Termination Detection in Logic Programs using Argument Sizes*

(Extended Abstract)

Kirack Sohn and Allen Van Gelder
University of California, Santa Cruz

Abstract

Progress on automated termination detection for logic programs is reported. The prospects for handling a large class of programs completely automatically appear promising, in contrast to the bleak picture for procedural languages. The methods reported are based on term size analysis of procedure arguments. Argument sizes of derivable facts involving an n-ary predicate are viewed as points in the positive orthant of R^n. We describe a method of finding a nonnegative linear combination of bound argument sizes that (if it is found) is guaranteed to decrease during top-down execution of recursive rules. Duality theory of linear programming is used.

This methodology can handle nonlinear recursion, mutual recursion, and cases in which no specific argument is certain to decrease; while it requires rules to have a certain form, this form is attainable by known syntactic transformations. Several programs that could not be shown to terminate by earlier published methods are handled successfully. However, it only analyzes a sufficient condition; a procedure may terminate without our methodology detecting that fact.

*Work partially supported by NSF grants CCR-89-58590 and IRI-89-02287

1 Introduction

Logic programming languages offer an attractive tool to integrate procedural requirements with database access; indeed, subgoals in the logical rules have the same meaning whether they refer to database (EDB) relations, or computed (IDB) relations. The `setof` meta-predicate provides one link, with its ability to convert a relational representation into a specific data structure, such as a list, for more efficient manipulation. Processing of the internal data structure is normally accomplished with recursive rules, and they are normally most efficiently executed top-down. There are large bodies of software that are compiled to execute top-down and achieve satisfactory performance; however, we are aware of no corresponding claim for bottom-up compilation of recursion on structure. Top-down execution will remain an important tool for programming over databases due to its significantly lower constant factor.

The question of whether a set of logic program rules, when executed in a top-down, left-to-right fashion (the Prolog algorithm), is sure to terminate has received considerable study. Of particular interest is the development of an automated method for detecting such termination guarantees for a sizable class of practical logic programs. Essentially, all developed methods seek to identify some function of the bound predicate arguments that is sure to decrease during recursive goal reduction [Nai83, SU84, UVG88, Wal88, APP+89, Plü90, BS89a, BS89b, BS91].

As mentioned in earlier work [VG90], in a deductive database environment, proving termination is not just an academic issue. There exist two approaches to rule evaluation: top-down and bottom-up. Typically, one converges naturally and the other does not on a given set of interdependent rules. Capture rules were introduced by Ullman [Ull85] as a way to plan the evaluation of queries in a "knowledge base." A capture rule is essentially a statement of the form, "if the rules satisfy such-and such conditions, then a good algorithm to evaluate them is such-and-such." In particular, top-down capture rules require a proof of termination to justify use of top-down rule evaluation. An advantage of the capture rule approach is that the system can attempt to choose an order for subgoals and rules that assures termination; not only does this remove the burden from the user, but different orders can be chosen for different bound-free query patterns.

Our approach to proving termination of top-down evaluation is to find some nonnegative linear combination of bound argument sizes that must decrease when a predicate is invoked recursively. It is frequently necessary to know constraints on the argument sizes of a subgoal predicate other than the recursive one, or more precisely, constraints on the *relationship among* the sizes of the subgoal predicate's various arguments. This question was addressed in earlier work [VG90], and we use that capability. Duality theory of linear programming is used.

This methodology can handle nonlinear recursion, mutual recursion, and cases in which no specific argument is certain to decrease; while it requires rules to have a certain form, this form is attainable by known syntactic transformations. Several programs that could not be shown to terminate by earlier published methods are handled successfully. However, it only analyzes a sufficient condition; a procedure may terminate but our methodology will not detect it.

1.1 Related Work

Termination detection for top-down execution amounts to constructing a proof that recursion terminates, which in turn amounts to proving that the goal reduction graph is well-founded. One method is to find a nonnegative function of the goal that is guaranteed to be smaller in each recursive subgoal than its value in the head of the rule. Several approaches have been developed in earlier work.

An early study by Naish considered the case when no bound argument increased and some bound argument, which varied with the rule, decreased [Nai83]. A typical procedure for his method is 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. (See Example 5.1 for a variant). His notion of "$<$" was "proper subterm"; e.g., the tail of a list is less than the whole list. This gives a partial order on terms. 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 introduced the idea of using some notion of term size to define a total order (actually preorder, as there are ties) on terms [UVG88]. They used "length of right spine" as the measure of term size. This corresponds to "length" for lists, but is less natural for binary trees. They gave algorithms for identifying and testing a restricted class of rules, those having the so-called "uniqueness" property. Such programs could be tested in polynomial time for termination. Plümer described some extensions for rules that did not satisfy the "uniqueness" property [Plü90]. Perhaps his main contribution is that certain constraints involving more than two variables can be used. However, he requires an "admissible" property on the rules and has no complexity bounds. Moreover, mutual recursion (two or more interdependent predicates) presents problems for his method.

Brodsky and Sagiv have studied the problem in a function-free setting, where certain partial order constraints and equality constraints on the EDB predicates are supplied externally [BS89a, BS89b]. They give a *necessary and* sufficient condition to guarantee termination on finite EDB relations, strengthening earlier results of Afrati *et al.* [APP+89]. The condition is tested by an exponential time algorithm. They also give a sufficient condition for infinite EDB relations. Although their work is presented in the context of function-free programs, with infinite EDB relations it is possible to simulate functions. We discuss the connection with our methods in Appendix B. Also, these authors have reported recent extensions of their methods [BS91].

Our most important extension to the methods developed earlier is that we can handle constraints among three and more variables (see discussion in Example 3.1). Such constraints arise frequently and naturally in structured logical terms. For example, the atom `p(A•B)`, in which $\bullet$ acts as a binary function symbol, immediately yields a constraint $p_1 = 2 + A + B$ (see Section 2.2).

Our approach is amenable to modularization, as each strongly connected component of predicates (Section 2.3) is analyzed separately. Although this point was not emphasized, it is true of most earlier work, too. In many cases some sort of inter-*argument* constraints are needed to support the termination test. These constraints must usually be established as a separate task from the termination test, which requires inter-*goal* constraints. That is, testing termination for one SCC uses inter-argument constraints from lower SCCs, but does not (necessarily) establish inter-argument constraints for the current SCC. However, inter-argument constraints for the current SCC *can* be established by building upon inter-argument constraints of the lower SCCs, usually without addressing the termination question at all. Brodsky and Sagiv build their termination methods [BS89b] upon a system of inter-argument partial order inference [BS89a]. We build our present system upon linear constraints on the sizes of derivable facts [VG90].

Although it was not aimed at termination detection, we have gained important insights from the work of Lassez [Las90], and Lassez *et al.* [LHM89], on implications among constraints. A related problem that also bridges logic and linear programming is that of linear quantifier elimination, which is treated by Eaves and Rothblum [ER87].

1.2 Outline of the Paper

Section 2 describes basic concepts on term and argument sizes, predicate dependency graphs, and notations used. Section 3 and 4 show how to derive termination conditions in a tractable form using linear programming theory. Section 5 handles an example with multiple bound arguments, and section 6 describes how to handle mutual and nonlinear recursion. Section 7 describes limitations of the methods, and section 8 concludes with a brief discussion. Appendices address various other technical points.

2 Basic Concepts

We assume familiarity with the view of a knowledge base as a relational database, together with a collection of logical rules with a Prolog-like syntax. The collection of all rules for a predicate p is called the *logic procedure for* p. The complete set of rules is referred to as the IDB, but we shall only be concerned with one strongly connected component (SCC) of interdependent predicates at a time. As discussed in Appendix D, our techniques also apply to logic programs with negation.

2.1 Logical Terms

The arguments of predicates are terms, as normally defined in logic: A *term* is a logical variable, a constant, or an uninterpreted function symbol with terms as its arguments. Such terms are usually best interpreted as data structures in the context of logic programming. A *ground term* is one without variables.

In examples, we shall use $\bullet$ (read as "cons") as an infix right associative binary function symbol; the syntax `H • T` conventionally denotes a list whose head is is H and tail is T. We shall use "`[]`" (read as "nil") as a constant, and it conventionally denotes an empty list.

2.2 Term and Argument Sizes

We shall work with a measure that we call *structural term size*, which for *ground terms* (those containing no variables) is defined informally to be the number of edges in the tree that represents the term. More precisely, regarding constants as functions of zero arity, the *structural term size* of a ground term is the sum of the arities of its function symbols. For example, the list `a • b • c • []` has structural term size 6.

For terms containing logical variables, we associate a real variable with each logical variable, and define the structural term size to be the obvious linear polynomial in these real variables. For example, the size of $f(u, v, a)$, where f is a function symbol, a is a constant, and u and v are logical variables, is the *polynomial*, $3 + u + v$. The u and v in the polynomial are *nonnegative real* variables representing the sizes of the *logical* variables u and v in the term. Although this overworks the variable names, which role they play is clear from context.

Similarly, when discussing the atomic formula $p(\vec{x})$, $x^{(i)}$ denotes the logical term that is the i-th argument of p, but when $x^{(i)}$ appears in a mathematical context it is a real variable that represents the *size* of the i-th argument of p in the above formula. For example, if the left side of a rule is $p(f(v_1, g(v_2), v_2), v_1)$, since f and g have arities 3 and 1, respectively, and logical variable v_2 occurs twice in the first argument of p, we obtain the two argument size equations:

$$\begin{aligned} x^{(1)} &= 4 + v_1 + 2v_2 \\ x^{(2)} &= 0 + v_1 \end{aligned}$$

We call these *argument size equations*. Note that these equations will always have nonnegative coefficients and constants when written in this form.

As in linear programming, we shall be concerned frequently with variables that are restricted to be nonnegative. Inequalities can be represented as equations by adding a "slack variable" to the appropriate side, using the convention of nonnegativity.

2.3 Predicate Structure and Dependency Graphs

We construct a digraph with predicates as nodes and arcs $p \rightarrow q$ for every node pair such that q is a subgoal of some rule for p. Intuitively, q supports the derivation, or solution, of p. We identify the strongly connected components (SCCs) of this digraph, and the partial order induced upon them. (We assume that predicates corresponding to database relations never appear on the left side of a rule, and hence are lowest in this partial order.) We shall analyze one SCC at a time for termination. Successful analysis usually requires that information about the predicates of lower SCCs is available.

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*.

2.4 Vector Notation

We shall use a number of conventions in discussing linear systems of equations. Lower-case letters are column vectors; v^{T} denotes the transpose of v, so is a row vector. Upper-case letters are matrices. The vector of argument sizes of the head of a rule is denoted by x; for the recursive subgoal of interest, we use y; for other predicates we use z; for logical variables we use v. Greek letters denote other vector and scalar variables. The relation $\geq$ applied to vectors and matrices is defined to hold if and only if it holds for each component, and thus defines a partial order.

3 Argument Size Constraints for Rules

We assume that preprocessing has arranged that every predicate has the same bound-free adornment, the rules have no positive use of equality, and every subgoal unifies with the heads of all rules of the same predicate name (see Appendix A). Treatment of negative subgoals is described in Appendix D, and involves no new ideas; for now we assume only positive subgoals occur.

We now describe the processing for one SCC of rules. 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_i and a recursive subgoal p_j has been chosen. We obtain linear equations

$$\begin{aligned} x &= a + A\lambda \\ y &= b + B\lambda \qquad x, y, \lambda \geq 0 \\ 0 &= c + C\lambda \end{aligned} \tag{1}$$

where x is the vector of *bound* argument sizes in p_i, y is the vector of *bound* argument sizes in the recursive subgoal p_j, and λ is the vector of other variables that appear in constraints, including those representing the sizes of logical variables and unbound arguments. We shall illustrate with an example where these constraints come from.

First, a, A, b, and B are derived as described in Section 2.2, and are nonnegative. They are derived directly from the rule under consideration.

The situation is different with c and C. These constraints constitute *inter-argument* constraints on p_i, p_j, and subgoals that precede p_j in the rule body. Such constraints are illustrated with `append` in the following example. These are linear *feasibility constraints* on the vectors that correspond to argument sizes of derivable facts for each predicate; they are obtained by methods of Van Gelder [VG90] (or supplied by other external means).

Example 3.1: The recursive rule for a permutation procedure might be

```
perm(P, X•L ) :-
        append( E, X•F, P),
        append( E, F, P1),
        perm(P1, L).
```

Here `append` succeeds when its third argument is the concatenation of the first two. (The base case rule is `perm([],[])`, but plays no part in termination analysis here.)

This example was studied by Plümer [Plü90]. It cannot be shown to terminate (with the first argument bound) 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, and modularity is violated.) 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 $\geq P1$. Now we show how Eq. 1 is set up for this rule.

Let λ be the vector $(P, X, L, E, F, P1)^{\mathrm{T}}$. Assume the first argument is bound and the second is free. We have $x = P$, so a and A are rather boring:

$$\begin{array}{rcl} a & = & \left[\begin{array}{c} 0 \end{array}\right] \\ A & = & \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

Similarly, $y = P1$, so

$$\begin{array}{rcl} b & = & \left[\begin{array}{c} 0 \end{array}\right] \\ B & = & \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

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

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

which must hold for any derivable `append` atom. Applying this to each `append` subgoal, and noting that the size of $\mathtt{X} \bullet \mathtt{F}$ 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. 1, gives

$$\begin{array}{rcl} c & = & \left[\begin{array}{c} 2 \\ 0 \end{array}\right] \\ C & = & \left[\begin{array}{cccccc} -1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 \end{array}\right] \end{array}$$

Discussion of this rule continues in Example 4.1. $\square$

4 Derivation of Termination Conditions

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. That is, our goal here is to find a value for the vectors α_i that guarantee this function decreases.

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. 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. 1 we have

$$\alpha^{\mathrm{T}} x \geq \beta^{\mathrm{T}} y + \delta_{ij} \tag{2}$$

where δ_{ij} is chosen to be 0 or 1 as described in Section 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

$$\forall x y \lambda \ \text{(Eq. 1 implies Eq. 2)} \tag{3}$$

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

$$\text{minimize: } \theta = (\alpha^{\mathrm{T}} x - \beta^{\mathrm{T}} y) \tag{4}$$

subject to Eq. 1. Letting θ^* be the minimum of the objective function, the implication Eq. 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

$$\begin{array}{l} \textbf{maximize:} \\ \quad (u^{\mathrm{T}}, v^{\mathrm{T}}, w^{\mathrm{T}}) \left[\begin{array}{c} a \\ b \\ c \end{array}\right] \\ \textbf{subject to:} \\ \quad (u^{\mathrm{T}}, v^{\mathrm{T}}, w^{\mathrm{T}}) \left[\begin{array}{ccc} I & 0 & -A \\ 0 & I & -B \\ 0 & 0 & -C \end{array}\right] \leq (\alpha^{\mathrm{T}}, -\beta^{\mathrm{T}}, 0) \end{array} \tag{5}$$

By duality theory this has the same optimum, θ^*, as the primal, and so Eq. 3 is true if and only if the dual constraints of Eq. 5 remain satisfiable with the addition of

$$(u^{\mathrm{T}}, v^{\mathrm{T}}, w^{\mathrm{T}}) \left[\begin{array}{c} a \\ b \\ c \end{array}\right] \geq \delta_{ij} \tag{6}$$

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 [Las90]. We incorporate the final constraints,

$$\alpha \geq 0 \qquad \beta \geq 0 \tag{7}$$

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

$$\left[\begin{array}{ccccc} -I & 0 & 0 & I & 0 \\ 0 & -I & 0 & 0 & -I \\ A^{\mathrm{T}} & B^{\mathrm{T}} & C^{\mathrm{T}} & 0 & 0 \\ a^{\mathrm{T}} & b^{\mathrm{T}} & c^{\mathrm{T}} & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{array}\right] \left[\begin{array}{c} u \\ v \\ w \\ \alpha \\ \beta \end{array}\right] \geq \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ \delta_{ij} \\ 0 \\ 0 \end{array}\right] \tag{8}$$

This set of constraints is very amenable to reduction by Fourier-Motzkin elimination [Sch86, LHM89, Las90]. 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.

In fact, from the origin of a, A, b, and B in Eq. 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. 8, giving

$$\begin{bmatrix} C^{\mathrm{T}} & A^{\mathrm{T}} & -B^{\mathrm{T}} \\ c^{\mathrm{T}} & a^{\mathrm{T}} & -b^{\mathrm{T}} \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} w \\ \alpha \\ \beta \end{bmatrix} \geq \begin{bmatrix} 0 \\ \delta_{ij} \\ 0 \\ 0 \end{bmatrix} \tag{9}$$

To claim a theoretical polynomial time bound, we stop with Eq. 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 4.1: Continuing with Example 3.1, the constraints corresponding to Eq. 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 $\alpha = 1/2$. □

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 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.

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, Plü90]. 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 5.1: The following procedure merges two input lists [VG90]. We assume the first and second arguments are bound and the third is free.

```
merge([], Ys, Ys).
merge(Xs, [], Xs).
merge(X•Xs, Y•Ys, X•Zs) :-
        X ≤ Y, merge(Y•Ys, Xs, Zs).
merge(X•Xs, Y•Ys, Y•Zs) :-
        Y ≤ X, merge(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. 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. 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. □

6 Mutual and Nonlinear 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.

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 (see Appendix A).

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. 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 6.1 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*. We discuss how to find such solutions in Appendix C.

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 6.1: 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`.

This example was studied by Plümer [Plü90], 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`.

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. 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 [VG90], 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

$$\begin{array}{rcl} c & = & \left[\begin{array}{c} 4 \end{array}\right] \\ C & = & \left[\begin{array}{cccc} -1 & 0 & 1 & 1 \end{array}\right] \end{array}$$

We also have

$$\begin{array}{rcl} a & = & \left[\begin{array}{c} 0 \end{array}\right] \\ A & = & \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \end{array}\right] \\ b & = & \left[\begin{array}{c} 0 \end{array}\right] \\ B & = & \left[\begin{array}{cccc} 0 & 0 & 1 & 0 \end{array}\right] \end{array}$$

Let us identify e with α, t with β, and n with γ. Then the constraints corresponding to Eq. 9 are (note that $(A^{\mathrm{T}} - B^{\mathrm{T}})$ and $(a^{\mathrm{T}} - b^{\mathrm{T}})$ form the second column):

$$\left[\begin{array}{rr} -1 & 1 \\ 0 & 0 \\ 1 & -1 \\ 1 & 0 \\ 4 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} w_1 \\ \alpha \end{array}\right] \geq \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \delta_{ee} \\ 0 \end{array}\right]$$

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

$$\left[\begin{array}{rr} -1 & 1 \\ 0 & 0 \\ 0 & 2 \\ 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \alpha \\ \gamma \end{array}\right] \geq \left[\begin{array}{c} 0 \\ 0 \\ \delta_{ne} \\ 0 \\ 0 \end{array}\right]$$

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. □

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 in Appendix A 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".

Also, it is possible that bound arguments really do shrink during recursion, but the inter-argument 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.

8 Conclusion

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. Most of the implementation for termination detection is already done in Prolog. The required imported feasibilty constraints are taken as input, but are not automated; there is still much work to be done in this area.

Acknowledgements

We thank J.-L. Lassez, C. H. Papadimitriou and Y. Sagiv for helpful discussions.

Appendix A Syntactic Transformations

Rules are formulated by people in a way that aids their own understanding. Frequently, mutual recursion is used when it is not strictly necessary.[1] Also, special cases of relations are often invoked as subgoals that do not unify with the heads of all rules with the same predicate name. The true computational structure of such rules can often be exposed by two sets of syntactic transformations, called *predicate splitting* and *safe unfolding*. Before describing them, we note that any rule with positive equality has a logical equivalent without positive equality; e.g., $r(Z)$:- $U{=}f(Z)\,p(U)$ is equivalent to $r(Z)$:- $p(f(Z))$. We assume in this discussion that positive equality has been eliminated. Because this material is somewhat tangential, we illustrate the main ideas with examples and eschew a formal treatment.

Predicate splitting was mentioned (but not named) by Ullman and Van Gelder [UVG88]. When term structure appears in the argument of a subgoal $p(\vec{t})$ (i.e., the arguments $\vec{t}$ are not simply variables) that subgoal may not unify with the heads of certain rules for p. As a simple example, let p have two rules,

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

and let r have a rule, `r(Z) :- p(f(Z))`. Then the subgoal `p(f(Z))` does not unify with the first rule for p.

The behavior of rules with which this subgoal cannot unify may obscure the overall termination behavior of the program, as shown by a merge sort example in the cited work. To alleviate this problem, the relation p can be split into three relations, p, p_1 and p_2. The heads of rules with which $p(\vec{t})$ does not unify are renamed to p_1; those with which $p(\vec{t})$ does unify are renamed to p_2. Rules for p are added: $p(\vec{X})$:- $p_1(\vec{X})$ and $p(\vec{X})$:- $p_2(\vec{X})$. Finally, other p subgoals in the program are specialized to p_1 or p_2 if possible. The above example transforms into:

```
p(X)  :- p1(X).       p(X) :- p2(X).
p1(a).
p2(X) :- q(X, Y), p(Y).
r(Z)  :- p2(f(Z)).
```

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 $p(\vec{X_i})$:- $B_i(\vec{X_i})$, then one rule in which p is a subgoal, say $q(\vec{Y})$:- $\ldots, p(\vec{Z}), \ldots$, can be replaced by k rules,

$$q(\vec{Y}\theta_i) \text{ :- } \ldots, B_i(\vec{X}\theta_i), \ldots$$

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 p, then to p_1, and gives:

```
p(X)  :- p1(X).       p(X) :- p2(X).
p1(a).
p2(X) :- q(X, a).
p2(X) :- q(X, Y), p2(Y).
r(Z)  :- p2(f(Z)).
```

If p and p_1 are 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.

Observe that predicate splitting may introduce mutual recursion, while safe unfolding may introduce additional term structure into arguments as a result of substitutions ($q(X, Y)$ became $q(X, a)$ above). Thus it is not clear whether repeatedly using both of these heuristics together is certain to terminate. Until this question is settled, an automated application should run alternate phases of safe unfolding and predicate splitting, and halt after a fixed number of phases, say 3 of each.

[1] For example, the behavior for e and t in Example 6.1 can be defined without introducing the third predicate n, but the resulting rules would be more complicated and difficult to understand.

Example A.1: The following (unmotivated) rules exhibit an apparent mutual recursion in which the argument size does not change. The argument of p is bound, and e is an EDB predicate. Observe that no predicate splitting is applicable. Our algorithm does not detect termination of these rules in their present form. We shall see that a sequence of automatic syntactic transformations puts the rules into a form in which termination is easily detected.

```
p(g(X)) :- e(X).
p(g(X)) :- q(f(X)).
q(Y)    :- p(Y).
q(f(Z)) :- p(Z), q(Z).
```

During safe unfolding each p subgoal is replaced by the body of each rule for p with an appropriate substitution.

```
   p(g(X)) :- e(X).
   p(g(X)) :- q(f(X)).
   q(g(X)) :- e(X).
   q(g(X)) :- q(f(X)).
q(f(g(X))) :- e(X), q(g(X)).
q(f(g(X))) :- q(f(X)), q(g(X)).
```

The SCC that used to consist of p and q now contains only q. But now the `q(f(X))` subgoal fails to unify with the heads of some rules for q, so predicate splitting on q is now applicable.

```
    p(g(X)) :- e(X).
    p(g(X)) :- q2(f(X)).
       q(U) :- q1(U).
       q(U) :- q2(U).
   q1(g(X)) :- e(X).
   q1(g(X)) :- q2(f(X)).
q2(f(g(X))) :- e(X), q1(g(X)).
q2(f(g(X))) :- q2(f(X)), q1(g(X)).
```

Finally, another phase of safe unfolding eliminates the dependence of q_2 on q_1. Assuming q was not called except by p, intermediate predicates q and q_1 can be discarded.

```
    p(g(X)) :- e(X).
    p(g(X)) :- q2(f(X)).
q2(f(g(X))) :- e(X), e(X).
q2(f(g(X))) :- e(X), q2(f(X)).
q2(f(g(X))) :- q2(f(X)), e(X).
q2(f(g(X))) :- q2(f(X)), q2(f(X)).
```

The fact that p is not genuinely recursive has been exposed. Termination of this system can be detected by a number of methods. (Considerable further simplifications are possible by subsumption, assuming a "pure" language without side-effects.) □

Appendix B Relation to Methods of Brodsky and Sagiv

A partial order 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 are the backbone of the *argument mapping* techniques developed by Brodsky and Sagiv [BS89a, BS89b]. They are easily representable in our context by argument size equations, such as

$$\begin{aligned} x^{(1)} &= 1+\lambda+\mu \\ x^{(2)} &= 0+\lambda \end{aligned}$$

which states that $x^{(1)} > x^{(2)}$. Variables λ and μ must not appear elsewhere. However, argument mapping techniques process constraints differently from our methods, so the mere fact of representation does not imply greater power. Indeed, argument mapping analysis can detect termination due to certain unification issues, such as "occurs check", which escape our notice.

Going the other direction, all unification can be pushed into explicit equality subgoals, which can then be regarded as (infinite) EDB predicates yielding partial order constraints. For example, `m(X, X•R)` is rewritten as `Y = X•R, m(X, Y)`. Then `Y = X•R` is treated as an EDB predicate $e(Y, X, R)$ with the partial order constraints $e_1 > e_2$ and $e_1 > e_3$. Evidently some information is lost in this transition. This translation was found to be sufficient to handle Example 5.1 and Example 6.1, but not Example 3.1. The latter example *can* be handled by argument mapping together with our constraint inference technique. Brodsky and Sagiv have reported new results recently [BS91].

Appendix C Negative Weights

Here we sketch the method of finding feasible solutions to systems of constraints like those in Section 6, but without imposing an arbitrary constraint that all $\delta_{ij} \geq 0$. Intuitvely, 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 6.

Appendix D Negative Subgoals

Negative subgoals preceding the recursive subgoal of interest are discarded, as they neither help nor hinder our analysis. The reason is that negative subgoals do not produce variable bindings. Thus, if the positive version of a negative subgoal cannot succeed, it can be discarded (which we do). On the other hand, if the positive version succeeds, the negative one fails, and the recursive subgoal is not reached.

When the recursive subgoal of interest is negative, it is treated as though it were positive. Normally negative subgoals are only attempted with all arguments bound.

References

[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.

[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.

[ER87] B. C. Eaves and U. G. Rothblum. Elimination of quantifiers of linear variables and corresponding transfer principles. Technical Report Operations Research, Stanford University, 1987.

[Las90] J.-L. Lassez. Querying constraints. In *Ninth ACM Symposium on Principles of Database Systems*, pages 288–298, 1990.

[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.

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

[Plü90] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1990.

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

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

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

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

[VG90] A. Van Gelder. Deriving constraints among argument sizes in logic programs. *Annals of Mathematics and Artificial Intelligence*, (to appear), 1990. Available as UCSC-CRL-89-41. Extended abstract appears in Ninth

ACM Symposium on Principles of Database Systems, 1990.

[Wal88] C. Walther. *Automated Termination Proofs*. PhD thesis, University of Karlsruhe, 1988.