

Solving Linear Arithmetic Constraints

(Submitted for Publication, 4 Mar 2005)

Harald Rueß and Natarajan Shankar

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{ruess, shankar}@csl.sri.com
<http://www.csl.sri.com/{ruess, shankar}>

Abstract. Linear arithmetic constraints in the form of equalities and inequalities constitute the vast majority of proof obligations that arise in embedded applications of theorem proving such as extended typechecking, software and hardware verification, and compiler optimization. Such constraints involve the conjunction of equalities, inequalities, and disequalities over arithmetic, uninterpreted functions, and other datatypes. Nelson presented a practical scheme for solving linear inequalities based on the simplex algorithm for linear programming. We extend Nelson's version of the simplex algorithm in a number of ways. Among these extensions are an improved treatment of the combination of restricted (non-negative slack) and unrestricted variables, a method for adding equalities and disequalities to a simplex tableau, an optimized method for propagating equality information, and efficient techniques for generating proofs and models. Our algorithms are supported by simple and rigorous correctness arguments.

1 Introduction

Theorem proving in the form of constraint solving has a number of applications including hardware and software verification, program analysis, compiler optimization, AI planning, and scheduling. As one might expect, linear inequalities and equalities constitute a significant fraction of the constraints that arise naturally in such applications. There are many peculiarities to the use of constraint solving in theorem proving applications:

1. Not all constraints are in the form of linear equalities and inequalities so that the arithmetic solver must be used in combination with other solvers.
2. The constraints must be processed *incrementally* by means of an *online* algorithm.
3. Equality information derived by the inequality solver must be propagated to other inference procedures.
4. The procedure must admit queries that determine if a given constraint is redundant or inconsistent with respect to the prior constraints.
5. The procedure should produce *models*, *explanations* in terms of an approximation of the minimal set of input formulas leading to an unsatisfiability, or even *proofs* that are constructed from the input formulas according to inference rules for arithmetic equality and inequality.

There are several other incremental algorithms for solving inequalities including Fourier elimination [2] and Shostak’s loop residue method [?], but these are exponential in complexity even on simple examples. In contrast, simplex-based methods have the advantage of being polynomial in the average case. We give a unified presentation of a simplex-based algorithm for solving linear inequality, equality, and disequality constraints together with equality propagation for deriving implied equalities, and proof and model construction. Our algorithm is comprehensive and well-optimized, and is accompanied by rigorous proofs of correctness.

Related Work Our work builds on Nelson’s method presented in his thesis as a method for incrementally adding inequalities to a simplex tableau [6]. Nelson maintains a single tableau that mixes restricted and unrestricted variables. Equalities are derived in an incomplete way by maximizing the newly added tableau entry to see if it is maximized at 0. The addition of equalities $a = b$ into the tableau is performed inefficiently by adding $a \leq b$ and $a \geq b$. Retraction of assertions is carried out by recording and undoing pivoting steps. The Simplify prover [3] observes and fixes the incompleteness in equality propagation by maximizing all tableau entries. It also has a procedure for adding equalities that improves on the method of Nelson’s thesis. Necula [5] gives a proof-producing version of Nelson’s algorithm. The Cassowary [1] algorithm improves on Nelson’s method by separating the tableau into restricted and unrestricted parts, as is also the case in our algorithm, and introduces a simple and elegant mechanism for retracting assertions.

Hentenryck and Graf [4] present an elegant normal form called SF2 for the simplex tableau where the first non-zero coefficient in each entry is positive. The entries themselves can be ordered lexicographically and a lexicographic pivoting rule is used to maintain the SF2 normal form of the tableau. The SF2 representation leaves no implicit equalities in the tableau and yields a simple algorithm for constructing models satisfying disequalities. The top-level algorithm for processing inequalities is not spelled out by Hentenryck and Graf [4] and equalities are handled as in Nelson’s algorithm. Lexicographic pivoting is an alternative to Bland’s rule [9] for avoiding pivoting cycles, but can be more expensive, particularly in situations where implicit equalities are rare.

Compared to the previous work, our procedure is comprehensive in processing equality, inequality, and disequality constraints, together with equality propagation, and proof and model generation. For instance, Cassowary does not treat disequalities or equality propagation. Nelson’s procedure is incomplete for equality propagation and does not address proof and model generation. Necula’s proof generation algorithm is roughly similar to the one given here but his construction is more complicated and his justification is missing the key invariant of input satisfiability. Our inference-style presentation of the simplex-based algorithm admits easy correctness arguments. In contrast, the correctness arguments in prior work are sketchy or non-existent. Our algorithm is also more efficient than the prior algorithms. The rules are carefully optimized to avoid unnecessary pivoting steps. Redundant input assertions are eliminated efficiently. We also introduce an efficient analysis technique for detecting implicit equalities in a simplex tableau with minimal pivoting. The algorithms presented here can be employed within a Nelson-Open [7] or Shostak-like [8] combination framework.

2 Preliminaries

A *linear arithmetic expression* is a sum of monomials of the form $r_0 + \dots + r_n$. A *monomial* is either a rational constant k or an expression of the form $k * x$ for a variable x and rational constant k . The set of variables in an expression a is represented as $\text{vars}(a)$. The negation $-r$ of a monomial $k * x$ is $(-k) * x$, and the negation $-a$ of a linear arithmetic expression a of the form $r_0 + \dots + r_n$ is $-r_0 + \dots + -r_n$. An assignment ρ for a set of variables X maps the variables in X to rational numbers. The interpretation $\rho(e)$ of a term e with respect to an assignment ρ applies the mapping as a substitution and evaluates the resulting expression.

Linear arithmetic *literals* are of the form $a \geq 0$, $a = 0$ or $a \neq 0$, where a is a linear arithmetic expression and the relation symbols $=$, \geq , and \neq have their expected interpretation. A literal is *satisfiable* iff there is an assignment under which it holds. The special unsatisfiable literal \perp can be seen as an abbreviation for $0 = 1$.

A *constraint* is a finite set of literals. The set of variables in a constraint Γ is represented as $\text{vars}(\Gamma)$. A constraint is *satisfiable* if there is some assignment ρ of rational values to variables under which each literal in the constraint is satisfied. Otherwise, the set of constraints is unsatisfiable. The implication $\Gamma \Rightarrow \perp$ is *valid* when the constraint Γ is unsatisfiable, and this is indicated by the judgement $\models \Gamma \Rightarrow \perp$. The judgement $\models \Gamma \Rightarrow p$ for a literal p is an abbreviation for $\models \Gamma, \neg p \Rightarrow \perp$. We write $\Gamma \simeq \Gamma'$ when any satisfying assignment for Γ can be extended to one for Γ' , and vice-versa.

We assume a total ordering on the variables so that $x < y$ when the variable x precedes the variable y in the ordering. Given such an ordering, a linear arithmetic expression a can be placed in an *ordered sum-of-products* form $k_0 + k_1 * x_1 + \dots + k_n * x_n$, where each k_i ($i = 1, \dots, n$) is a nonzero rational constant and $x_i < x_j$ if $i < j$. An expression in the ordered sum-of-products form is said to be *canonical* and we write $a \equiv b$ when the canonical forms of a and b are identical. Note that $a = b$ is valid if and only if $a \equiv b$. If a is a canonical expression of the form $k_0 + k_1 x_1 + \dots + k_n x_n$, we write $|a|$ to represent the constant part k_0 , a^- to represent the negative part $\sum_{\{i:k_i < 0\}} k_i x_i$, and a^+ to represent the positive part $\sum_{\{i:k_i > 0\}} k_i x_i$. When e is canonical, the equality $a = |a| + a^- + a^+$ is valid. Finally, let $\kappa(a, v)$ represent the coefficient of v in the canonical expression a . We assume that arithmetic expressions are always maintained in canonical form.

The operation $\text{solve}(x)(a = b)$ returns the result of solving $a = b$ for a variable x in $\text{vars}(a - b)$. The result is \perp if $a = b$ is unsatisfiable (in the rationals), the empty set \emptyset if $a = b$ is valid, or the equivalent solved form for x . For example, $\text{solve}(x)(x = x)$ is \emptyset , $\text{solve}(x)(x = x + 1)$ is \perp , and $\text{solve}(x)(x + y - 3 = 0)$ is $x = 3 - y$. When the first argument is omitted, as in $\text{solve}(a = b)$, then we can pick any element of $\text{vars}(a - b)$ for the first argument.

The basic data structure of our simplex algorithm is a *solution set* S of the form $\{x_1 = a_1, \dots, x_n = a_n\}$, where $x_i \neq x_j$ for $i \neq j$, no x_i occurs in any a_j , and each a_j is in canonical form. For a solution set S , the *lookup* $S(x)$ of x in S is e if $(x = e) \in S$, and x otherwise. The *substitution* $S[a]$ represents the result of replacing each x in a by $S(x)$, and placing the result in canonical form. The extensions $S[a = b]$ and $S[\Gamma]$ of substitution to equalities and to a set of literals Γ are defined in the obvious manner. Let $\text{dom}(S)$ represent $\{x \mid S(x) \neq x\}$, and $\text{vars}(S)$ represent the set of variables in S . The *fusion* $S_1 \triangleright S_2$ of two solution sets S_1 and S_2 is $\{x = S_2[e] \mid (x = e) \in S_1\}$. Such

a fusion is a solution set when $\text{vars}(S_2[e]) \cap \text{dom}(S_1) = \emptyset$ for each $(x = e) \in S_1$. Our use of fusion is restricted to situations where the result can be shown to be a solution set. The *composition* $S_1 \circ S_2$ of two domain-disjoint solution sets S_1 and S_2 is just $(S_1 \triangleright S_2) \cup S_2$. As with fusion, we use the composition operation only when it yields a solution set as the result.

Proposition 1.

- $\{x = e\} \simeq \{a = b\}$ for $x \in \text{vars}(a = b)$ and $(x = e) \equiv \text{solve}(x)(a = b) \neq \perp$.
- $S_1 \cup S_2 \simeq S_1 \circ S_2$, for domain-disjoint solution sets S_1 and S_2 .
- $S[a] \equiv S[b]$ iff $\models S \Rightarrow a = b$, for solution set S .
- $S[S[a]] \equiv S[a]$, for solution set S .

The simplex procedure preprocesses inequalities $a \geq 0$ into equalities $w = a$, where w is a fresh *slack variable* that is interpreted over nonnegative rationals. Equalities $e = 0$ are similarly preprocessed into the form $z = e$, where z is a *zero-slack variable* and interpreted over the single value 0. The zero-slack variables cannot be simply replaced by 0 in our algorithms, since this would result in the loss of proof information. We partition the set of variables into the set X of ordinary *unrestricted* variables that range over the rational numbers, and the set V of *restricted* or *slack* variables that range over the *non-negative* rational numbers. The set of zero-slack variables is written as V_0 , where $V_0 \subseteq V$. The metavariables x and y range over the unrestricted variables, u, v, w range over slack variables, and z ranges over zero-slack variables. An expression a in canonical form is *restricted* if it contains only slack variables. A restricted expression a is *maximized* at $|a|$ if $a^+ \equiv 0$, that is, $\text{vars}(a^+) \subseteq V_0$. Similarly, it is *minimized* at $|e|$ if $e^- \equiv 0$. For example, $3+z_1-2*v_2-3*v_3$ is maximized at 3, and $3-2*z_1-3*z_2+2*v_1+5*v_2$ is minimized at 3.

A *simplex tableau* T is a solution set which contains no unrestricted variables. A tableau T partitions the slack variables into the dependent variables, that is, the left-hand side variables in T , and the independent variables. We typically use the metavariable u to range over the dependent variables, and v to range over the independent variables in the given tableau. A tableau T is *feasible* if $|T(u)| \geq 0$ for all $u \in \text{vars}(T)$.

Given a solution set S , the *basic assignment* ρ_S is a solution set whose domain is $\text{vars}(S) - \text{dom}(S)$ such that $\rho_S(y) = 0$ for each independent variable y in S . The *completion* of an assignment ρ with respect to a solution set S is given by ρ/S and defined as $\rho/S(x) = \rho(S(x))$ for $x \in \text{dom}(S)$, and $\rho/S(y) = \rho(y)$ for $y \in \text{vars}(S) - \text{dom}(S)$. An assignment ρ satisfies the tableau T if $\rho(v) \geq 0$ for each $v \in \text{vars}(T)$, $\rho(z) = 0$ for each $z \in \text{vars}(T) \cap V_0$, and $\rho(u) = \rho(e)$ for each $u = e \in T$. Given a feasible tableau T , the assignment ρ_T/T satisfies T .

The most significant operation in simplex is that of *pivoting*. A pair of variables (u, v) is a *pivot candidate* in T if u is a dependent variable and v is an independent variable in T . Pivoting when applied to a tableau T and pivot candidate (u, v) returns the tableau obtained by exchanging u and v as follows.

$$\text{pivot}(T)(u, v) = (T - \{u = T(u)\}) \circ \text{solve}(v)(u = T(u))$$

For example, for the feasible tableau T_0 of the form $\{u_1 = 1 + 2 * v_1 - 3 * v_2, u_2 = 3*v_1-2*v_2\}$, $\text{pivot}(T_0)(u_1, v_2)$ yields the non-feasible tableau $\{v_2 = \frac{1}{3} + \frac{2}{3}*v_1 - \frac{1}{3}*u_1, u_2 =$

$-\frac{2}{3} + \frac{5}{3} * v_1 + \frac{2}{3} * u_1$ }. As shown below, the simplex algorithms rules out such pivoting steps that do not preserve feasibility. An independent variable v is *unbounded* in tableau T if $\kappa(T(u), v) \geq 0$ for all $u \in \text{dom}(T)$. A restricted expression a is unbounded in T if there is a variable $v \in \text{vars}(a^+) - V_0$ that is unbounded in T . An independent variable v is *bounded* in T if it is not unbounded in T . For example, v_1 is unbounded in the tableau T_0 , and v_2 is bounded.

Given a feasible tableau T , and pivot candidate (u, v) , the *gain* $g(T, u, v)$ of (u, v) in T is defined as

$$g(T, u, v) = -|T(u)|/\kappa(T(u), v), \text{ for } \kappa(T(u), v) < 0 \quad (1)$$

Note that $g(T, u, v) \geq 0$. The pivot candidate (u, v) is *pivotable* in T , formally $\text{pivotable}(T)(u, v)$, if v is not a zero-slack variable, $\kappa(T(u), v) < 0$, and for all pivot candidates (u', v') , the triple $\langle g(T, u, v), v, u \rangle$ is lexicographically smaller than $\langle g(T, u', v'), v', u' \rangle$. This lexicographic ordering on the gain, and the variable ordering on the independent and dependent variables in the pivot candidate captures Bland's rule for preventing pivoting cycles [9] to ensure that any pivoting sequence on a tableau converges to a tableau where no further pivots are applicable. For example, in tableau T_0 above $\text{pivotable}(T_0)(u_2, v_2)$ holds, but $\text{pivotable}(T_0)(u_1, v_2)$ does not. If T is a feasible tableau and v is bounded in T , then there is always a dependent variable u such that $\text{pivotable}(T)(u, v)$.

Proposition 2. *If tableau T is feasible with $\text{pivotable}(T)(u, v)$, then $\text{pivot}(T)(u, v)$ is feasible with $T \simeq \text{pivot}(T)(u, v)$.*

Given a feasible tableau T and $v \notin \text{dom}(T)$, we define the *feasible gain* for v in T as

$$G(T, v) = \begin{cases} \infty, & \text{if } v \text{ unbounded in } T \\ g(T, u, v), & \text{for } u : \text{pivotable}(T)(u, v), \text{ otherwise.} \end{cases}$$

The feasible gain is always non-negative.

Proposition 3. *If T is a feasible tableau, $v \notin \text{dom}(T)$, $K \leq G(T, v)$ for some non-negative rational number K , and ρ_T is a basic assignment for T , then $\rho_T\{v \leftarrow K\}/T$ is a satisfying assignment for T .*

Proposition 4. *For a restricted expression a with $v \in \text{vars}(a^+)$ and a feasible tableau T and $T' = \text{pivot}(T)(u, v)$: $|T'[a]| \geq |a|$. In particular, $|T'[a]| - |a| = \kappa(a, v) * G(T, v)$.*

We now give a method for maximizing a restricted expression a with respect to a feasible tableau T . The operation $\text{max}(a, T)$ computes a pair (a', T') according to the definition below.

$$\text{max}(a, T) = \begin{cases} (a, T), & \text{if } a \text{ unbounded in } T \\ \text{max}(T'[a], T'), & \text{otherwise, where } \text{pivotable}(T)(u, v), \\ & \kappa(a, v) > 0, \text{ and } T' = \text{pivot}(T)(u, v) \end{cases} \quad (2)$$

$$\text{max0}(a, T) = \begin{cases} (a, T), & \text{if } a \text{ unbounded in } T \\ (a, T), & \text{if } |a| > 0 \\ \text{max0}(T'[a], T'), & \text{otherwise, where } \text{pivotable}(T)(u, v), \\ & \kappa(a, v) > 0, \text{ and } T' = \text{pivot}(T)(u, v) \end{cases} \quad (3)$$

The max and max0 operations are terminating as defined since we have ruled out pivoting cycles.

Proposition 5. Given a feasible tableau T and a restricted expression a such that $T[a] \equiv a$, let $\text{max}(a; T) = (a', T')$; then T' is feasible, $T \simeq T'$, $\models T \Rightarrow a = a'$, and a' is either unbounded in T' or is maximized. Similarly, for $\text{max0}(a; T) = (a', T')$, we have $T \simeq T'$, $\models T \Rightarrow a = a'$, and a is either unbounded, maximized at or below 0, or $|a| > 0$.

Example 1. In the tableau T of the form $\{u_1 = 1 + v_1 - 2 * v_2, u_2 = 3 - 2 * v_1 + 2 * v_2\}$ the computation of $\text{max}(-1 + v_1 - v_2; T)$ returns the maximized expression $1/2 - (1/3) * u_2$ together with tableau $\{u_1 = 5/2 - (1/3) * u_2 - v_2, v_1 = (3/2) - (1/3) * u_2 + v_2\}$.

A *zero-tableau* Z is either empty or it is a sequence of the form $Z', w = e$ where Z' is recursively a zero-tableau and $|e| = 0$ and $\text{vars}(e^+) \subseteq \text{vars}(Z') \cup V_0$. For example, $\{w_0 = -v_0, w_1 = z + v_0 - v_1, w_2 = z + v_1 - v_2\}$ is a zero-tableau. In a zero-tableau Z , all the variables in $\text{vars}(Z)$ are maximized at 0 and hence there are no unbounded variables. If T is a feasible tableau with $\text{dom}(T) \cap \text{vars}(Z) = \emptyset$, then $T; Z$ is also a feasible tableau. A few definitions have to be modified to take into account the zero-tableau. In particular $\text{pivotable}(T; Z)(u, v)$ only holds when u and v do not appear in $\text{vars}(Z)$. Also equivalence, maximization, and minimization can be defined modulo Z so that $a \equiv 0(Z)$ holds when $\text{vars}(a) \subseteq \text{vars}(Z) \cup V_0$, and a is maximized (respectively, minimized) to $|a|(Z)$ if $a^+ \equiv 0(Z)$ (respectively, $a^- \equiv 0(Z)$).

3 Online Processing of Linear Constraints

We describe an online procedure for processing constraints $a \geq 0$, $a = 0$, and $a \neq 0$. The procedure is required to detect unsatisfiability and propagate equalities. We also identify a crucial invariant that is preserved by the simplex algorithm and employed in proof generation. The decision procedure is presented as an *inference system* $\langle \Psi, \vdash \rangle$ consisting of a set of configurations Ψ and a binary inference relation \vdash between configurations. In the inference systems we consider, each configuration is just a constraint. A configuration ψ is *irreducible* if there is no ψ' such that $\psi \vdash \psi'$. An inference system is *progressive* if there are no infinite chains of the form $\psi_0 \vdash \psi_1 \vdash \dots$, for any initial configuration ψ_0 . An inference system is *canonizing* when any non- \perp irreducible configuration ψ is satisfiable. We write $\psi \simeq \psi'$ when any model of ψ can be extended to one for ψ' , and vice-versa. An inference relation \vdash is *conservative* if whenever $\psi \vdash \psi'$, then $\psi \simeq \psi'$. A progressive, conservative, and canonizing inference system provides a *decision procedure* for satisfiability. An inference relation is *deterministic* when $\psi \vdash \psi'$ and $\psi \vdash \psi''$ implies $\psi' \equiv \psi''$ for all ψ . If f is a deterministic inference system, then $f(\psi)$ represents the irreducible configuration ψ^* such that $\psi \vdash^* \psi^*$.

Configurations. The simplex procedure is presented in the form of an *inference system* over configurations of the form $\Gamma; R; T; Z; D; W$.

- Γ is a finite set of *input constraints* of the form $a \geq 0$, $a = 0$, and $a \neq 0$ for unrestricted a .
- R is a solution set such that the variables in $\text{dom}(R) \subseteq X$.
- T is a feasible simplex tableau.

AddIneqR	$\frac{a \geq 0, \Gamma; R; T; Z; D; W}{\Gamma; R \circ \text{solve}(y)(w = a'); T; Z; D; w = a, W}$	for $a' = S[a]$, $y \in \text{vars}(a') \cap X$ $w \notin \text{dom}(W)$
AddIneqT	$\frac{a \geq 0, \Gamma; R; T; Z; D; W}{\Gamma; R \triangleright (T'; Z'); T'; Z'; D; W'}$	if $W' = w = a, W$ for $w \notin \text{dom}(W)$ $T'; Z' = \text{addineq}(w = S[a]; T; Z; W')$
DeleteEq	$\frac{a = 0, \Gamma; R; T; Z; D; W}{\Gamma; R; T; Z; D; W}$	if $S[a] \equiv 0(Z)$
AddEqR	$\frac{a = 0, \Gamma; R; T; Z; D; W}{\Gamma; R \circ \text{solve}(y)(S[a] = 0); T; D; W; Z}$	if $y \in \text{vars}(S[a]) \cap X$
AddEqT	$\frac{a = 0, \Gamma; R; T; Z; D; W}{\Gamma; R \triangleright (T'; Z'); T'; Z'; D; W'}$	if $\text{vars}(S[a]) \cap X = \emptyset$ $a' = \begin{cases} -S[a], & \text{if } S[a] > 0, \\ S[a], & \text{otherwise} \end{cases}$ $W' = z = a', W$ for $z \notin \text{dom}(W)$ $T'; Z' = \text{addeq}(z = a'; T; Z; W')$
AddDisEq	$\frac{a \neq 0, \Gamma; R; T; Z; D; W}{\Gamma; R; T; Z; a \neq 0, D; W}$	
Contrad	$\frac{\Gamma; R; T; Z; a \neq 0, D; W}{\perp}$	if $S[a] \equiv 0(Z)$

Fig. 1. The *process* inference system for online constraint processing

- Z is a zero-tableau, and $\text{dom}(T) \cap \text{vars}(Z) = \emptyset$.
- D is a finite set of disequalities of the form $a \neq 0$ with unrestricted a .
- W is a *slack map*, a solution set with entries of the form $w = a$, where w is a slack variable introduced for an input constraint $a \geq 0$ or $a = 0$ so that a is an unrestricted expression and $\text{vars}(T; Z) \subseteq \text{dom}(W)$ and $\text{vars}(R) \cap V \subseteq \text{dom}(W)$. A configuration satisfies a crucial invariant (*input satisfiability*) which asserts that each entry $l = e$ in $R; T; Z$ is such that $W(l)$ and $W[e]$ are identical as canonical expressions, that is, W satisfies $R; T; Z$.

Since the only steps in the algorithm are the introduction of new slack variables, solving existing equalities in the configuration, and composing solution sets, the preservation of the *input satisfiability* invariant follows from the propositions below.

Proposition 6. *If S is a solution set, and W is a slack map satisfying S , then*

1. $W[S[e]] \equiv e$ for any unrestricted term e .
2. For any solution set S' such that W also satisfies S' , W satisfies $S \circ S'$ provided S and S' have non-overlapping domains.

Proposition 7. *If W is a slack map satisfying $a = b$, then for any variable x in $\text{vars}(a = b)$, W also satisfies $\text{solve}(x)(a = b)$*

Inference Rules. The inference rules are presented in Figure 1. In processing $a \geq 0$ or $a = 0$, the first step is to compute a' as $S[a]$. Then $a \geq 0$ is processed as $w = a'$, and $a = 0$ as $z = a'$ for fresh slack variables w or z . When a' contains an unrestricted

variable y , then the rules **AddIneqR** and **AddEqR** solve for y and add the resulting equation to R . When a' is a restricted expression, the auxiliary (deterministic) inference system *addineq* is invoked by the rule **AddIneqT** to add $w = a'$ to T . Similarly, the **AddEqT** rule uses the inference system *addeq* to add $z = a'$ to T . The **AddDiseq** rule simply moves an input disequality $a \neq 0$ to D . The **Contrad** rule checks if $S[a] \equiv 0(Z)$ for some $a \neq 0$ in D to report a contradiction.

Correctness. Given a configuration $\Gamma; R; T; Z; D; W$ of *process*, we abbreviate $R; T; Z$ as S and clearly, ρ_S/S satisfies S .

Theorem 1. *The process procedure is progressive, conservative, and canonizing.*

Proof. Propositions 8 and 9 assert that *addineq* and *addeq* are progressive, conservative, and canonizing. The *process* inference system in Figure 1 is then progressive since it consumes an input formula in each inference step. This inference system is also conservative since the auxiliary systems are conservative, as are the steps introducing slack variables, executing solving, application, and composition (by Proposition 1). The assignment ρ_S/S satisfies S but might not satisfy D . The model construction in Section 7 proves the canonicity of *process* as asserted in Proposition 13. ■

Example 2. The *process* inference procedure when applied to the input constraint $x \geq 0, y \geq 0, z \geq 0, x + y \geq 2$ and $z - y \leq -3$ yields $R; T; Z; D; W$, where $R = \{x = w_0, y = 3 + w_2 + w_4, z = w_2\}$, $T = \{w_1 = 3 + w_2 + w_4, w_3 = 1 + w_0 + w_2 + w_4\}$, $Z = \emptyset$, $D = \emptyset$, and $W = \{w_0 = x, w_1 = y, w_2 = z, w_3 = -2 + x + y, w_4 = -3 + y - z\}$. Then, it is easy to check input satisfiability. For example, $W(w_1) \equiv y \equiv W[3 + w_4 + w_2]$ and $W(w_3) = -2 + x + y = W[1 + w_0 + w_2 + w_4]$.

4 Processing Inequalities

Given a feasible simplex tableau T , a zero-tableau Z , a slack map W for $T; Z$, a restricted expression a , and a slack variable w , $w \notin \text{vars}(T; Z) \cup \text{vars}(a)$, the addition of $w = a$ to T is carried out by the inference system in Figure 2. The inference configuration is either \perp or of the form $w = a; T; Z; W$ or $T; Z; W$. We assume that a is canonical with respect to $T; Z$, that is, $(T; Z)[a] \equiv a$. If a is maximized below 0, a contradiction is reported by the **Maximize** rule. If a is minimized at or above $0(Z)$, then the inequality $a \geq 0$ is already implied by $T; Z$ and is therefore redundant and is discarded by the **Minimize** rule. If $|a| \geq 0$, then $w = a$ can be safely composed with T while preserving feasibility. There are two cases according to whether a is maximized at 0. If it is, and $a', T' = \text{max}0(a, T; Z)$, then zero-propagation is invoked on $T'; Z$ using *incZeroes*($T'; Z$)($w = a'$) to turn implicit equalities into explicit ones as described in Section 6. Otherwise, as we argue in Section 6, no new implicit equalities are introduced, and $w = a$ can be composed with T . If $|a| < 0$ and a is unbounded in $T; Z$, then a^+ contains an occurrence of an independent variable v that is unbounded in $T; Z$. We can solve $w = a$ for v to obtain $v = a'$ which can be composed with T since $|a'| > 0$. This case also does not generate any new implicit equalities. In the remaining case, when $|a| < 0$, a is neither unbounded, maximized below $0(Z)$, nor minimized at or above $0(Z)$, then we pick $v \in \text{vars}(a^+)$ and u such that *pivotable*($T; Z$)(u, v) and let T' be *pivot*(T)(u, v). As noted in Proposition 4, $|T'[a]| \geq |a|$. The pivoting step is repeated until either a

Maximize	$\frac{w = a; T; Z; W}{\perp}$	if a is maximized at $ a (Z)$, $ a < 0$
Minimize	$\frac{w = a; T; Z; W}{T; Z; W}$	if a is minimized at $ a (Z)$, $ a \geq 0$
FeasibleZero	$\frac{w = a; T; Z; W}{T''; Z''; W}$	$ a \geq 0$ a is not minimized at $0(Z)$ $a', T' = \max 0(a, T; Z)$ a' is maximized at $0(Z)$ $T''; Z'' = \text{incZeroes}(T'; Z)(w = a')$
FeasibleNonzero	$\frac{w = a; T; Z; W}{T \circ \{w = a\}; Z; W}$	$ a \geq 0$ $a', T' = \max 0(a, T; Z)$ $ a' > 0$
Unbounded	$\frac{w = a; T; Z; W}{T \circ \text{solve}(v)(w = a); Z; W}$	if a is unbounded in $T; Z$, $ a < 0$, $v \in \text{vars}(a^+)$ v unbounded in $T; Z$
Pivot	$\frac{w = a; T; Z; W}{w = T'[a]; T'; Z; W}$	if $ a < 0$, a not unbounded in $T; Z$ $\text{pivotable}(T; Z)(u, v)$ $T' = \text{pivot}(T)(u, v)$

Fig. 2. The *addineq* inference system.

is unbounded, maximized below $0(Z)$, minimized at or above $0(Z)$, or $|a| \geq 0$. The inference system is progressive because of the selection strategy of pivotable variable pairs based on Bland's rule [9].

Proposition 8. *The addineq procedure is progressive, conservative, and canonizing.*

5 Adding Restricted Equalities

We now describe the *addeq* procedure used in Figure 1. It is invoked on a configuration $z = e; T; Z; W$, where e is restricted and $(T; Z)[e] \equiv e$. The inference rules are displayed in Figure 3. The point of this procedure is turn the zero-slack variable z into an independent variable before composing the new equality with T . If e is maximized below $0(Z)$, then a contradiction is reported by the **EqMax** rule. If $|e| < 0$ and e^+ contains a variable $v \notin \text{vars}(Z) \cup V_0$ such that $G(T; Z, v) \geq -|e|/\kappa(e, v)$, then we solve $z = e$ for v to obtain $v = e'$ which is composed with T to yield T' (**EqSwap**). In this case, v can also be unbounded in $T; Z$. Since $G(T; Z, v) \geq |e'| \geq 0$, the composition $T \circ \{v = e'\}$ yields a feasible tableau. For example, if we are adding $z = -1 + v_1 - v_2$ to the tableau T containing $u = 1 - v_1$, then one application of **EqSwap** yields $v_1 = 1 + z + v_2$ which can be composed with T to yield $u = 0 - z - v_2$, $v_1 = 1 + z + v_2$ for T' . Note that u and v_2 are maximized to 0 in T' . Rule **Eq0** handles the case $|e| = 0$, then we can solve $z = e$ for v for some $v \in \text{vars}(e)$ and compose the resulting solution $v = e'$ with T to obtain T' . The solution set T' is still feasible since $|e'| = 0$. If $|e| < 0$, then e^+ must contain a variable v since e is not maximized below 0. We find a u such that $\text{pivotable}(T; Z)(u, v)$ and perform the pivot to obtain $T' = \text{pivot}(T)(u, v)$ (**EqPivotUp**). Note that $|T'[e]| \geq |e|$.

EqMax	$\frac{z = e; T; Z; W}{\perp}$	if e is maximized below $0(Z)$
EqSwap	$\frac{z = e; T; Z; W}{T'; Z'; W}$	if $v \in \text{vars}(e^+) - (\text{vars}(Z) \cup V_0)$ $ e < 0, - e /\kappa(e, v) \leq G(T; Z, v)$ $(v = e') = \text{solve}(v)(z = e)$ $T'; Z' = \text{incZeroes}(T; Z)(v = e')$
EqDelete	$\frac{z = e; T; Z; W}{T; Z; W}$	if $e \equiv 0(Z)$
Eq0	$\frac{z = e; T; Z; W}{T'; Z'; W}$	if $ e = 0,$ $v \in \text{vars}(e) - (\text{vars}(Z) \cup V_0)$ $(v = e') = \text{solve}(v)(z = e)$ $T'; Z' = \text{incZeroes}(T; Z)(v = e')$
EqPivotUp	$\frac{z = e; T; Z; W}{z = T'[e]; T'; Z; W}$	if e not unbounded in $T; Z,$ $ e < 0, v \in \text{vars}(e^+)$ $\text{pivotable}(T; Z)(u, v)$ $G(T, v) \leq - e /\kappa(e, v)$ $T' = \text{pivot}(T)(u, v)$

Fig. 3. The *addeq* inference system.

Each inference step in *addeq* either involves a *solve* or a *pivot* operation which preserves solutions as well as the feasibility of the tableau and the invariants on a valid *process* configuration.

Proposition 9. *The addeq procedure is progressive, conservative, and canonizing.*

6 Detecting and Propagating Implicit Equalities

The zero-tableau Z serves to collect those slack variables that are explicitly equal to 0. A variable v is explicitly equal to 0 in $T; Z$ if it is a zero-slack or $v \in \text{vars}(Z)$. A slack variable v is implicitly equal to 0 in $T; Z$ if $v \notin \text{vars}(Z) \cup V_0$ and v is maximized to $0(Z)$ in $T; Z$. The set of variables that are implicitly equal to 0 in T are called the *zero-variables* of T . When there are no zero-variables in T , the pair $T; Z$ is said to be 0-closed. If $T; Z$ is 0-closed, then an expression e can be simplified as $(T; Z)[e]$ to yield e' , and e' can be maximized to e'' in $T; Z$. If $e'' \neq 0(Z)$, then by Proposition 3, we can find an assignment ρ satisfying $T; Z$ such that $\rho(e) \neq 0$.

Proposition 10. *If $T; Z$ is 0-closed, then $T; Z \Rightarrow e = 0$ iff $(T; Z)[e] \equiv 0(Z)$.*

Two tableaux T_1 and T_2 are said to be *0-equivalent*, in symbols $T_1 \simeq_0 T_2(Z)$, when for each $v \in \text{vars}(T) - V_0$, $e_1, T'_1 = \text{max}0(v, T_1; Z)$ and $e_2, T'_2 = \text{max}0(v, T_2; Z)$, e_1 is maximized to $0(Z)$ iff e_2 is maximized to $0(Z)$. A transformation from T to T' is said to be zero-preserving when $T \simeq_0 T'(Z)$. The detection of zero-variables in T involves such zero-preserving transformations. Define $\bar{T} = \{u = e \in T \mid |e| = 0\}$. Then $T \simeq_0 \bar{T}(Z)$ since if $e', T' = \text{max}0(v, T; Z)$ such that $|e'| \equiv 0(Z)$, then the maximization must not have involved any pivoting steps using the entries in $T - \bar{T}$. Let $\text{bounded}(T; Z)$ represent the

tableau $\{u = e \in T \mid e \text{ is not unbounded in } T; Z\}$. For example, if \bar{T} is $\{u_2 = v_1 - v_2, u_3 = v_2 - v_3, u_4 = v_3 - v_2\}$ and Z is empty, then $\text{bounded}(\bar{T}; Z)$ is $\{u_3 = v_2 - v_3, u_4 = v_3 - v_2\}$ since v_1 is unbounded in \bar{T} . Then $\bar{T} \simeq_0 \text{bounded}(\bar{T}; Z)$ (Z) again because if $e', T' = \text{max}0(v, \bar{T}; Z)$ such that $|e'| \equiv 0(Z)$, then the maximization must not have employed any pivoting steps on entries in $\bar{T} - \text{bounded}(\bar{T})$ since this would have yielded an e' that is unbounded in $T; Z$.

From the tableau T , we can construct the 0-equivalent tableau $\text{bounded}^*(\bar{T})$ without any pivoting steps. We abbreviate $\text{bounded}^*(\bar{T})$ by T^b , and its complement $T - T^b$, by T^\sharp . The mutually recursive operations *FindZeroes* and *maxentries* are used to compute a 0-closed $T'; Z'$ from $T; Z$ such that $T'; Z' \simeq T; Z$.

$$\text{FindZeroes}(\emptyset; Z) = \emptyset; Z$$

$$\text{FindZeroes}(T; Z) = T''; Z', \text{ where}$$

$$T'; Z' = \text{maxentries}(T^b; Z)$$

$$T'' = T^\sharp \cup T'$$

$$\text{maxentries}(\emptyset; Z) = \emptyset; Z$$

$$\text{maxentries}(w = e, T; Z) = \text{FindZeroes}(u = e', T''; Z''), \text{ where}$$

$$e', T' = \text{max}0(e, T; Z)$$

$$T''; Z'' = \begin{cases} u = e', T'; Z, & \text{if } e' \text{ is unbounded in } T'; Z \\ T'; Z, u = e' & \text{otherwise.} \end{cases}$$

The *FindZeroes* operation always terminates since the number of unbounded entries in T decreases with each recursive call. For example, if T is $\{u_3 = v_2 - v_3, u_4 = v_3 - v_2\}$, then $v_2 - v_3$ is maximized to $-1 * u_4$, and hence u_3 and u_4 are found to be equal to 0, while T' becomes $\{v_2 = v_3\}$. Since v_3 is unbounded, the result of $\text{FindZeroes}(T; \emptyset)$ is $\{v_2 = v_3\}; \{u_3 = -1 * u_4\}$.

Proposition 11. *If $T'; Z' = \text{FindZeroes}(T; Z)$ for zero-tableau Z and T such that $|T(u)| = 0$ for $u \in \text{dom}(T)$, then Z' is also a zero-tableau, $T'; Z' \simeq T; Z$, and $T^b = \emptyset$. In other words, $T'; Z'$ is 0-closed.*

We now introduce incremental zero propagation to the *addineq* and *addeq* procedures. When a new entry $w = a$ is added to a 0-closed tableau $T; Z$ in the *addineq* procedure in Figure 2, the rules that affect the simplex tableau are **Pivot**, **Unbounded**, **FeasibleZero**, and **FeasibleNonzero**. The **Unbounded** rule does not introduce any new zero-variables into the tableau since we solve for an unbounded variable v to obtain $v = a'$ with $|a'| > 0$, which is then composed with the tableau. Since v occurs only positively in the tableau, no new zero-variable candidates are introduced. When pivoting the tableau T to obtain T' , we have $T \simeq T'$, and since we have already analyzed T for zero-variables, no new ones are introduced in T' . If a' is either positive or unbounded in T' , then we simply add $u = a'$ to T' using **FeasibleNonzero** and no further analysis is needed. That leaves the **FeasibleZero** rule as the only source of new zero-variables. This can only happen when $w = a$ is composed with T and $|a| = 0$. Similarly, in *addeq*, the **EqPivotUp** step does not introduce any new implicit zero-variables in T , so the only rules that trigger zero propagation are **EqSwap** and **Eq0**.

When a new entry $u = e$ is composed with T to obtain T' , any affected entries $u' = e'$ must also be analyzed to see if they generate zeroes. We do not need to examine all of $\overline{T'}$ for this purpose but only the transitive closure of the affected entries as defined below. Define $\text{vars}^-(T) = \{v \mid v \in \text{vars}(e^-) \text{ for } u = e \in T\}$.

$$\begin{aligned} \text{posvar}(T; Z)(X) &= \{u = e \in T \mid |e| = 0 \wedge (\text{vars}(e^+) - \text{vars}(Z)) \cap X \neq \emptyset\} \\ T; Z/T' &= \text{posvar}(T; Z)(\text{vars}^-(T')) \\ \text{incbounded}(T; Z)(T') &= \begin{cases} \text{incbounded}(T; Z)(T' \cup T; Z/T') & \text{if } T; Z/T' \neq \emptyset \\ T' & \text{otherwise} \end{cases} \end{aligned}$$

In other words, we are only interested in those entries that contain positive occurrences of a variable v that is being set to 0, and transitively, the entries thus influenced.

Lemma 1. *If T is 0-closed and $w = a$ is such that $T[a] = a$, $|a| = 0$, and $w \notin \text{vars}(T)$, then for any entry $u = e \in T - \text{incbounded}(T)(w = a)$ if e' ; $T' = \text{max}0(e; T)$, then e' is not maximized at 0.*

Proof. Let $T'' = \text{incbounded}(T)(w = a)$, we know that for any positive variable v in e , that v does not occur negatively in T'' . Hence, all the pivoting steps in the computation of $\text{max}0(e; T)$ occur within the entries in $T - T''$ and hence within T itself. Since T is 0-closed, we know that e is not maximized to 0 in T . ■

We can now use *incbounded* and *FindZeroes* to define a single operation for incremental zero propagation when adding the entry $u = e$ to the tableau $T; Z$.

$$\begin{aligned} \text{incZeroes}(T; Z)(u = e) &= T'''; \hat{Z}, \text{ where} \\ T' &= T \circ u = e, \\ T'' &= \text{incbounded}(T'; Z)(\overline{T' - T}), \\ T^+ &= T - T'', \\ \hat{T}; \hat{Z} &= \text{FindZeroes}(T''; Z) \\ T''' &= (T^+ \circ \hat{T}) \triangleright \hat{Z} \end{aligned}$$

By Proposition 11 and Lemma 1, we have the following proposition.

Proposition 12. *If $T, u = e$ is a feasible tableau, Z is a zero-tableau, $T; Z$ is 0-closed and T' ; $Z' = \text{incZeroes}(T; Z)(u = e)$, then Z' is a zero-tableau, T' is a feasible tableau, $T'; Z'$ is also 0-closed, and $T'; Z' \simeq T; Z$.*

Example 3. Let T be of the form $\{u_0 = v_2 - v_3, u_1 = v_2 - v_4, u_2 = v_1 - v_2, u_3 = v_5 - v_1\}$ and Z be empty. If we add the equality $w = -v_1 + v_3$ to T , then the **FeasibleZero** rule is applicable. The operation $\text{max}0(-v_1 + v_3; T)$ yields e' of the form $-u_0 - u_2$ and T' of the form $\{v_3 = v_1 - u_0 - u_2; u_1 = v_1 - v_4 - u_2; v_2 = v_1 - u_2; u_3 = v_5 - v_1\}$. Now, $\text{incZeroes}(T')(w = e')$ yields $T'; \{w = -u_0 - u_2\}$ since T' does not have any positive occurrences of the variables u_0 and u_2 .

7 Model Generation

Given an irreducible state $S; D$ for the *process* inference system, the assignment ρ_S/S satisfies S , but may violate some of the disequalities in D . For model generation in the

presence of disequalities, we use a total ordering on variables such that the unrestricted variables precede the restricted ones. First, we compute D' as $S[D]$. Since $S; D$ is irreducible, and particular, the **Contrad** rule is inapplicable, it must be the case that for each disequality $e \neq 0 \in D'$, $e \neq 0(Z)$. Then we rearrange D' to get $sld(D')$ in which each disequality $e \neq 0$ from D' is placed in a solved form $x \neq e'$, where x is the minimal variable in $vars(e) - (vars(Z) \cup V_0)$. Given an assignment ρ , let $\rho_{>v}$ be the solution set $\{v' = \rho(v') \mid v' > v\}$. For $a', T' = \max(v, T)$ let $M(v, T, \rho)$ be defined to be $|a'|$ if a' is bounded, and ∞ otherwise, where $a'' = \rho_{>v}[a']$. Define $Q(x, D, \rho)$ as $\{\rho(e') \mid x \neq e' \in D\}$. Starting with $\rho = \rho_S$ and D' , a model is constructed by repeating the following steps. Since D might contain variables not occurring in S , we assume that $\rho_S(x) = 0$ for such variables.

$$\begin{aligned}
model(S, D) &= mdl(S, sld(S[D]), vars(S[D]) - (vars(Z) \cup V_0), \rho_S) \\
mdl(S, D, \emptyset, \rho) &= \rho \\
mdl(S, D, Y, \rho) &= mdl(S, D, Y - \{x\}, \rho'), \text{ where} \\
&\quad \text{unrestricted } x \text{ is maximal in } Y \\
&\quad \rho' = \rho\{x \leftarrow q\} \text{ for } q \notin Q(x, D, \rho) \\
mdl(S, D, Y, \rho) &= mdl(D, Y - \{x\}, \rho'), \text{ where} \\
&\quad \text{restricted } v \text{ is maximal in } Y \\
&\quad S = R; Z; T \\
&\quad \rho' = \rho\{x \leftarrow q\} \text{ for } 0 < q < M(v, T, \rho), q \notin Q(v, D, \rho)
\end{aligned}$$

Since every configuration in the *process* inference system is 0-closed, by Proposition 12, we know that since $v \notin vars(Z) \cup V_0$, it can consistently take on any value between 0 and $M(v, T, \rho)$. In the definition of $mdl(S, D, Y, \rho)$, the set Y contains the yet unassigned variables in $vars(S[D]) - vars(Z)$. We can check that whenever $mdl(S, D, Y, \rho)$ is invoked, then ρ/S satisfies S , $S[D] \equiv D$, $M(v, T, \rho) > 0$ for any $v \in Y$, and for any disequality $x \neq e$ in D such that $vars(x \neq e) \cap Y \subseteq vars(Z) \cup V_0$, ρ satisfies $x \neq e$, then for $\rho' = mdl(S, D, Y, \rho_S)$, ρ'/S satisfies $S; D$. Hence, $model(S, D)$ satisfies $S; D$.

Proposition 13. *The process inference system is canonizing.*

8 Proof Generation

Proof construction can be used to justify the validity of the contradictions generated by the rules **Contrad**, **Maximize**, and **EqMax**. The proofs depend only on the inputs corresponding to the slack variables that actually appear in principal literals in the premise of these rules. For example, in **Contrad**, only the input literals corresponding to the slack variables in $S[a]$ are relevant that for any variable. For proof construction, we need the proof rules shown in Figure 4. The basic proof constructors are defined in terms of those given by the proof rules.

Definition 1.

$$zeroproof(Z, u = e; W)(v) = zeroproof(Z; W)(v), \text{ if } v \in vars(Z)$$

Axiom	$\frac{}{\Gamma \vdash e \geq 0} e \geq 0 \in \Gamma$	Equal	$\frac{\Gamma_1 \vdash e \geq 0 \quad \Gamma_2 \vdash e \leq 0}{\Gamma_1 \cup \Gamma_2 \vdash e = 0}$
Scale	$\frac{\Gamma \vdash e \geq 0}{\Gamma \vdash k * e \leq 0} k < 0$	ZeroScale	$\frac{\Gamma \vdash v = 0}{\Gamma \vdash k * v = 0}$
IneqSum	$\frac{\Gamma_1 \vdash e_1 \leq 0 \quad \Gamma_2 \vdash e_2 \leq 0}{\Gamma_1 \cup \Gamma_2 \vdash e_1 + e_2 \leq 0}$	ZeroLeq	$\frac{\Gamma \vdash e = 0}{\Gamma \vdash e \leq 0}$
StrictSum	$\frac{\Gamma \vdash e \leq 0}{\Gamma \vdash k + e < 0} k < 0$	ZeroGeq	$\frac{\Gamma_1 \vdash e = 0}{\Gamma_1 \vdash e \geq 0}$
Contrad	$\frac{\Gamma_1 \vdash e \geq 0 \quad \Gamma_2 \vdash e < 0}{\Gamma_1 \cup \Gamma_2 \vdash \perp}$	Diseq	$\frac{\Gamma_1 \vdash e = 0 \quad \Gamma_2 \vdash e \neq 0}{\Gamma_1 \cup \Gamma_2 \vdash \perp}$

Fig. 4. The Proof Rules

$$\begin{aligned}
\text{zeroproof}(Z, u = e; W)(v) &= \text{Equal}(\text{Axiom}(W(u) \geq 0), \text{maxproof}(Z; W)(e)), \\
&\quad \text{if } u \equiv v, v \notin \text{vars}(Z), v \in \text{dom}(W) \\
\text{zeroproof}(Z, u = e; W)(v) &= \text{zeroproof}(Z, v = e'; W)(v), \text{ if } (v = e') = \text{solve}(v)(u = e) \\
\text{maxproof}(Z; W)(k + e) &= \text{StrictSum}(k)(\text{maxproof}(Z; W)(e)), \text{ if } k < 0 \\
\text{maxproof}(Z; W)(k * v + e) &= \text{IneqSum}(\text{maxproof}(Z; W)(k * v), \text{maxproof}(Z; W)(e)) \\
\text{maxproof}(Z; W)(k * v) &= \text{Scale}(k)(\text{Axiom}(W(v) \geq 0)), \text{ if } k < 0, v \in \text{dom}(W) \\
\text{maxproof}(Z; W)(k * v) &= \text{ZeroLeq}(\text{ZeroScale}(k)(\text{zeroproof}(Z; W)(v))), \text{ if } v \in \text{vars}(Z) \\
\text{eqzproof}(Z; W)(e) &= \text{Equal}(\text{maxproof}(Z; W)(e), \text{maxproof}(Z; W)(-e)) \\
\text{ineqproof}(Z; W)(u = e) &= \text{Contrad}(\text{Axiom}(W(u) \geq 0), \text{maxproof}(Z; W)(e)), \text{ if } u \in \text{dom}(W) \\
\text{eqproof}(Z; W)(z = e) &= \text{Contrad}(\text{ZeroGeq}(\text{Axiom}(W(z) = 0)), \text{maxproof}(Z; W)(e)) \\
\text{diseqproof}(R; T; Z; D; W)(e) &= \text{Diseq}(\text{eqzproof}(Z; W)(S[e]), \text{Axiom}(e \neq 0)), \\
&\quad \text{if } S = R; T; Z, S[e] \equiv 0(Z) \text{ for } e \neq 0 \in D
\end{aligned}$$

By the definition of a zero-tableau, we know that if Z is of the form $Z', u = e$ and v does not occur in Z' , then it must either be u or occur negatively in e . If $u \equiv v$, then by the **Axiom** rule, $\Gamma \vdash W(v) \geq 0$. The expression e is of the form $k_1 * v_1 + \dots + k_n * v_n$, where for each i , $1 \leq i \leq n$, either $v_i \in \text{vars}(Z')$ or $k_i < 0$. When $v_i \in \text{vars}(Z')$, we use $\text{zeroproof}(Z'; W)(v_i)$ to construct a proof of $\Gamma \vdash W(v_i) = 0$, and **ZeroScale** to construct a proof of $\Gamma \vdash W[k_i * v_i] = 0$. When $k_i < 0$, then the **Axiom** and **Scale** rules can be used to construct a proof of $W[k_i * v_i] \leq 0$. These proofs of can be combined with **ZeroLeq** and **IneqSum** to derive a proof of $\Gamma \vdash W[e] \leq 0$. Since by input satisfiability $W(v) \equiv W[e]$, we can use **Equal** to derive $\Gamma \vdash W(v) = 0$ from those of $\Gamma \vdash W(v) \geq 0$ and $\Gamma \vdash W(v) \leq 0$. If $u \not\equiv v$, then v occurs negatively in e . Then $\text{solve}(v)(u = e)$ yields $v = e'$, where e' is also maximized at 0 modulo Z' . Note that v does not occur in Z' . The above proof construction can be applied to $v = e'$ to yield a proof of $\Gamma \vdash W(v) = 0$.

Theorem 2. *If W is a slack map, Z is a zero-tableau, $\text{vars}(Z) \subseteq \text{dom}(W)$, $\text{vars}(e) \subseteq \text{dom}(W)$, and $\Gamma = \{W(v) \geq 0 \mid v \in \text{dom}(W)\}$, then*

1. $\text{maxproof}(Z; W)(e)$ proves $\Gamma \vdash W[e] \leq 0$, if e is maximized at $0(Z)$.
2. $\text{maxproof}(Z; W)(e)$ proves $\Gamma \vdash W[e] < 0$, if e is maximized below $0(Z)$.
3. $\text{zeroproof}(Z; W)(v)$ proves $\Gamma \vdash W(v) = 0$.
4. (**Maximize**) $\text{ineqproof}(Z; W)(w = e)$ proves $\Gamma \vdash \perp$, if e is maximized below $0(Z)$.
5. (**EqMax**) $\text{eqproof}(Z; W)(z = e)$ proves $\Gamma \vdash \perp$, if e is maximized below $0(Z)$.
6. (**Contrad**) $\text{diseqproof}(R; T; Z; D; W)(e)$ proves $\Gamma, D \vdash \perp$, if $S = R; T; Z$, $S[e] \equiv 0(Z)$ for $e \neq 0 \in D$.

9 Conclusions

We have presented the details of an incremental constraint solver for linear arithmetic equality, disequality, and inequality constraints based on the simplex algorithm for linear programming. Our procedure builds on a large body of prior work in linear programming. Our algorithm propagates equalities and generates models and proofs. The equality propagation step uses a novel analysis to avoid unwanted pivoting. We have identified a number of crucial properties of the simplex procedure including the input satisfiability invariant. This invariant is used to justify proof generation and is also the key to efficient retraction, which we have omitted for lack of space. Our constraint solver has been implemented in the ICS system available from SRI International where it is part of a combination involving a number of other decision procedures including a SAT solver. Our procedure can also be extended to the case of equalities and inequalities involving integer variables and mixed-integer problems. This is done through a combination of pure and mixed-integer Gomory cuts and case analysis based on the integer disequalities. The generation of efficient explanations and proofs is the key to a number of applications.

References

1. G.J. Badros, A. Borning, and P.J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, December 2001.
2. G.B. Dantzig and B. Curtis. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory*, pages 288–297, 1973.
3. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A theorem prover for program checking. Technical report, Hewlett-Packard Systems Research Center, 2003.
4. P. Van Hentenryck and T. Graf. Standard forms for rational linear arithmetics. *Annals of Mathematics and Artificial Intelligence*, 5:303–320, 1992.
5. G.C. Necula. Compiling with proofs. Technical Report CMU-CS-98-154, School of Computer Science, Carnegie Mellon University, 1998.
6. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca., 1981.
7. G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
8. N. Shankar and H. Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, 2002. Springer Verlag.
9. R.J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer’s International Series, 2001.