector R such that

Phase I:

Find first feasible basis.

Try every possible combination of pivot columns

and rows until you get a feasible basis.

If you find a row whose coefficients are all nonnegative

and whose constant is negative while pivoting, or

there is no feasible basis,

then the linear system is inconsistent, stop.

Phase II:

DFS(basis, tableau)

Mark the basis visited.

Print an extreme point and rays (if any).

/* NONDETERMINISM HERE */

Pick a column (entering variable) which is not in basis

and at least one of whose members is positive.

(If the column is all nonpositive, it's an extreme ray.)

Find rows whose θ is minimum so the entering variable

guarantees new basis forms an extreme point.

If no such column and row is found, fail.

else pivot on tableau to produce new tableau.

DFS(new neighboring basis, new tableau).

Figure B.1: Algorithm to find all extreme points and rays

$R_j = 1$.

$R_k = 0$ for $k \neq j, k \notin \mathcal{B}$.

$R_{\mathcal{B}(i)} =$ - i-th component of A_j.

The reasoning is we can move from P along R indefinitely. More details can be found in [VG91, CH78].

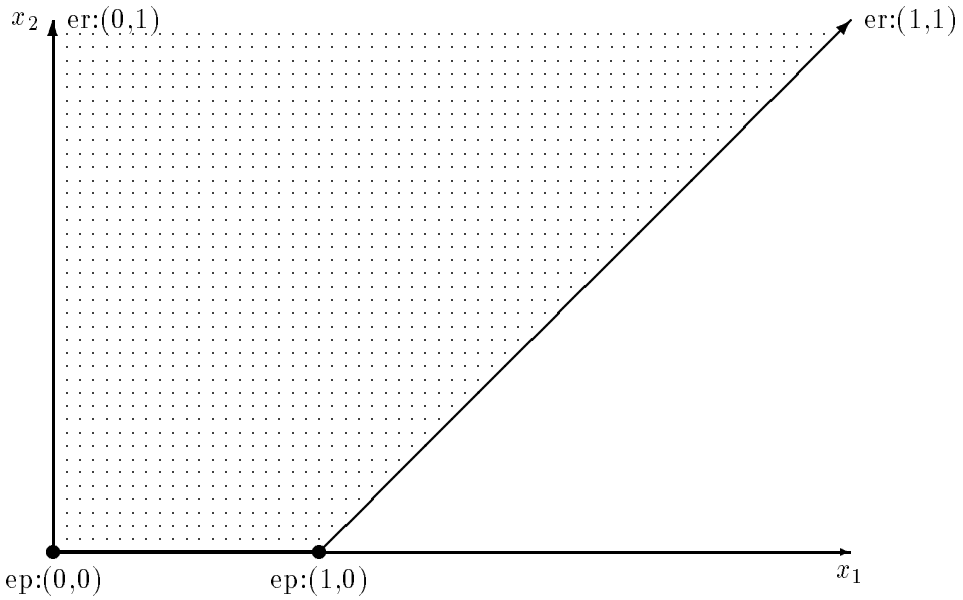


Figure B.2: Extreme points and extreme rays of a polycone.

Consider the following system of linear inequalities. Suppose x_1 and x_2 are constrained to nonnegativity.

$$\begin{aligned} -x_1 \quad &\leq \quad 0 \\ -x_2 \quad &\leq \quad 0 \\ x_1 \quad -x_2 \quad &\leq \quad 1 \end{aligned} \tag{B.1}$$

The shaded area of Figure B.2 depicts the feasible region. As shown in the picture, we have two extreme points: (0,0) and (1,0) and two extreme rays: (0,1) and (1,1). In what follows, we describe how to find those extreme points and rays with this example. By adding slack variables x_3, x_4, and x_5 and putting the system into matrix form, we have a tableau in standard form:

	x_1	x_2	x_3	x_4	x_5
0	-1	0	1	0	0
0	0	-1	0	1	0
1	$\underline{1}$	-1	0	0	1

The Basis is $\{x_3, x_4, x_5\}$ so we find an extreme point (0,0,0,0,1). The second column is all nonpositive so we find one extreme ray (0,1,0,1,1). This basis is marked as visited. The underlined entry is a pivot element, so we move to a new basis $\{x_3, x_4, x_1\}$. The important point is Simplex method never gets back to this tableau because it always marches towards a optimal solution while our method using DFS gets back to explore other adjacent vertices. So we need to keep the state of this tableau, hence big overhead.

	x_1	x_2	x_3	x_4	x_5
1	0	-1	1	0	$\underline{1}$
0	0	-1	0	1	0
1	1	-1	0	0	-1

Here we find an extreme point (1,0,1,0,0) and an extreme ray (1,1,1,1,0). By entering x_5 in the basis, we have a new basis $\{x_5, x_4, x_1\}$.

	x_1	x_2	x_3	x_4	x_5
1	0	-1	$\underline{1}$	0	1
0	0	-1	0	1	0
0	1	0	-1	0	0

Here we find an extreme point (0,0,0,0,1) and an extreme ray (0,1,0,1,1). By entering x_3, we move to a new basis $\{x_3, x_4, x_1\}$. Since this basis is already marked, we look for another pivot element. In this simple example, there are no other pivot elements, so we backtrack to the previous tableau.

Finally, deleting slack variables from the extreme points and rays provides us with two extreme points: (0,0) and (1,0) and two extreme rays: (0,1) and (1,1) for the initial linear system.

Appendix C. Sessions for Test Programs

C.1 Permutation

```
| ?- terminate(perm(b,f),perm).

+++ Rule-Goal Graph +++
goal(perm(b,f),[perm(b,f),append(b,b,f),append(f,f,b)],
     [rule(1,[]),rule(2,[append(f,f,b),append(b,b,f),perm(b,f)])])
goal(append(b,b,f),[append(b,b,f)],[rule(3,[]),rule(4,[append(b,b,f)])])
goal(append(f,f,b),[append(f,f,b)],[rule(3,[]),rule(4,[append(f,f,b)])])

+++ Predicate Dependency Graph +++
adj(perm(b,f),[perm(b,f),append(b,b,f),append(f,f,b)])
adj(append(b,b,f),[append(b,b,f)])
adj(append(f,f,b),[append(f,f,b)])

+++ Strongly Connected Components +++
[perm(b,f)] [append(f,f,b)] [append(b,b,f)]

+++ SCC([perm(b,f)]): testing termination +++
range of delta(perm(b,f),perm(b,f)): (1.0,inf(+))
range of phi(perm(b,f),perm(b,f)): (1.0,1.0)
range of alpha(perm(b,f),1): (0.5,inf(+))
+++ SCC([perm(b,f)]): shown to terminate +++

+++ SCC([append(f,f,b)]): testing termination +++
range of delta(append(f,f,b),append(f,f,b)): (1.0,inf(+))
range of phi(append(f,f,b),append(f,f,b)): (1.0,1.0)
```

```
range of alpha(append(f,f,b),1): (0.5,inf(+))
+++ SCC([append(f,f,b)]): shown to terminate +++

+++ SCC([append(b,b,f)]): testing termination +++
range of alpha(append(b,b,f),2): (0.0,inf(+))
range of delta(append(b,b,f),append(b,b,f)): (1.0,inf(+))
range of phi(append(b,b,f),append(b,b,f)): (1.0,1.0)
range of alpha(append(b,b,f),1): (0.5,inf(+))
+++ SCC([append(b,b,f)]): shown to terminate +++

| ?- statistics(runtime,T).
T = [9030,310]
```

C.2 Expression Parser

```
| ?- terminate(e(b,f),ent).

+++ Rule-Goal Graph +++
goal(e(b,f),[e(b,f),t(b,f)],[rule(1,[t(b,f),e(b,f)]),rule(2,[t(b,f)])])
goal(t(b,f),[t(b,f),n(b,f)],[rule(3,[n(b,f),t(b,f)]),rule(4,[n(b,f)])])
goal(n(b,f),[e(b,f),z(b)],[rule(5,[e(b,f)]),rule(6,[z(b)])])
goal(z(b),[],[rule(7,[]),rule(8,[]),rule(9,[])])

+++ Predicate Dependency Graph +++
adj(e(b,f),[e(b,f),t(b,f)])
adj(t(b,f),[t(b,f),n(b,f)])
adj(n(b,f),[e(b,f),z(b)])
adj(z(b),[])

+++ Strongly Connected Components +++
[e(b,f),n(b,f),t(b,f)]

+++ SCC([e(b,f),n(b,f),t(b,f)]): testing termination +++
range of alpha(t(b,f),1): (0.5,inf(+))
range of alpha(n(b,f),1): (0.5,0.5)
range of alpha(e(b,f),1): (0.5,0.5)
range of delta(t(b,f),t(b,f)): (1.0,2.0)
range of delta(t(b,f),n(b,f)): (0.0,-0.0)
range of delta(n(b,f),e(b,f)): (1.0,1.0)
range of delta(e(b,f),t(b,f)): (0.0,-0.0)
range of phi(t(b,f),n(b,f)): (0.0,-0.0)
range of delta(e(b,f),e(b,f)): (1.0,2.0)
```

```
range of phi(n(b,f),e(b,f)): (1.0,1.0)
range of phi(t(b,f),t(b,f)): (1.0,1.0)
range of phi(t(b,f),e(b,f)): (1.0,1.0)
range of phi(e(b,f),t(b,f)): (0.0,-0.0)
range of phi(n(b,f),t(b,f)): (1.0,1.0)
range of phi(n(b,f),n(b,f)): (1.0,1.0)
range of phi(e(b,f),n(b,f)): (0.0,-0.0)
range of phi(e(b,f),e(b,f)): (1.0,-0.0)
+++ SCC([e(b,f),n(b,f),t(b,f)]): shown to terminate +++

| ?- statistics(runtime,T).
T = [11299,1319]
```

Appendix D. Test input programs

D.1 Append

```
% the third argument is a concatenation of the first two
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

D.2 Merge

```
% merge U and V into W
merge(U, [], U).
merge([], V, V).
merge([U | Us], [V | Vs],[U | Rs]) :- U > V, merge([V | Vs], Us, Rs).
merge([U | Us],[V | Vs],[U, V | Rs]) :- U = V, merge(Vs, Us, Rs).
merge([U | Us],[V | Vs],[V | Rs]) :- U < V, merge(Vs, [U | Us], Rs).
```

D.3 Mergesort

```
% merge sort
msort([], []).
msort([X], [X]).
msort([X, Y | R],Z) :-
        split([X, Y | R],U,V),
        msort(U, U1),
        msort(V, V1),
        merge(U1, V1, Z).

% split U into V and W evenly
split([], [], []).
split([E | U], [E | V], W) :- split(U, W, V).
```

```
% merge U and V into W
merge(U, [], U).
merge([], V, V).
merge([U | Us], [V | Vs],[U | Rs]) :- U > V, merge([V | Vs], Us, Rs).
merge([U | Us],[V | Vs],[U, V | Rs]) :- U = V, merge(Vs, Us, Rs).
merge([U | Us],[V | Vs],[V | Rs]) :- U < V, merge(Vs, [U | Us], Rs).
```

D.4 Permutation

```
% one argument is a permutation of the other
perm([],[]).
perm(L,[H|T]) :-
        append(V,[H|U],L),
        append(V,U,W),
        perm(W,T).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

D.5 Parser

```
% arithmetic expression  parser
z(L).
e(L, T) :- t(L, T).
e(L, T) :- t(L, ['+' | C]), e(C, T).
t(L, T) :- n(L, T).
t(L, T) :- n(L,['*' | C]), t(C, T).
n([L | T], T) :- z(L).
n(['(' | A], T) :- e(A, [')' | T]).
```

References

[AB91] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9:335–363, 1991.

[AP90] K. R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In *Proceedings of Esprit symposium on computational logic*, pages 150–176, Brussels, November 1990.

[APP+89] F. Afrati, C. Papadimitriou, G. Papageorgiou, A. R. Roussou, Y. Sagiv, and J. D. Ullman. On the convergence of query evaluation. *Journal of Computer and System Sciences*, 38(2):341–359, 1989.

[BJCB87] M. Bruynooghe, G. Janssens, A. Callebaut, and Demoen B. Abstract interpretation: Towards the global optimisation of Prolog programs. In *Proceedings of the 1987 International Symposium on Logic Programming, IEEE Press*, 1987.

[BS89a] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *Eighth ACM Symposium on Principles of Database Systems*, pages 190–199, 1989.

[BS89b] A. Brodsky and Y. Sagiv. On termination of Datalog programs. In *First International Conference on Deductive and Object-Oriented Databases*, pages 95–112, Kyoto, Japan, 1989.

[BS91] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *Tenth ACM Symposium on Principles of Database Systems*, 1991.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[CD93] M. Codish and B. Demoen. Analysing logic programs using "prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*, pages 114–129, Cambridge, Massachusetts, 1993. MIT Press.

[CFW91] G. Cortesi, G. Filè, and W. Winsborough. PROP revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings of Sixth IEEE Symposium on Logic In Computer Science*, pages 322–327, Cambridge, Massachusetts, 1991. IEEE Computer Society Press.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, pages 488–493. MIT Press, New York, 1990.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1981.

[Dev90] P. Devienne. Weighted graphs: a tool for studying the halting problem and time complexity in term rewriting systems and logic programming. *Theoretical Computer Science*, 75(2):157–215, 1990.

[DSD] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. preprint.

[DSVB90] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A practical technique for detecting non-terminating queries for a restricted class of horn clauses, using directed, weighted graphs. In *Proc. 7th Int'l Conf. on Logic Programming*, pages 649–663, Jerusalem, 1990.

[DW88] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.

[Grü67] B. Grünbaum. *Convex Polytopes*. John Wiley and Sons, New York, 1967.

[Hog90] C. J. Hogger. *Essentials of Logic Programming*. Oxford University Press, New York, 1990.

[KLTZ83] M. H. Karwan, V. Lofti, J. Telgen, and S. Zionts. *Redundancy in Mathematical Programming: A State-of-the-Art Survey*. Springer-Verlag, New York, 1983.

[Kow74] R. A. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP'74*, pages 569–574, Amsterdam, 1974. North-Holland.

[Kow79] R. A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–431, 1979.

[Las90a] J.-L. Lassez. Parametric queries, linear constraints and variable elimination. In *Proceedings of DISCO 90, Springer Verlag Lecture Notes in Computer Science*, 1990.

[Las90b] J.-L. Lassez. Querying constraints. In *Ninth ACM Symposium on Principles of Database Systems*, pages 288–298, 1990.

[LCVH93] B. Le Charlier and P. Van Hentenryck. Groundness analysis for prolog: implementation and evaluation of the domain PROP. In *Proceedings of Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1993.

[LHM89] J.-L. Lassez, H. Huynh, and K. McAloon. Simplification and elimination of redundant linear arithmetic constraints. In *North American Conf. on Logic Programming*, pages 37–51, 1989.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 1984.

[Mel81] C. S. Mellish. The automatic generation of mode declaration for logic programs. Technical Report DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1981.

[Mel85] C. S. Mellish. Some global optimizations for a prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[Mel87] C. S. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horword, Chichester, U.K., 1987.

[MR80] T. H. Matheiss and D. S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5:167–185, 1980.

[MS88] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In S. K. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the Fifth International Conference*, pages 733–748, Cambridge, Massachusetts, 1988. MIT Press.

[MS89] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. North American Conf. on Logic Programming, 1989.

[MU87] H. Mannila and E. Ukkonen. Flow analysis of Prolog programs. In *Proceedings of the 1987 International Symposium on Logic Programming*. IEEE Press, 1987.

[MUVG86] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the Nail! system. In *Third Int'l Conf. on Logic Programming*, pages 554–568, 1986.

[Nai83] L. Naish. Automatic generation of control for logic programs. Technical Report 83/6, Dept. of Computer Science, University of Melbourne, Melbourne, Australia, 1983.

[Plü90a] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.

[Plü90b] L. Plümer. Termination proofs for logic programs based on predicate inequalities. In *Proc. 7th Int'l Conf. on Logic Programming*, pages 634–648, Jerusalem, 1990.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Roc70] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, NJ, 1970.

[Sag91] Y. Sagiv. A termination test for logic programs. In *Proceedings of ILPS'91*, pages 160–171, San Diego, 1991. MIT Press.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, New York, 1986.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.

[SU84] Y. Sagiv and J. D. Ullman. Complexity of a top-down capture rule. Technical Report STAN–CS–84–1009, Stanford University, 1984.

[SVG91] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Tenth ACM Symposium on Principles of Database Systems*, 1991.

[Ull85] J. D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-base Systems*. Computer Science Press, Rockville, MD, 1989.

[UV88] J. D. Ullman and M. Y. Vardi. The complexity of ordering subgoals. In *Proceedings of Principles of Database Systems*, pages 74–81, 1988.

[UVG85] J. D. Ullman and A. Van Gelder. Testing applicability of top-down capture rules. Technical Report STAN–CS–85–1046, Dept. of Computer Science, Stanford University, Stanford, CA, April 1985.

[UVG88] J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.

[VG91] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, 3, 1991. Extended abstract appears in Ninth ACM Symposium on Principles of Database Systems, 1990.

[War77] D. H. D. Warren. Implementing prolog - compiling predicate logic programs. Technical Report DAI Research Paper 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1977.