Deriving Constraints Among Argument Sizes in Logic Programs*

Allen Van Gelder
University of California, Santa Cruz

Abstract

In a logic program the feasible argument sizes of derivable facts involving an n-ary predicate are viewed as a set of points in the positive orthant of R^n. We investigate a method of deriving constraints on the feasible set in the form of a polyhedral convex set in the positive orthant, which we call a *polycone*. Faces of this polycone represent inequalities proven to hold among the argument sizes. These inequalities are often useful for selecting an evaluation method that is guaranteed to terminate for a given logic procedure. The methods may be applicable to other languages in which the sizes of data structures can be determined syntactically.

For any atomic formula (atom, for short) in a rule, we show how to express the vector of its argument sizes as a system of linear equations and inequalities involving sizes of the logical variables that occur in it. This system defines a polycone, which represents the set of *feasible* argument size vectors. Transformations combine polycones for all atoms in one rule to give the feasible polycone for the entire rule.

We introduce a *generalized Tucker representation* for systems of linear equations. We prove that every polycone has a unique *normal form* in this representation, and give an algorithm to produce it. This in turn gives a decision procedure for the question of whether two set of linear equations define the same polycone.

When a predicate has several rules, the union of the individual rule's polycones gives the set of feasible argument size vectors for the predicate. Because this set is not necessarily convex, we instead operate with the smallest enclosing polycone, which is the closure of the convex hull of the union. Retaining convexity is one of the key features of our technique.

Recursion is handled by finding a polycone that is a fixpoint of a transformation that is derived from both the recursive and nonrecursive rules. Some methods for finding a fixpoint are presented, but there are many unresolved problems in this area.

*An extended abstract of this paper appeared in *Ninth ACM Symposium on Principles of Database Systems*, March 1990.

1 Introduction and Basic Concepts

Top-down capture rules were introduced by Ullman [14] and studied by Sagiv and Ullman [12], Ullman and Van Gelder [15], Afrati *et al.* [1], and elsewhere, as a way to plan the evaluation of queries in a "knowledge base." Capture rules require a proof of termination to justify use of top-down rule evaluation. Top-down rule evaluation is similar to the evaluation method of Prolog, except that the system decides on the order for subgoals and rules. It is often called "back-chaining" in the artificial intelligence community.

In a knowledge base 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. Even if the less appropriate method can be made to converge, it is likely to be very inefficient. Rules that define a predicate by "recursion on structure" should usually be evaluated top-down, while those that define their predicate by "inductive closure" (e.g., transitive closure) usually require a bottom-up component in their evaluation. Mixtures of these elementary types are bound to occur in complex systems.

A useful approach to proving termination of top-down evaluation methods is to find some convex linear combination of argument sizes that must decrease when a predicate is invoked recursively. To prove that a bottom-up method converges, find an upper bound on all argument sizes that cannot be exceeded when rules are applied in the "forward" direction. However, to prove that the desired function of the goal predicate's argument sizes decreases (for top-down) or remains bounded (for bottom-up), it is frequently necessary to know constraints on the argument sizes of a subgoal predicate, or more precisely constraints on the relationship among the sizes of the subgoal predicate's various arguments. This paper is concerned with deriving such constraints.

Our model of a knowledge base, as in [14, 15, 7, 16], is a relational database, together with a collection of logical rules in the form of Horn clauses

$$p(\vec{x}) \leftarrow q_1(\vec{z_1}),\ \ldots,\ q_k(\vec{z_k})$$

The set of rules is referred to as a logic program. As will be seen, our techniques also apply to logic programs with stratified negation.

Here $\vec{x} \equiv (x_1, \ldots, x_n)$ is a vector of p's arguments; each argument x_i is a *term* as normally defined in logic. Such a rule is read "$p(\vec{x})$ is true if there exist assignments to the logical variables appearing in $\vec{z_1}, \ldots, \vec{z_k}$ but not in $\vec{x}$ such that $q_1(\vec{z_1}), \ldots, q_k\vec{z_k})$ are all true." We call this a *rule for p*, since predicate p appears on its left side, which is called the *head* of the rule. The atomic formulas (atoms, for short) $q_1(\vec{z_1}), \ldots, q_k\vec{z_k})$ on the right of the rule are called its *subgoals*. The collection of all rules for p is called the *logic procedure for p*.

A "top-down" interpretation of the above rule is: To find a tuple that satisfies p, find "joinable" tuples that satisfy $q_1, \ldots, q_k$. The "bottom-up" interpretation is: Given tuples satisfying $q_1, \ldots, q_k$, infer a tuple satisfying p. One or more of the q_j may actually be p, making the rule immediately recursive, and recursion may also occur by p becoming a subgoal of itself through a chain of several rules.

A frequently applicable way to ensure that a top-down approach terminates in the presence of recursion is to show that the bound arguments grow smaller in some sense as the recursion progresses. Top-down capture rules are discussed extensively in Ullman [14], Sagiv and Ullman [12], Ullman and Van Gelder [15].

This paper develops methods to find, through syntactic examination of a logic program, a set of linear inequalities for each predicate p such that if $p(x_1, \ldots, x_n)$ is derivable by the program, then the term sizes of $(x_1, \ldots, x_n)$ (viewed as a vector) must satisfy all of these inequalities. Different predicates have different sets of inequalities that their arguments must satisfy. We extend the methods developed in [15] in two important ways. First, our approach is amenable to modularization, as each strongly connected component of predicates (Section 1.6) is analyzed separately. Second, our methods can handle constraints among three and more variables (see discussion in Example 5.1).

1.1 Related Work

We visualize application of the present work in the automated construction of termination proofs based on analysis of argument sizes. The problem of proof construction has been studied in its own right by Naish [8], Ullman and Van Gelder [15], Walther [17], Afrati *et al.* [1], Plümer [10], and Brodsky and Sagiv [2].

A central problem in our technique is that of deciding polycone equivalence. While this problem does not appear to have been studied in this precise form, the closely related problem of eliminating redundant constraints in linear programs has been studied extensively; Karwan *et al.* [5] survey the field. Implicit equalities are inequality constraints that can in fact only be satisfied by equality. Such constraints also represent a form of redundancy. Identification of implicit inequalities has been studied by Telgen [13], Freund *et al.* [4], and very recently by Lassez and McAloon [6]. A less closely related problem that also bridges logic and linear programming is that of linear quantifier elimination, which is treated by Eaves and Rothblum [3].

1.2 Outline of the Paper

In the remainder of Section 1 we show how to associate a set of linear inequalities with each rule in a program, and describe how the program's dependency graph determines which ones must be treated as a group. Each rule defines a polycone (Definition 1.1).

Section 2 describes several useful operations on polycones, in particular the *closure of the convex hull of union* CHU. This key operation allows us to combine feasible regions of several rules, getting a polycone that encloses their union. The motivation for using convex hulls in the this work is that convex sets can be described conjunctively, that is, as a set of conditions that must *all* hold. Each rule for a given predicate in a logic program can be satisfied only by arguments whose sizes lie in a certain convex set (in fact, a polycone). However, the possible sizes of arguments for which *one of several* rules is satisfied form a *union* of convex sets, which is not necessarily convex. The information that a point is in such a union needs to be represented and reasoned with *disjunctively*. The extra complexity is unmanageable in practice. However, the information that a point is in the closure of the convex hull of the union can still be represented conjunctively, and intuitively seems to be the most specific conjunctive statement that implies that the point is in the union. It is our belief that in practice the constraints given by CHUs are sharp enough to be useful in evaluating convergence of top-down capture rules. A rich theory has been developed for convex sets, and has been collected by Rockafellar [11]. Rigor supporting our rather informal exposition of polycones may be found there.

Section 3 defines two transformations on polycones, called Ψ and T, that are associated with sets of interdependent recursive rules. Section 4 describes how a polycone that bounds the feasible region of argument sizes can be defined as a fixpoint of the transformation T.

Remaining sections explore the problems of finding a such a fixpoint. Section 5 describes a method to verify a conjectured fixpoint that may be fast, but is not certain to work. Section 6 introduces a *normal form* for polycones that makes the question of polycone equivalence decidable. Section 7 offers an heuristic for guessing a fixpoint. Section 8 discusses future directions.

1.3 Logical Terms

The arguments of predicates are terms, as normally defined in logic: A *term* is 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 the infix operator “ · ” (read as “cons”) as a binary function symbol to construct lists, a class of terms that occur frequently in practice; we shall use “□” (read as “nil”) as the constant that represents the empty list. Thus $a \cdot R$ represents the list whose head (first element, *car*) is a and whose tail (remaining elements, *cdr*) is R. Since both a and R may have structure, “ · ” is effectively a constructor for nodes of a binary tree, but in logic programming it is normally used to build lists, as other function symbols are available for different structures. Unlike several popular versions of Prolog syntax, we do not enclose lists in square brackets.

1.4 Term and Argument Sizes

Several measures of term size are possible. We shall work with one 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 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: the constant for this polynomial is the sum of the arities of the function symbols in the term, and the coefficient of each real variable is the number of occurrences in the term of its associated logical variable. 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 each argument term x_i, let Q_i be the polynomial that is its structural term size. Then we have the obvious linear equation

$$x_i = Q_i$$

involving the real variable x_i and real variables corresponding to logical variables in the term. We call these *argument size equations*. 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:

$$\begin{aligned} x_1 &= 4 + v_1 + 2v_2 \\ x_2 &= 0 + v_1 \end{aligned}$$

Note that these equations will always have nonnegative coefficients and constants when written in this form. Since two logical variables can appear in one term only if there is at least a binary function symbol to connect them, we can see that argument size equations satisfy these further constraints:

1. If the additive constant is 0, there is at most one positive coefficient of a variable, which must be 1. (I.e., the equation is simply $x_i = v_j$.)

2. If there is more than one positive coefficient of a variable, then the additive constant is at least as large as the sum of the coefficients of the variables.

1.5 Inequalities and Slack Variables

As in linear programming, we shall be concerned almost exclusively 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. Conversely, an equation can be converted to an inequality by "projecting out" a nonnegative variable. In general, the set of points satisfying an equation or set of equations will be restricted to have all nonnegative components, i.e., the set will lie in the *positive orthant* of the appropriate vector space.

1.6 Predicate Structure of Logic Programs

For simplicity, let us assume that each head of a rule in our logic program can unify with *every* occurrence of the same predicate in subgoals of rules. (This involves no real loss of generality, as a program can be effectively transformed to have this property.) Now 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 the SCCs according to their partial order, from lowest to highest. Thus, at the time we are deriving constraints on the argument sizes of a certain SCC, we should already have constraints derived for all predicates that are outside this SCC, but appear as subgoals in the rules being analyzed. Specifically, if p is in the SCC being analyzed, and q appears as a subgoal of a rule for p, then either q is in the same SCC, or is in one already analyzed. If the latter, then constraints on the arguments of q have already been derived, and are available for use in the current analysis.

Recursion in rules can occur in more and less complex forms. A *recursive subgoal* is one whose predicate is in the same SCC as the head of the rule. If each rule in an SCC has at most one recursive subgoal, then the recursion in this SCC is said to be *linear*. If recursion is linear and in addition there is just one predicate in the SCC, we say the recursion is *simple*.

For the main presentation, we shall assume that the SCC contains only *simple recursion*, so a typical recursive rule is of the form:

$$p(x_1, \ldots, x_n) \leftarrow p(y_1, \ldots, y_n),\ r(z_1, \ldots, z_m)$$

where q (if present) is in a lower SCC than p. Linear recursion *does* involve a loss of generality,[1] but provides a clearer environment for the exposition of the main ideas, and covers many common cases. In Appendix A we outline the changes needed to accomodate general recursion.

[1] For example, divide-and-conquer procedures typically have two recursive subgoals in one rule.

1.7 Notation and Definitions

We shall use a number of conventions in discussing linear systems of equations. Lower-case letters are vectors; upper-case are matrices. The vector of argument sizes of the head of a rule is denoted by x; for the recursive occurrence of the same predicate on the right, we use y; for non-recursive predicates we use z; for logical variables we use v. Greek letters denote other vector variables.

The variables appearing on the left side of an equation are considered to be the *dependent* variables; those on the right are the *independent* variables. All independent variables are implicitly restricted to be nonnegative. 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.

Linear systems of equations will be represented in a special matrix notation that we now illustrate. A set of equations in the form $x = a + Av$, where a is a vector constant, and A is a matrix, is called a *Tucker representation* [11]. These equations are shown as:

$$\begin{array}{c} \begin{array}{ccc} (x) & & (v) \end{array} \\ \left[\begin{array}{c||c|c} I & a & A \end{array}\right] \end{array}$$

or if x and v are understood:

$$\left[\begin{array}{c||c|c} I & a & A \end{array}\right]$$

The usefulness of this notation is apparent when we look at an example in which larger matrices (or vectors) are composed from smaller ones. To represent both $x = a + Av$ and $z = b + Bv$ as a set of x-points generated by z's and v's, we write:

$$\begin{array}{c} \begin{array}{cccc} (x) & & (v) & (z) \end{array} \\ \left[\begin{array}{c||c|cc} I & a & A & 0 \\ 0 & b & B & -I \end{array}\right] \end{array} \tag{1.1}$$

or, since x is understood and the name of v is not important:

$$\begin{array}{c} \begin{array}{cccc} & & & (z) \end{array} \\ \left[\begin{array}{c||c|cc} I & a & A & 0 \\ 0 & b & B & -I \end{array}\right] \end{array} \tag{1.2}$$

In general, the vertical double line marks the location of the equal signs. Columns to the left of it represent the dependent variables (usually x). The one column between the double line and single line is the constants column. Finally, columns to the right of the single line represent independent variables; we shall use the term *right-hand columns* to refer to them. In this paper we shall rarely need to talk about columns to the left of the double line, so we shall denote the constants column as column 0 and the right-hand columns as columns 1, 2, etc. When speaking of right-hand columns, we shall mean "column" in a generalized sense: this "column" will often contain a submatrix. (However, column 0, the constants column will always be a single column.) Thus numbering should be interpreted relative to the diagram in question. Similarly we shall refer to generalized "rows," starting with 1. In Eq. 1.2, for example, we say that A is in row 1, column 1, and that b is in row 2, column 0.

Frequently, we shall use these matrices to represent sets of points in the following sense (where I_n is the $n \times n$ identity, and A_1 and A_2 have m columns):

$$\left[\begin{array}{c||c|c} I_n & a_1 & A_1 \\ 0 & a_2 & A_2 \end{array}\right] \tag{1.3}$$

represents the set of points $x \in R^n$ such that there exists λ ($\in R^m$) ≥ 0 that satisfies $0 = a_2 + A_2\lambda$ (the *lower rows*) and for which $x = a_1 + A_1\lambda$ (the *upper rows*). We say that each λ satisfying the lower rows *generates* the x that results from substituting that λ into the upper rows. Geometrically, we are projecting a set in R^{n+m} with coordinates (x, λ) into R^n, retaining the x coordinates.

Definition 1.1: A *polyhedral convex set* in R^n is the (possibly unbounded) set of points constituting the intersection of a given finite set of closed half-planes [11].

A *polycone* in R^n is a polyhedral convex set that lies entirely within the positive orthant. Equivalently, it is the set of points that satisfy a given set of linear "$\geq$" inequalities, among them $x_i \geq 0$ for $1 \leq i \leq n$.

A matrix in the form of Eq. 1.3, and the set of equations it represents, are called a *generalized Tucker representation* of a polyhedral convex set in R^n. □

It is easy to show, by introducing slack variables and using Gaussian elimination, that any polycone can be expressed in the form of Eq. 1.3. (If $Ax \geq a$ includes the constraints $x \geq 0$, then A has full rank.) Similarly, the solution set of Eq. 1.3, projected on the dependent variables, must be a polycone. The understanding that independent variables are restricted to nonnegative values is central to this representation of polycones, and will not be mentioned further. We observe that a generalized Tucker representation reduces to a Tucker representation when A_2 has no rows in Eq. 1.3.

The empty set $\emptyset$ is technically a polycone. We generally ignore this case in our presentation.

Example 1.1: When invoking the following logic procedure, the first argument should be a list to be reversed, the second argument should be □. The procedure instantiates the third argument to the reversed list. (The second argument functions as a place-holder for the partially reversed list; read "D" as "Done.") The Prolog convention of using capital letters for logical variables has been employed.

$$rev(\square,\, R,\, R).$$
$$rev(E \cdot L,\, D,\, R) \leftarrow rev(L,\, E \cdot D,\, R).$$

The matrices for these two rules are shown below.

$$\begin{array}{ccc} (x) & & (R) \\ \left[\begin{array}{c} 100 \\ 010 \\ 001 \end{array}\right\Vert & \begin{array}{c} 0 \\ 0 \\ 0 \end{array} \Big| & \left.\begin{array}{c} 0 \\ 1 \\ 1 \end{array}\right] \end{array} \qquad \left[\begin{array}{cc||c|cccc} (x) & (y) & & (E) & (L) & (D) & (R) \\ 100 & 000 & 2 & 1 & 1 & 0 & 0 \\ 010 & 000 & 0 & 0 & 0 & 1 & 0 \\ 001 & 000 & 0 & 0 & 0 & 0 & 1 \\ 000 & 100 & 0 & 0 & 1 & 0 & 0 \\ 000 & 010 & 2 & 1 & 0 & 1 & 0 \\ 000 & 001 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{1.4}$$

The left matrix applies to the first, nonrecursive, rule and should be fairly self-explanatory. The right matrix applies to the second rule. The coefficients for sizes of E, L, D, R are in columns 1 through 4, respectively. (Recall the column numbering convention.) The 2's that appear in the constants column arise from " $\cdot$ " being a binary function symbol. Finally, y (a 3-vector) represents argument sizes in the recursive occurrence of *rev* on the right side of the rule.

We can interpret the left matrix in Eq. 1.4 as the constraints $x_1 = 0$ and $x_2 = x_3 \geq 0$. The right matrix can be reduced to the form

$$x = y + \left[\begin{array}{r} 1 \\ -1 \\ 0 \end{array}\right] \lambda + \left[\begin{array}{r} 2 \\ -2 \\ 0 \end{array}\right]$$

which can be thought of as a transformation of point y into a ray. The point represents the (vector of) argument sizes of the subgoal, and the ray represents the possible argument sizes that result on the left, when the second rule above is used. □

When we regard a matrix as representing a polycone, the independent variables are essentially anonymous. In particular, a column that contains coefficients of an independent variable (i.e., a *right* column) can be rescaled by a positive quantity without changing what polycone is represented. Furthermore, a lower row (i.e., involving only independent variables) can be rescaled by any nonzero quantity and/or added to any other row (upper or lower) leaving the polycone invariant. (Such an operation with an upper row would also be valid, but is avoided, as it is preferable for our purposes to keep the identity matrix on the left.)

2 Operations on Polycones

We now examine several operations on polycones that prove useful in combining several polycones, each representing constraints on argument sizes of a single rule, into a polycone that represents constraints on a predicate that hold throughout the logic program.

2.1 Closure of Convex Hull of Union

The first operation constructs the closure of the convex hull of the union of two polycones. Recall that a *convex combination* of a finite set of points $x_1, \ldots, x_k$ is a point $(\lambda_1 x_1 + \cdots + \lambda_k x_k)$, where $\lambda_i \geq 0$ and $\lambda_1 + \cdots + \lambda_k = 1$. A *nontrivial* convex combination is one in which at least two λ components are nonzero.

Definition 2.1: The *closure of the convex hull of the union* (CHU) of two sets of points $S_1, S_2 \subseteq R^n$ is defined as the closure of the set of points that are convex combinations of two points in the union of the sets; it is denoted by $S_1 \mathbin{\bar{\cup}} S_2$. The symbol $\bar{\cup}$ is intended to suggest the combination of "union" and "filling in." □

That is, if S_1 and S_2 are convex (the only case we are concerned with) and $x_1 \in S_1$, $x_2 \in S_2$, $\alpha_1 + \alpha_2 = 1$, and α_1 and α_2 are both nonnegative, then $\alpha_1 x_1 + \alpha_2 x_2$ is in the convex hull of $S_1 \cup S_2$. But moreover, the boundary points of the set so formed are also included in the closure of the convex hull. Because we allow unbounded polycones, it is necessary to take the closure to guarantee that the CHU of two (or any finite number of) polycones is itself a polycone.

Example 2.1: Consider $\left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] \mathbin{\bar{\cup}} \left[\begin{smallmatrix} \lambda \\ 0 \end{smallmatrix}\right]$, i.e., the CHU of a point off the x_1-axis and a ray along the x_1-axis. The convex hull of the union is $\{x_1 \geq 0 \wedge 0 \leq x_2 < 1\} \cup \{(0,1)\}$, which excludes the boundary points $\{x_1 > 0 \wedge x_2 = 1\}$, and is not a polycone. Taking the closure includes these boundary points, and yields the CHU, which is described by $\{x_1 \geq 0 \wedge 0 \leq x_2 \leq 1\}$, and clearly *is* a polycone. □

The form of Eq. 1.3 is very convenient for forming the CHU of two polycones, as shown by the following basic theorem:

Theorem 2.1: Let two polycones S_1 and S_2 be specified by

$$\begin{array}{c} \\ \left[\begin{array}{c||c|c} I & b_1 & B_1 \\ 0 & b_2 & B_2 \end{array}\right] \end{array}\!\!\!\!\begin{array}{c} \end{array} \quad \text{with columns } \quad \tag{2.1}$$

respectively. Then $S_1 \mathbin{\bar{\cup}} S_2$, the closure of the convex hull of their union, is the polycone S_3 given by:

$$\begin{array}{ccccccc} & & & (\alpha_1) & (\lambda_1) & (\alpha_2) & (\mu_2) \\ \left[I \right. & \| & 0 & b_1 & B_1 & c_1 & \left. C_1 \right] \\ 0 & \| & 0 & b_2 & B_2 & 0 & 0 \\ 0 & \| & 0 & 0 & 0 & c_2 & C_2 \\ 0 & \| & -1 & 1 & 0 & 1 & 0 \end{array} \tag{2.2}$$

where α_1 and α_2 are scalar variables. (The variable names over the columns are given to help in the proof, but are otherwise immaterial.)

Proof: Let x_3 be any convex combination of points $x_1 \in S_1$ and $x_2 \in S_2$; that is, $x_3 = \alpha x_1 + (1-\alpha)x_2$, where $0 \leq \alpha \leq 1$. There exist λ and μ satisfying

$$\begin{aligned} x_1 &= b_1 + B_1\lambda \\ 0 &= b_2 + B_2\lambda \\ x_2 &= c_1 + C_1\mu \\ 0 &= c_2 + C_2\mu \end{aligned}$$

Thus x_3 corresponds to a point in S_3 for which $\alpha_1 = \alpha$, $\alpha_2 = (1-\alpha)$, $\lambda_1 = \lambda\alpha_1$, and $\mu_2 = \alpha_2\mu$. This shows that S_3 contains all the interior points in the convex hull of $S_1 \cup S_2$, and since S_3 is a closed set, it contains the boundary points as well. To see that the closure of the convex hull of $S_1 \cup S_2$ contains S_3, let $x_3 \in S_3$ be generated by α_1, $\alpha_2 = (1-\alpha_1)$, λ_1, and μ_2 that satisfy rows 2 and 3 of Eq. 2.2. If $0 < \alpha_1 < 1$, define $\alpha = \alpha_1$, let $x_1 \in S_1$ be generated by $\lambda = \lambda_1/\alpha_1$, and let $x_2 \in S_2$ be generated by $\mu = \mu_2/\alpha_2$. (Clearly λ and μ satisfy row 2 of their respective matrices in Eq. 2.1.) But $x_3 = \alpha x_1 + (1-\alpha)x_2$, so x_3 is in the convex hull. If $\alpha_1 = 0$, consider a sequence of points $x_3^{(1)}, x_3^{(2)}, \ldots, x_3^{(k)}, \ldots$ generated by the same λ_1 and μ_2, but a decreasing sequence $\alpha_1^{(k)} = 1/k$. By the preceding argument, each $x_3^{(k)}$ is in the convex hull and the sequence converges to x_3, so x_3 is in its closure. Similarly, if $\alpha_1 = 1$, consider an increasing sequence of $\alpha_1^{(k)} = 1 - \frac{1}{k}$. ▮

Example 2.2: Let S_1 and S_2 correspond to Example 2.1:

$$\begin{array}{c} \\ \left[\begin{array}{cc||c|c} 1 & 0 & 0 & \\ 0 & 1 & 1 & \end{array}\right] \end{array} \quad \text{and} \quad \begin{array}{r} (\lambda) \\ \left[\begin{array}{cc||c|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{array}\right] \end{array}$$

Then $S_1 \overline{\cup} S_2$ is given by:

$$\begin{array}{cccccc} & & & (\alpha_1) & (\alpha_2) & (\lambda_2) \\ \left[\begin{array}{c} 1\,0 \\ 0\,1 \\ 0\,0 \end{array}\right\Vert\right. & \begin{array}{r} 0 \\ 0 \\ -1 \end{array} & \Big| & \begin{array}{c} 0 \\ 1 \\ 1 \end{array} & \begin{array}{c} 0 \\ 0 \\ 1 \end{array} & \left.\begin{array}{c} 1 \\ 0 \\ 0 \end{array}\right] \end{array}$$

The third line specifies $\alpha_1 + \alpha_2 = 1$. Together with the requirement that all variables be nonnegative, this implies $0 \leq \alpha_1 \leq 1$. But the second line specifies that $x_2 = \alpha_1$, so we see that these equations specify the same polycone as the CHU found in Example 2.1. □

2.2 Removing Redundant Rows and Columns

We should point out that the representation of the CHU given by Eq. 2.2 may be highly redundant. This is the price we pay for being able to form it quickly. Operations to be introduced later also tend to introduce redundancy. Here we consider a number of situations in which lower rows and/or right-hand columns of the matrix specifying a polycone can be determined to be redundant. A row and/or column is considered to be redundant if removing it from the matrix leaves a matrix that specifies the same polycone. Two matrices are said to be equivalent ($\equiv$) in this context if they specify the same polycone.

Removal of redundancy in linear inequalities has been studied extensively in the context of linear programming [13, 5, 4], and more recently by Lassez and McAloon in the context of "constraint logic programming" [6] (*q.v.* for further bibliography). However, we consider only operations that preserve the generalized Tucker form of the equations. Our operations bear no obvious correspondence to the redundancy classes described by Lassez and McAloon, partly because of the differences in representation (they use a "solved form" consisting of equalities and inequalities). However, the possible relationship merits further study.

Intuitively, a lower row (i.e., one that does not involve the dependent variables) is redundant if it does not restrict the possible values of other independent variables. This situation is most easily identified when the matrix has the form shown below on the left, where C is a row vector, c is a scalar, and both are nonnegative.

$$\begin{array}{cc|c|rc} & & & (\lambda) & \\ \left[\right. I & b & & 0 & B \left.\right] \\ \end{array}$$

$$\left[\begin{array}{c||c|rc} & & (\lambda) & \\ I & b & 0 & B \\ 0 & c & -1 & C \\ 0 & d & 0 & D \end{array}\right] \equiv \left[\begin{array}{c||c|c} I & b & B \\ 0 & d & D \end{array}\right] \quad \text{if } c \geq 0, C \geq 0. \tag{2.3}$$

Clearly, any solution to the matrix on the right can be augmented by a nonnegative value of λ to give an equivalent solution to the matrix on the left (i.e., the value of λ can be decided last); also any solution to the left matrix is a solution to the right one. Thus the two matrices define the same set of points and the left one can be reduced to the right one.

This simplification is important because it seems to occur frequently in practice and is very efficient to recognize. All that is needed is to identify a lower row in which one variable's column has opposite sign from all other nonzero entires in that row. Then pivoting (described below) sets up Eq. 2.3.

Another situation that allows deletion of one row and two columns occurs when the matrix has the form shown below on the left, where C is any row vector and d_1 and d_2 are nonnegative scalars.

$$\left[\begin{array}{c||c|crc} & & (\lambda) & (\mu) & \\ I & b & 0 & 0 & B \\ 0 & c & d_1 & -d_2 & C \\ 0 & e & 0 & 0 & E \end{array}\right] \equiv \left[\begin{array}{c||c|c} I & b & B \\ 0 & e & E \end{array}\right] \quad \text{if } d_1 \geq 0, d_2 \geq 0. \tag{2.4}$$

Clearly, any solution to the matrix on the right can be augmented by nonnegative values of λ and μ to give an equivalent solution to the matrix on the left; also any solution to the left matrix is a solution to the right one.

Thus the two matrices define the same set of points and the left one can be reduced to the right one.

Pivoting operations are usually needed to set up such situations. The operation of "pivoting on (i, j)" consists of adding the appropriate multiple of row i to each other row to cause its entry in column j to become 0. An effective method of choosing pivot elements to expose redundancies is a topic for further investigation.

Another form of redundancy similar to the previous one, but something of a special case is shown below. Here some positive combination of independent variables equals zero, so they all must be zero. Hence their columns can be deleted.

$$\left[\begin{array}{c||c|cc} I & a & b & B \\ 0 & 0 & c & C \\ 0 & d & e & E \end{array}\right] \equiv \left[\begin{array}{c||c|c} I & a & B \\ 0 & 0 & C \\ 0 & d & E \end{array}\right] \qquad \text{if } c > 0,\ C \geq 0 \tag{2.5}$$

The diagram shows only one column being deleted, but the process continues until C has no positive elements left. Then row 2 is all zeros and is deleted. The scan for this situation can be incorporated into the scan for the situation of Eq. 2.3, and of course, if a negative combination is found, the row can be multiplied by -1 to set up Eq. 2.5.

If some right-hand column is a positive linear combination of other right columns, it may be deleted. For example, in the matrix below left, suppose that the (single) column for α is expressible as a nonnegative linear combination ν^{T} of the columns for λ. (Superscript "T" denotes "transpose," and converts a column vector to a row vector.) If $\alpha = \alpha_1$, $\lambda = \lambda_1$ generates some point x_1, then that same point can be generated by $\alpha = 0$ and $\lambda = \lambda_1 + \nu\alpha_1$.

$$\begin{array}{c} \begin{array}{cccc} & & (\alpha) & (\lambda) \end{array} \\ \left[\begin{array}{c||c|cc} I & a & b & B \\ 0 & d & e & E \end{array}\right] \end{array} \equiv \left[\begin{array}{c||c|c} I & a & B \\ 0 & d & E \end{array}\right] \quad \text{if } \left[\begin{smallmatrix} b \\ e \end{smallmatrix}\right] = \nu^{\mathrm{T}} \left[\begin{smallmatrix} B \\ E \end{smallmatrix}\right] \text{ for some } \nu \geq 0 \tag{2.6}$$

It follows that the column for α can be deleted without removing any points from the set defined by the matrix. Unlike previous cases, there is no row deletion associated with this simplification.

An important special case of Eq. 2.6 is when one column is a positive multiple of another. (Negative multiples are treated below.) This occurs often in practice and is fairly efficient to recognize.

It turns out that if a right-hand column is a *negative* linear combination of other right-hand columns and has a non-zero entry in a lower row, then it can be deleted eventually, but some preliminary row operations are necessary. The row operations pivot on this lower nonzero element, making it the only nonzero element of the column to be deleted. Columns remain the same linear combinations of each other after row operations. This situation after pivoting is shown on the left below, where the column containing the scalar

d is assumed to be some negative linear combination $(-\nu^{\mathrm{T}})$ of the other right-hand columns.

$$\left[\begin{array}{c||c|cc} I & b & 0 & B \\ 0 & c & d & C \\ 0 & e & 0 & E \end{array}\right] \equiv \left[\begin{array}{c||c|ccc} I & b & 0 & 0 & B \\ 0 & c & d & -d & C \\ 0 & e & 0 & 0 & E \end{array}\right] \equiv \left[\begin{array}{c||c|c} I & b & B \\ 0 & e & E \end{array}\right] \tag{2.7}$$

$$\text{if} \quad \left[\begin{array}{c} 0 \\ -d \\ 0 \end{array}\right] = \nu^{\mathrm{T}} \left[\begin{array}{c} B \\ C \\ E \end{array}\right] \quad \text{for some } \nu \geq 0$$

The result is obtained by first applying Eq. 2.6 from right to left, then applying Eq. 2.4.

The technique of combining pivoting operations with the deletion (or sometimes the introduction) of redundant rows is a useful tool for proving properties of polycones. This technique is used in the next corollary to derive a formula for the CHU of three polycones. The extension to a k-ary CHU is obvious.

Corollary 2.2: Let three polycones S_1, S_2 and S_3 be specified by

$$\left[\begin{array}{c||c|c} I & b_1 & B_1 \\ 0 & b_2 & B_2 \end{array}\right] \quad \text{and} \quad \left[\begin{array}{c||c|c} I & c_1 & C_1 \\ 0 & c_2 & C_2 \end{array}\right] \quad \text{and} \quad \left[\begin{array}{c||c|c} I & d_1 & D_1 \\ 0 & d_2 & D_2 \end{array}\right] \tag{2.8}$$

respectively. Then the closure of the convex hull of their union, is the set S_4 given by:

$$\left[\begin{array}{c||c|cccccc} I & 0 & b_1 & B_1 & c_1 & C_1 & d_1 & D_1 \\ 0 & 0 & b_2 & B_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_2 & C_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d_2 & D_2 \\ 0 & -1 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right] \tag{2.9}$$

Consequently, binary CHU is associative and commutative, and the notation $S_1 \mathbin{\bar{\cup}} S_2 \mathbin{\bar{\cup}} S_3$ is unambiguous.

Proof: By Theorem 2.1, the matrix for $(S_1 \mathbin{\bar{\cup}} S_2) \mathbin{\bar{\cup}} S_3$ is:

$$\left[\begin{array}{c||c|ccccccc} I & 0 & 0 & b_1 & B_1 & c_1 & C_1 & d_1 & D_1 \\ 0 & 0 & 0 & b_2 & B_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_2 & C_2 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d_2 & D_2 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right]$$

Add row 4 to row 6, giving:

$$\left[\begin{array}{c||c|ccccccc} I & 0 & 0 & b_1 & B_1 & c_1 & C_1 & d_1 & D_1 \\ 0 & 0 & 0 & b_2 & B_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_2 & C_2 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d_2 & D_2 \\ 0 & -1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right]$$

Now, as argued in connection with Eq. 2.3, row 4 (a genuine single row) is redundant because it can always be satisfied *after* all other rows are satisfied, and hence does not restrict the set of points in the defined polycone. Delete row 4 and column 1 (again, recall the numbering scheme), giving Eq. 2.9. ▮

3 Transformations that Correspond to Logic Procedures

Now we describe how a polycone in $R^n \times R^n$ represents a transformation on polycones in R^n. Let $x \in R^n$, $y \in R^n$, and let a set of equations or equivalent matrix be given:

$$\begin{aligned} x &= c_1 + C_1\rho \\ y &= c_2 + C_2\rho \\ 0 &= c_3 + C_3\rho \end{aligned} \qquad \left[\begin{array}{cc||c|c} I & 0 & c_1 & C_1 \\ 0 & I & c_2 & C_2 \\ 0 & 0 & c_3 & C_3 \end{array}\right] \tag{3.1}$$

which specifies a polycone in $R^n \times R^n$. Let A be any polycone in R^n, specified by:

$$\begin{aligned} y &= a_1 + A_1\lambda \\ 0 &= a_3 + A_3\lambda \end{aligned} \qquad \left[\begin{array}{c||c|c} I & a_1 & A_1 \\ 0 & a_3 & A_3 \end{array}\right] \tag{3.2}$$

Definition 3.2: The *natural transformation* Ψ associated with Eq. 3.1 is the mapping that takes any polycone A, given by Eq. 3.2, into the polycone $\Psi(A)$ (also in R^n) specified by:

$$\begin{aligned} x &= c_1 + C_1\rho \\ 0 &= c_2 - a_1 + C_2\rho - A_1\lambda \\ 0 &= c_3 + C_3\rho \\ 0 &= a_3 + A_3\rho \end{aligned} \qquad \left[\begin{array}{c||c|cc} I & c_1 & C_1 & 0 \\ 0 & (c_2 - a_1) & C_2 & -A_1 \\ 0 & c_3 & C_3 & 0 \\ 0 & a_3 & 0 & A_3 \end{array}\right] \tag{3.3}$$

In terms of matrices,

$$\left[\begin{array}{cc||c|c} I & 0 & c_1 & C_1 \\ 0 & I & c_2 & C_2 \\ 0 & 0 & c_3 & C_3 \end{array}\right] \text{ acting on } \left[\begin{array}{c||c|c} I & a_1 & A_1 \\ 0 & a_3 & A_3 \end{array}\right]$$

$$\text{produces } \left[\begin{array}{c||c|cc} I & c_1 & C_1 & 0 \\ 0 & (c_2 - a_1) & C_2 & -A_1 \\ 0 & c_3 & C_3 & 0 \\ 0 & a_3 & 0 & A_3 \end{array}\right]$$

□

Definition 3.3: *Imported constraints* consist of constraints on argument sizes of subgoals from other (lower) SCCs that were derived when those SCCs were analyzed (Sect. 1.6). □

Now suppose the logic procedure for p consists of several nonrecursive rules and several simple recursive rules. (Recall that simple recursion means p recurs only on itself.) Each nonrecursive rule has a set of equations, comprising its argument size equations (Sect. 1.4) and imported constraints. The set of all these equations for one rule can be expressed in the form of Eq. 3.2, which in turn defines a polycone. Let B be the CHU of all such polycones. Clearly, B contains the set of feasible argument sizes of p arising immediately from the nonrecursive rules. The equations specifying B are of the same general form, which we represent by:

$$\begin{aligned} x &= b_1 + B_1\lambda \\ 0 &= b_3 + B_3\lambda \end{aligned} \qquad \left[\begin{array}{c||c|c} I & b_1 & B_1 \\ 0 & b_3 & B_3 \end{array}\right] \tag{3.4}$$

Similarly there are argument size equations and imported constraints for each simple recursive rule, which may be written in the form of Eq. 3.1, and interpreted to represent a polycone in R^{2n}. Again, the CHU of the polycones for all the recursive rules is of the same form. Therefore, let us say that the CHU is in fact given by Eq. 3.1.

Example 3.1: The following procedure is intended to merge its first two arguments, which should be sorted lists, and instantiate the third argument to the result. In order to assure balanced treatment of the two "input" lists, they are interchanged upon recursion. The Prolog style of using *Xs* to name a list of *X*, etc., has been adopted.

$$\begin{array}{l} \mathit{merge}(\square,\ Ys,\ Ys). \\ \mathit{merge}(Xs,\ \square,\ Xs). \\ \mathit{merge}(X \cdot Xs,\ Y \cdot Ys,\ X \cdot Zs) \leftarrow X \le Y,\ \mathit{merge}(Y \cdot Ys,\ Xs,\ Zs). \\ \mathit{merge}(X \cdot Xs,\ Y \cdot Ys,\ Y \cdot Zs) \leftarrow Y \le X,\ \mathit{merge}(Ys,\ X \cdot Xs,\ Zs). \end{array}$$

The matrices for the two nonrecursive rules are shown below.

$$\left[\begin{array}{c||c|c} & 0 & 0 \\ I_3 & 0 & 1 \\ & 0 & 1 \end{array}\right] \qquad \left[\begin{array}{c||c|c} & 0 & 1 \\ I_3 & 0 & 0 \\ & 0 & 1 \end{array}\right] \tag{3.5}$$

Consequently the polycone B for *merge* is their CHU:

$$B \equiv \left[\begin{array}{c||c|c} & 0 & 0\,0\,0\,1 \\ I_3 & 0 & 0\,1\,0\,0 \\ & 0 & 0\,1\,0\,1 \\ 0 & -1 & 1\,0\,1\,0 \end{array}\right] \equiv \left[\begin{array}{c||c|c} & 0 & 0\,1 \\ I_3 & 0 & 1\,0 \\ & 0 & 1\,1 \end{array}\right] \tag{3.6}$$

The argument size equations for the two recursive rules lead to the matrices shown below. The assumed built-in predicate "$\le$" happens to provide no imported constraints, so there are no rows for C_3 of the individual rules. The right-hand columns have labels over them to assist in seeing how the matrices are derived.

$$\begin{array}{c}
\begin{array}{cc|ccccc}
(x)\ (y) & & (X) & (Xs) & (Y) & (Ys) & (Zs)
\end{array} \\
\left[\begin{array}{c||c|ccccc}
 & 2 & 1 & 1 & 0 & 0 & 0 \\
I_3 \ \ 0 & 2 & 0 & 0 & 1 & 1 & 0 \\
 & 2 & 1 & 0 & 0 & 0 & 1 \\
 & 2 & 0 & 0 & 1 & 1 & 0 \\
0 \ \ I_3 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right] \\
\\
\begin{array}{cc|ccccc}
(x)\ (y) & & (X) & (Xs) & (Y) & (Ys) & (Zs)
\end{array} \\
\left[\begin{array}{c||c|ccccc}
 & 2 & 1 & 1 & 0 & 0 & 0 \\
I_3 \ \ 0 & 2 & 0 & 0 & 1 & 1 & 0 \\
 & 2 & 0 & 0 & 1 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 \ \ I_3 & 2 & 1 & 1 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]
\end{array} \tag{3.7}$$

The CHU of these two matrices defines the matrix underlying Ψ for *merge* (per Definition 3.2), at least in its initial form:

$$\left[\begin{array}{c||c|c}
I\ 0 & c_1 & C_1 \\
0\ I & c_2 & C_2 \\
0\ 0 & c_3 & C_3
\end{array}\right] \equiv
\left[\begin{array}{c||c|cccccccccccc}
 & 0 & 2 & 1 & 1 & 0 & 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 \\
I_3 \ \ 0 & 0 & 2 & 0 & 0 & 1 & 1 & 0 & 2 & 0 & 0 & 1 & 1 & 0 \\
 & 0 & 2 & 1 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 1 & 0 & 1 \\
 & 0 & 2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 \ \ I_3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 \ \ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}\right] \tag{3.8}$$

This matrix has a number of redundant columns: numbers 5, 8, 9, 11, and 12. After eliminating them, we can express Ψ as:

$$\Psi \left[\begin{array}{c||c|c}
 & b_1 & a_1 \\
I_3 & b_2 & a_2 \\
 & b_3 & a_3 \\
0 & c & C
\end{array}\right] =
\left[\begin{array}{c||c|cccccccc}
 & 0 & 2 & 1 & 1 & 0 & 0 & 2 & 0 & 0 \\
I_3 & 0 & 2 & 0 & 0 & 1 & 0 & 2 & 1 & 0 \\
 & 0 & 2 & 1 & 0 & 0 & 1 & 2 & 1 & 0 \\
 & -b_1 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & -a_1 \\
0 & -b_2 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & -a_2 \\
 & -b_3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -a_3 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 & C
\end{array}\right] \tag{3.9}$$

In the above equation, a_i are row vectors with as many columns as matrix C, and c is a column vector with as many rows as C. □

The generalization of Ψ to SCCs with several predicates and rules with nonlinear recursion is cumbersome, but involves no new ideas; the method is outlined in Appendix A.

Definition 3.4: The *recursive transformation* T of a logic procedure for p, where B and Ψ are as defined in Eqs. 3.1–3.4 is the mapping that takes any polycone $A \subset R^n$ into the polycone $T(A) \subset R^n$, where:

$$T(A) = \Psi(A) \mathbin{\overline{\cup}} B \tag{3.10}$$

□

From their definitions it is clear that both Ψ and T are monotonic transformations. In order to study the structure of Ψ and T, we first observe that the polycones of R^n with operation $\mathbin{\overline{\cup}}$ form a commutative monoid.

Lemma 3.3: The transformation Ψ defined by Eqs. 3.1–3.4 is a homomorphism with respect to $\mathbin{\overline{\cup}}$; that is, for any polycones $P, Q \subset R^n$:

$$\Psi(P \mathbin{\overline{\cup}} Q) = \Psi(P) \mathbin{\overline{\cup}} \Psi(Q) \tag{3.11}$$

Proof: Let P and Q be given by:

$$P \equiv \left[\begin{array}{c||c|c} I & p_1 & P_1 \\ 0 & p_3 & P_3 \end{array}\right] \qquad Q \equiv \left[\begin{array}{c||c|c} I & q_1 & Q_1 \\ 0 & q_3 & Q_3 \end{array}\right] \tag{3.12}$$

Use Eq. 2.2 to form $P \mathbin{\overline{\cup}} Q$. Use Eq. 3.3 with $P \mathbin{\overline{\cup}} Q$ in the role of A:

$$\Psi(P \mathbin{\overline{\cup}} Q) \equiv \left[\begin{array}{c||c|ccccc} I & c_1 & C_1 & 0 & 0 & 0 & 0 \\ 0 & c_2 & C_2 & -p_1 & -P_1 & -q_1 & -Q_1 \\ 0 & c_3 & C_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_3 & P_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & q_3 & Q_3 \\ 0 & -1 & 0 & 1 & 0 & 1 & 0 \end{array}\right] \tag{3.13}$$

Similarly:

$$\Psi(P) \mathbin{\overline{\cup}} \Psi(Q) \equiv \left[\begin{array}{c||c|cccccc} I & 0 & c_1 & C_1 & 0 & c_1 & C_1 & 0 \\ 0 & 0 & (c_2{-}p_1) & C_2 & -P_1 & (c_2{-}q_1) & C_2 & -Q_1 \\ 0 & 0 & c_3 & C_3 & 0 & c_3 & C_3 & 0 \\ 0 & 0 & p_3 & 0 & P_3 & q_3 & 0 & Q_3 \\ 0 & -1 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}\right] \tag{3.14}$$

In Eq. 3.13, multiply the last row by c_1 and add it to row 1. Then multiply the last row by c_2 and add it to row 2. Finally, multiply the last row by c_3

and add it to row 3, giving:

$$\Psi(P \sqcup Q) \equiv \left[\begin{array}{c||c|ccccc} I & 0 & C_1 & c_1 & 0 & c_1 & 0 \\ 0 & 0 & C_2 & (c_2 - p_1) & -P_1 & (c_2 - q_1) & -Q_1 \\ 0 & 0 & C_3 & c_3 & 0 & c_3 & 0 \\ 0 & 0 & 0 & p_3 & P_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & q_3 & Q_3 \\ 0 & -1 & 0 & 1 & 0 & 1 & 0 \end{array}\right] \tag{3.15}$$

In Eq. 3.14 column 5 is a duplicate of column 2, so may be deleted. This also gives Eq. 3.15, except that columns 1 and 2 are interchanged. ▌

Corollary 3.4: For any polycone P, and $k > 0$,

$$T^k(P) = B \sqcup \Psi(B) \sqcup \Psi^2(B) \sqcup \ldots \sqcup \Psi^{k-1}(B) \sqcup \Psi^k(P) \tag{3.16}$$

Also, the sequence B, $T(B)$, $T^2(B)$, $\ldots$, $T^k(B)$, $\ldots$ is monotonic.

Proof: Eq. 3.16 follows by Lemma 3.3 and a trivial induction on k. Montonicity of $T^k(B)$ follows from Eq. 3.16 with $P = B$. ▌

4 The Search for a Fixpoint

Let B be the polycone specified by Eq. 3.4, which contains the set of feasible argument sizes arising from the nonrecursive rules. Let F be a polycone in the positive orthant of R^n that is a fixpoint of T; that is, F satisfies the equation $F = T(F)$. It is straightforward to show F contains the set of feasible argument sizes arising from *all* of the rules for p, and thus completes our analysis of p's SCC. That is, if $p(x)$ is derived without any recursions on p, then $x \in B \subseteq F$; if all $p(y)$ derived with $k-1$ recursions are in F, then all $p(x)$ derived in k recursions are in $\Psi(F) \subseteq F$; use induction on the number of applications of a recursive rule for p.

How to find such an F in general is not known at present. A solution in many simple examples is simply $F = T(B)$. If that does not work, one could try more iterates, but as the next example shows, this may also fail. A more complicated heuristic is described later in Section 7.

Example 4.1: This example shows that iterating T may not reach a fixpoint in a finite number of steps, even though one exists. The logic procedure below might test for precedence in some partial order, thinking of s as successor.

$$\begin{array}{l} p(X, X). \\ p(X, s(Y)) \leftarrow p(X, Y). \end{array}$$

We find that:

$$B \equiv \left[\begin{array}{cc||c|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{array}\right] \quad \Psi(B) \equiv \left[\begin{array}{cc||c|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right] \quad T(B) \equiv \left[\begin{array}{cc||c|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 1 \end{array}\right]$$

and for iterated applications of T:

$$T^k(B) \equiv \left[\begin{array}{cc||c|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & k & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 1 \end{array}\right]$$

It is clear that this sequence does not reach a fixpoint in a finite number of steps. However, we note that rescaling a column by a positive quantity does not change the polycone defined by the matrix, provided the column contains coefficients for a slack (independent) variable, i.e., it is a positively numbered column. Taking advantage of this, we divide column 1 of $T^k(B)$ by k, giving:

$$T^k(B) \equiv \left[\begin{array}{cc||c|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & \frac{1}{k} & 0 & 1 \end{array}\right] \xrightarrow{\text{as } k\rightarrow\infty} \left[\begin{array}{cc||c|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{array}\right]$$

$$\equiv \left[\begin{array}{cc||c|cc} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array}\right] \equiv F$$

It is now easy to verify that $T(F) = F$, so F is indeed the desired fixpoint. The inequalities repesented are $x_2 \geq x_1 \geq 0$.

This example is discussed further in Example 7.1. □

5 Verification of a Fixpoint

Suppose it is conjectured that a certain set of equations F specifies a fixpoint of T, namely:

$$F \equiv \left[\begin{array}{c||c|c} I & f_1 & F_1 \\ 0 & f_3 & F_3 \end{array}\right] \tag{5.1}$$

Substituting into the right side of Eq. 3.3, then forming the CHU with Eq. 3.4 gives

$$T(F) = \Psi(F) \mathbin{\overline{\cup}} B \equiv \left[\begin{array}{c||c|ccccc} I & 0 & c_1 & C_1 & 0 & b_1 & B_1 \\ 0 & 0 & (c_2{-}f_1) & C_2 & -F_1 & 0 & 0 \\ 0 & 0 & c_3 & C_3 & 0 & 0 & 0 \\ 0 & 0 & f_3 & 0 & F_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & b_3 & B_3 \\ 0 & -1 & 1 & 0 & 0 & 1 & 0 \end{array}\right] \tag{5.2}$$

To verify the conjecture, we must show that Eq. 5.2 is equivalent to (represents the same polycone as) Eq. 5.1. One method is to use the earlier observations about redundancies, or other arguments, to transform Eq. 5.2 back to Eq. 5.1. Although this can be done *ad hoc* on examples, an efficient general algorithm for finding redundancies is not known. A general method of deciding equivalence, based on conversion to normal forms, is described in the next section.

Example 5.1: Using the matrices for the *rev* procedure of Example 1.1, we get

$$B \equiv \left[\begin{array}{c||c|c} & 0 & 0 \\ I_3 & 0 & 1 \\ & 0 & 1 \end{array}\right]$$

and

$$\Psi(B) \equiv \left[\begin{array}{c||c|ccccc} & 2 & 1 & 1 & 0 & 0 & 0 \\ I_3 & 0 & 0 & 0 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 1 & 0 \\ & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 & -1 \\ & 0 & 0 & 0 & 0 & 1 & -1 \end{array}\right] \equiv \left[\begin{array}{c||c|cc} & 2 & 1 & 0 \\ I_3 & 0 & 0 & 1 \\ & 2 & 1 & 1 \end{array}\right] \qquad (5.3)$$

The second form of $\Psi(B)$ was obtained by the following series of simplifications (equation numbers in parentheses provide the justifications): Subtract row 6 from row 5 and delete row 6 and col. 5 (by 2.3). Add row 5 to row 3 and delete row 5 and col. 4 (by 2.3). Delete col. 2 and row 4 (by 2.5).

Using the reduced form of $\Psi(B)$, we compute $T(B) = \Psi(B) \mho B$ as:

$$T(B) \equiv \left[\begin{array}{c||c|ccccc} & 0 & 2 & 1 & 0 & 0 & 0 \\ I_3 & 0 & 0 & 0 & 1 & 0 & 1 \\ & 0 & 2 & 1 & 1 & 0 & 1 \\ 0 & -1 & 1 & 0 & 0 & 1 & 0 \end{array}\right] \equiv \left[\begin{array}{c||c|cc} & 0 & 1 & 0 \\ I_3 & 0 & 0 & 1 \\ & 0 & 1 & 1 \end{array}\right] \qquad (5.4)$$

The second form of $T(B)$ was obtained by the following series of simplifications: Delete col. 5, as it equals col. 3 (by 2.6). Delete col. 1, as it equals col. 4 + 2×col. 2 (by 2.6). Delete row 4 and original col. 4 (by 2.3).

We define $F = T(B)$ as the hypothetical fixpoint, and try to verify it:

$$\Psi(F) \equiv \left[\begin{array}{c||c|cccccc} & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ I_3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 & 0 & -1 \\ & 0 & 0 & 0 & 0 & 1 & -1 & -1 \end{array}\right] \equiv \left[\begin{array}{c||c|cc} & 2 & 1 & 0 \\ I_3 & 0 & 0 & 1 \\ & 2 & 1 & 1 \end{array}\right] \qquad (5.5)$$

To get the simpler form we subtract row 6 from row 3, add row 5 to row 3, and add row 4 to row 3. Then we delete in turn rows 6, 5, and 4. This makes makes columns 4–6 zero, so they are deleted. Finally, col. 2 is a duplicate of col. 1, so is deleted. Now we note that $\Psi(F) = \Psi(B)$, so it follows that $T(F) = T(B)$ by definition of T. We have verified that F is indeed a fixpoint of T.

By inspection, we see that Eq. 5.4 is equivalent to the set of constraints: $x_1 \geq 0$, $x_2 \geq 0$, and $x_3 = x_1 + x_2$. This is precisely what we expect for $rev(x_1, x_2, x_3)$, and represents a tight bound.

We observe that the methods developed in [15] do not handle constraints among three variables. Thus those methods could conclude only $x_3 \geq x_1$ and $x_3 \geq x_2$. While these weaker constraints are sufficient to prove termination for top-down evaluation of *rev* itself, they may not be sufficient for whatever procedures *use* *rev*. □

6 Normal Forms and Polycone Equivalence

In this section we use the machinery of convex analysis and linear programming to develop a normal form representation of polycones and an algorithm to convert any matrix representation of a polycone to its normal form. This algorithm provides a decision procedure for the question of whether two matrices represent the same polycone.

We begin with some standard definitions [11]. For these definitions, let A be a convex set in R^n. A *ray* is a closed half-line, i.e., a set of points $\{a + d\theta \mid \theta \geq 0\}$, where a and d are vectors in R^n and θ is a scalar.

Definition 6.1: Vector $d \in R^n$ is a *direction of recession* of A if $\exists a \in A$ such that the ray $\{a + d\theta \mid \theta \geq 0\}$ is contained in A. □

If A is nonempty, then "$\exists a \in A$" can be replaced by "$\forall a \in A$" in the above definition.

Definition 6.2: An *extreme point* of A is a point that cannot be expressed as a nontrivial convex combination of (other) points in A. An *extreme direction* of A is a direction of recession that cannot be expressed as a nontrivial convex combination of (other) directions of recession of A. □

It can be shown [11, Theorem 19.6] that a convex set P is polyhedral (Definition 1.1) if and only if it has a finite set of extreme points $\{c_i\}$ and a finite number of extreme directions $\{d_j\}$. This set of points and directions is said to *finitely generate* P; that is, point $a \in P$ if and only if a has a representation as

$$a = \sum \lambda_i c_i + \sum \mu_j d_j \qquad \lambda \geq 0, \quad \sum \lambda_i = 1, \quad \mu \geq 0$$

Now let us restrict P to be a polycone, that is, restrict it to lie in the positive orthant of R^n. Let C be the matrix whose columns are c_i, the extreme points of P. Let D be the matrix whose columns are d_j, the extreme directions of P. Clearly, each $c_i \geq 0$ and each $d_j \geq 0$. Let $\mathbf{1}$ denote a row vector of 1's. A generalized Tucker representation for polycone P is given by

$$\begin{array}{c||c|cc} & & (\lambda) & (\mu) \\ I & 0 & C & D \\ 0 & -1 & \mathbf{1} & 0 \end{array}$$

Definition 6.3: Let matrices C and D be as described above for polycone P. Further, let the elements of each column d_j be relatively prime integers, let the columns of D be arranged lexicographically, and let the columns of C be arranged lexicographically. Then the generalized Tucker representation:

$$\left[\begin{array}{c||c|cc} I & 0 & C & D \\ 0 & -1 & \mathbf{1} & 0 \end{array}\right] \tag{6.1}$$

is called the *normal form* representation of P. □

It is clear that the normal form of a given polycone is unique. Thus, to verify whether two representations are actually the same polycone, as is necessary to verify fixpoints, it is sufficient to reduce each to its normal form. This boils down to the following problem: given some matrix that represents a polycone, find its set of extreme points and directions.

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 [9, Ch. 2], and elsewhere; we review the essentials briefly. Assume we have transformed a generalized Tucker representation into a *standard form* linear programming problem, 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.

A *basic feasible solution* (bfs) is a nonnegative vector X_0 such that

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

$X_{0\mathcal{B}(i)} = i$-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.

Now suppose A_j is any column not in a current basis B that corresponds to bfs X_0. Then it is a linear combination of the columns of B, so satisfies

$$A_j = B\,X_j \tag{6.3}$$

for a certain vector X_j. We recall that the simplex algorithm maintains a "tableau", the matrix X, with a zero-th column X_0 that is the current bfs; the columns $X_{\mathcal{B}(i)}$ comprise an $m \times m$ identity matrix; column X_j for each $j \notin \mathcal{B}$ satisfies Eq. 6.3.

For θ a nonnegative scalar, we have the identity

$$B\,(X_0 - \theta X_j) + \theta\,A_j = b \tag{6.4}$$

We can "move" away from vertex X_0 in direction X_j by increasing θ from 0. We remain in the feasible region as long as $(X_0 - \theta X_j) \geq 0$. To "pivot" in the simplex algorithm, we need to determine the maximum value of θ that stays in the feasible region. We have three cases:

1. For some row k, we have $X_{kj} > 0$ and $X_{k0} = 0$. Then $\theta_{\max} = 0$.
2. For some k, $X_{kj} > 0$, and for all k such that $X_{kj} > 0$, we also have $X_{k0} > 0$. Then $\theta_{\max} > 0$, but finite. In this case "moving to $\theta_{\max}$" corresponds to traversing an edge of P to another vertex.
3. For no k is $X_{kj} > 0$. Then θ can increase indefinitely and $(X_0 - \theta X_j)$ remains feasible. Evidently, $-X_j$ is a direction of recession of P.

Most linear programming texts make an early assumption that the feasible region is bounded, so that case (3) cannot occur; consequently, we cannot use their results directly. The modifications for unbounded feasible regions are fairly straightforward, and we present them below.

Algorithm (6.1): To find the extreme points and directions of P, the feasible region of Eq. 6.2.

METHOD: (Outline)

> Visit all the vertices of P by pivoting and never choosing a column that fits case (3) above. Use standard methods, such as depth first search, to ensure that each vertex is visited once.
>
> At each vertex (bfs), determine which nonbasis columns fall into case (3). The associated directions, $-X_j$, are the extreme directions of P.

□

To justify this algorithm's correctness, we need to prove two facts:

1. Every instance of case (3) gives an extreme direction.
2. Every extreme direction arises as an instance of case (3). That is, if d is an extreme direction, then there is some vertex c such that $c + \theta d$ describes a one-dimensional face of P.

The following lemmas address these problems.

Lemma 6.1: Every instance of case (3) gives an extreme direction.

Proof: It is sufficient to show that $\{(X_0 - \theta X_j) \mid \theta \geq 0\}$ is a (one-dimensional) face of P. We use the fact that a subset of P is a face if and only if there is some linear function $h(\xi)$ that takes its maximum within P precisely on that set [11, Sect. 18]. A suitable $h(\xi)$ is obtained by setting the coefficients of the basis elements and the j-th element to 0, and setting the remaining coefficients to -1. Then the maximum, which is 0, is attained precisely on the desired ray. ▮

Lemma 6.2: If d is an extreme direction of polycone P, then there is some vertex (extreme point) c such that $\{c + \theta d \mid \theta \geq 0\}$ describes a one-dimensional face of P.

Proof: Let H be the hyperplane through the origin that is orthogonal to d, and consider the projection P' of polycone P on H. P' is polyhedral, and any extreme point of P' must have an extreme point of P in its inverse image. If P' has an extreme point, choose any extreme point c in its inverse image. If P' has no extreme points, then clearly d is not an extreme direction of P. (We can find a pair $(d', -d')$ of directions of recession of P' whose inverse images can be copmbined to yield d in P.) ▮

Theorem 6.3: It is decidable whether two generalized Tucker representations with rational coefficients define the same polycone.

Proof: Algorithm 6.1 finds the extreme points and extreme directions of P in the space of all variables, ξ. It is only necessary to project them onto the x components of ξ, express directions with relatively prime integers, and arrange columns in lexicographic order to obtain the normal form of Eq. 6.1. Two representations define the same polycone if and only if their normal forms are the same. ▮

Example 6.1: Consider the equations developed in Example 3.1 for the *merge* procedure. We sketch the evaluation of the normal form of $\Psi(B)$, where Ψ was given by Eq. 3.9 and B by Eq. 3.6. By substitution,

$$\Psi \left[\begin{array}{c||c|cc} & 0 & 0 & 1 \\ I_3 & 0 & 1 & 0 \\ & 0 & 1 & 1 \end{array}\right] = \left[\begin{array}{c||c|ccccccccc} & 0 & 2 & 1 & 1 & 0 & 0 & 2 & 0 & 0 & 0 \\ I_3 & 0 & 2 & 0 & 0 & 1 & 0 & 2 & 1 & 0 & 0 \\ & 0 & 2 & 1 & 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & -1 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & -1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \tag{6.5}$$

To put this in the form of Eq. 6.2, we move the constants column to the left side, move the coefficients of x to the right side, and multiply the three upper rows by -1. In addition, the columns are labeled with variable names:

$$\begin{array}{c||c|cccccccccccc} & & (x_1) & (x_2) & (x_3) & (\lambda_1) & (\lambda_2) & (\lambda_3) & (\lambda_4) & (\lambda_5) & (\lambda_6) & (\lambda_7) & (\lambda_8) & (\lambda_9) \\ 0 & & 1 & 0 & 0 & -2 & -1 & -1 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & & 0 & 1 & 0 & -2 & 0 & 0 & -1 & 0 & -2 & -1 & 0 & 0 \\ 0 & & 0 & 0 & 1 & -2 & -1 & 0 & 0 & -1 & -2 & -1 & 0 & 0 \\ 0 & & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & -1 & 0 \\ 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & -1 \\ 1 & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array} \tag{6.6}$$

The interested reader is invited to check our computation of Algorithm 6.1, which yields two extreme points:

$$\begin{aligned}(x,\lambda) &= (2,2,4,1,0,0,0,2,0,0,2,0)\\(x,\lambda) &= (2,2,4,0,0,0,0,2,1,0,0,2)\end{aligned}\tag{6.7}$$

and several extreme directions:

$$\begin{aligned}(x,\lambda) &= (0,1,1,0,0,0,0,0,0,0,0,0)\\(x,\lambda) &= (0,1,1,0,0,0,0,1,0,0,1,0)\\(x,\lambda) &= (0,1,1,0,0,0,1,1,0,0,0,0)\\(x,\lambda) &= (1,0,1,0,0,0,0,0,0,0,0,0)\\(x,\lambda) &= (1,0,1,0,0,0,0,1,0,0,0,1)\\(x,\lambda) &= (1,0,1,0,0,1,0,1,0,0,0,0)\end{aligned}\tag{6.8}$$

Projecting on x, we find the normal form to be:

$$\left[\begin{array}{c||c|ccc} & 0 & 2 & 0 & 1\\ I_3 & 0 & 2 & 1 & 0\\ & 0 & 4 & 1 & 1\\ 0 & -1 & 1 & 0 & 0\end{array}\right]\tag{6.9}$$

Geometrically, this states $x_1 \geq 2$, $x_2 \geq 2$, and $x_3 = x_1 + x_2$, which is precisely what we expect after one application of a recursive *merge* rule to a base case.

Interestingly, Eq. 6.9 can also be obtained efficiently by redundancy elimination procedures. Briefly, in Eq. 6.5 pivoting allows rows 4, 5, and 6 to be eliminated by setting up Eq. 2.3, then duplicate columns are coalesced, resulting in Eq. 6.9. Finally, we observe that $T(B) = \Psi(B) \mathbin{\bar{\cup}} B$ reduces to B, so B has been shown to be a fixpoint. □

7 An Heuristic that Often Works

The following polycone has been found to provide a fixpoint in several examples. Let Eqs. 3.1–3.4 describe the logic procedure. Let F be given by:

$$F \equiv \left[\begin{array}{c||c|cccccc} I & b_1 & c_1-b_1 & B_1 & -B_1 & C_1 & 0 & 0\\ 0 & b_3 & 0 & B_3 & 0 & 0 & 0 & 0\\ 0 & 0 & b_3 & 0 & B_3 & 0 & 0 & 0\\ 0 & 0 & c_2-b_1 & 0 & 0 & C_2 & -B_1 & 0\\ 0 & 0 & c_3 & 0 & 0 & C_3 & 0 & 0\\ 0 & 0 & b_3 & 0 & 0 & 0 & B_3 & 0\\ 0 & b_1 & c_1-b_1 & B_1 & -B_1 & C_1 & 0 & -I\end{array}\right]\tag{7.1}$$

This matrix was arrived at by trying to "enlarge $T(B)$ in the Ψ direction." Recall that a point $x \in T(B)$ can be represented as $\alpha(v-u)+u$, where $u \in B$, $v \in \Psi(B)$, and $0 \leq \alpha \leq 1$. If we drop the requirement $\alpha \leq 1$ we are in a sense projecting rays from B through $\Psi(B)$. If B and $\Psi(B)$ are disjoint, the portion of the set for which $\alpha \geq 1$ is called the *penumbra of* $\Psi(B)$ *with respect to* B [11]. Projecting B through $\Psi(B)$ does not necessarily produce a convex set. However, Eq. 7.1 specifies a polycone that does enclose the set so produced.

Example 7.1: Consider the same rules given in Example 5.2, in which no finite $T^k(B)$ was a fixpoint.

$$
\begin{array}{l}
p(X, X). \\
p(X, s(Y)) \leftarrow p(X, Y).
\end{array}
$$

The heuristic of Eq. 7.1 gives the following matrix for F, initially.

$$
F \equiv \left[\begin{array}{cc||c|cccccccc}
1 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & -1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 & 1 & -1 & 0 & -1 & 0 & 0 & -1
\end{array}\right] \tag{7.2}
$$

This matrix can be reduced as follows. Pivot on row 6, column 3. Columns 2 and 3, and row 6, are eliminated using Eq. 2.4. Rows 5, 4, and 3 are then redundant, and eliminated after pivoting. The result is the same matrix that was found to be a fixpoint in Example 4.1, namely:

$$
F \equiv \left[\begin{array}{cc||c|cc}
1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 1
\end{array}\right] \tag{7.3}
$$

□

8 Directions for Further Work

There are two principal directions in which this work needs to be extended. First, we would like to have a more efficient procedure for determining equivalence of polycones, and a good analysis of the running time. Second, and most important, we need more ways to generate candidates for the fixpoint.

Frequently, the "recursive activity" of an SCC occurs in only a small number of dimensions. A method that covers two or three "significant" dimensions would be useful.

One possibly exploitable property of $[b_1|B_1]$ $[c_1|C_1]$, and $[c_2|C_2]$, when they pertain to a single rule, is this: in any row with a positive constant column, the sum of the coefficients in the row cannot exceed the constant in that row; furthermore, when the constant for the row is zero, there is at most one positive coefficient, and that equals 1. Specifically, for $[b_1|B_1]$:

$$
\sum_j B_1^{(ij)} \leq \begin{cases} 1, & \text{if } b_1^{(i)} = 0; \\ b_1^{(i)}, & \text{if } b_1^{(i)} > 0. \end{cases}
$$

This is a consequence of the definition of term size, together with the fact that logical variables within the same term must be connected by function symbols of arity at least 2.

Looking at Eqs. 3.1–3.3 and Appendix A, we see that there is predictable sparseness in the generalized Tucker representations. It might be possible to take advantage of this.

Since T is monotonic, it has a least fixpoint, but this fixpoint (and others) may not be a polycone. It would be interesting to formulate conditions under which the least fixpoint of T is a polycone.

To conclude, while this paper offers a beginning, there is still much work to be done in the automatic analysis of argument term size constraints.

Acknowledgements

We wish to thank Jean-Louis Lassez for stimulating and helpful discussions. This work was partially supported by NSF grants CCR-89-58590 and IRI-89-02287.

Appendix A Programs with General Recursion

Here we outline how to construct the Ψ and T transformations in SCCs that have nonlinear recursion and/or several predicates in one SCC. Suppose there are s different predicates in the SCC, $p_1, \ldots, p_s$. The main idea is that we define $\Psi_1, \ldots, \Psi_s$ as mappings for those predicates. Each mapping operates on a *vector* of polycones, $A_1, \ldots, A_s$. Then Ψ for the whole SCC is the direct product:

$$\Psi(A_1, \ldots, A_S) = (\Psi_1(A_1, \ldots, A_s), \ldots, \Psi_1(A_1, \ldots, A_s))$$

Similarly, we formulate base case polycones $B_1, \ldots, B_s$ and use their direct product, together with Ψ, to define T for the whole SCC. To keep the notation reasonable, we illustrate the details for an SCC with two predicates p and q.

Consider a recursive rule for p with k p-subgoals, m q-subgoals, and possibly a nonrecursive subgoal. The argument size equations are

$$\begin{aligned} x &= c + C\rho \\ y_1 &= d_1 + D_1\rho \\ &\cdots \\ y_k &= d_k + D_k\rho \\ z_1 &= e_1 + E_1\rho \\ &\cdots \\ z_m &= e_m + E_m\rho \\ 0 &= f + F\rho \end{aligned} \qquad (A.1)$$

Here x represents argument sizes of p in the head of the rule; y_i represents argument sizes of the i-th p-subgoal; z_j represents argument sizes of the j-th q-subgoal; ρ contains variables corresponding to the logical variables in the rule. Now suppose the polycones to be operated upon by Ψ are presented as

$$\begin{aligned} y &= a + A\lambda & \qquad z &= b + B\mu \\ 0 &= g + G\lambda & \qquad 0 &= h + H\mu \end{aligned} \tag{A.2}$$

We substitute a separate copy (with λ renamed) of the left equations for each of y_i and substitute a separate copy of the right equations for each z_j (*cf.* Eqs. 3.1–3.3), giving:

$$\left[\begin{array}{c||c|ccccccc}
 & & (\rho) & (\lambda_1) & \ldots & (\lambda_k) & (\mu_1) & \ldots & (\mu_m) \\
I & c & C & 0 & \ldots & 0 & 0 & \ldots & 0 \\
0 & (d_1-a) & D_1 & -A & & 0 & & & \\
 & \vdots & \vdots & & \ddots & & & 0 & \\
0 & (d_k-a) & D_k & 0 & & -A & & & \\
0 & (e_1-b) & E_1 & & & & -B & & 0 \\
 & \vdots & \vdots & & 0 & & & \ddots & \\
0 & (e_m-b) & E_m & & & & 0 & & -B \\
0 & f & F & 0 & \ldots & 0 & 0 & \ldots & 0 \\
0 & g & 0 & G & & 0 & & & \\
 & \vdots & \vdots & & \ddots & & & 0 & \\
0 & g & 0 & 0 & & G & & & \\
0 & h & 0 & & & & H & & 0 \\
 & \vdots & \vdots & & 0 & & & \ddots & \\
0 & h & 0 & & & & 0 & & H
\end{array}\right] \tag{A.3}$$

This defines the feasible polycone for one rule! The CHU of all recursive rules for p defines Ψ_p evaluated at the pair of polycones in Eq. A.2. Ψ_q is defined similarly, based on the recursive rules for q, and Ψ is the direct product of Ψ_p and Ψ_q.

Now let the base case polycones for p and q be B_p and B_q, respectively. We define

$$\begin{aligned} T_p(P,Q) &= \Psi_p(P,Q) \mathbin{\bar{\cup}} B_p \\ T_q(P,Q) &= \Psi_q(P,Q) \mathbin{\bar{\cup}} B_q \\ T(P,Q) &= (T_p(P,Q),\, T_q(P,Q)) \end{aligned} \tag{A.4}$$

where (P,Q) is the vector of polycones to be transformed, represented by Eq. A.2.

References

[1] 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.

[2] 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.

[3] B. C. Eaves and U. G. Rothblum. *Elimination of Quantifiers of Linear Variables and Corresponding Transfer Principles.* Technical Report Operations Research, Stanford University, 1987.

[4] R. M. Freund, R. Roundy, and M. J. Todd. *Identifying the Set of Always-Active Constraints in a System of Linear Inequalities by a Single Linear Program.* Technical Report 1674-85, Sloan School of Management, MIT, 1985.

[5] 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.

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

[7] 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.

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

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

[10] L. Plümer. *Termination Proofs for Logic Programs.* PhD thesis, Dortman University, 1988.

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

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

[13] J. Telgen. Minimal representation of convex polyhedral sets. *Journal of Optimization Theory and Application*, 38(1):1–24, 1982.

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

[15] 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.

[16] A. Van Gelder. A message passing framework for logical query evaluation. In *1986 ACM-SIGMOD Conf. on Management of Data*, pages 155–165, 1986.

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