[Draft. Please do not distribute] Online Proof-Producing Decision Procedure for Mixed-Integer Linear Arithmetic*

Sergey Berezin, Vijay Ganesh, and David L. Dill

Stanford University
{berezin,vganesh,dill}@stanford.edu

Abstract. Efficient decision procedures for arithmetic play a very important role in formal verification. In practical examples, however, arithmetic constraints are often mixed with constraints from other theories like the theory of arrays, Boolean satisfiability (SAT), bit-vectors, etc. Therefore, decision procedures for arithmetic are especially useful in combination with other decision procedures. The framework for such a combination is implemented at Stanford in the tool called Cooperating Validity Checker (CVC) [SBD02].

This work augments CVC with a decision procedure for the theory of mixed integer linear arithmetic based on the Omega-test [Pug91] extended to be *online* and *proof producing*. These extensions are the most important and challenging part of the work, and are necessary to make the combination efficient in practice.

1 Introduction

Formal verification methods benefit greatly from efficient automatic decision procedures. There has been ample research in developing efficient decision procedures for various satisfiability problems like *Boolean satisfiability* (SAT [MMZ+01,MSS99]), *bit-vectors* [Möl98], *linear integer and real arithmetic*, etc.

In practical examples, constraints from different theories are often mixed together. For example, it is not uncommon to see a constraint like the following:

$$2 * y + x > z \wedge f(x) \neq z \rightarrow A[2 * y + x] > 0,$$

which belongs to the combined theory of arithmetic, arrays, and uninterpreted functions. Consequently, there is a need for a decision procedure for the combination of theories.

It is natural to construct a combined decision procedure out of decision procedures for individual theories. However, such a construction is not a straightforward task. Several combination methods like Nelson-Oppen [NO79], Shostak [Sho84], and their variants [RS01,BDS02a] have been developed in the past. All these methods impose certain

* This research was supported by GSRC contract SA2206-23106PG-2. The content of this paper does not necessarily reflect the position or the policy of GSRC or the Government, and no official endorsement should be inferred.

requirements on the individual decision procedures in order to achieve a sound, complete, and efficient combination. Satisfying these requirements greatly improves the usability of the individual decision procedures.

One of the tools that combine decision procedures is the *Cooperating Validity Checker* (CVC [SBD02]) developed at Stanford. It is based on the framework of cooperating decision procedures developed by Barrett [BDS02a] which, in turn, is based on Nelson-Oppen [NO79] and Shostak [Sho84] frameworks. Each decision procedure in the framework is responsible for solving satisfiability problem for only one particular theory and does not interact directly with the other theories.

The current implementation involves decision procedures for the theory of uninterpreted functions, the theory of arrays [SBDL01], the theory of datatypes, and the theory of linear arithmetic over integers and reals, which is the subject of this paper. Additionally, Boolean combinations of constraints are handled on the top level by the SAT solver [BDS02b] based on Chaff [MMZ$^+$01]. Thus, CVC as a whole serves as a decision procedure for the quantifier-free first-order theory of equality with arrays, recursive datatypes, and linear arithmetic [BDS00].

As with all combination methods, CVC imposes certain requirements on the individual decision procedures. In particular, each decision procedure must be *online* and *proof producing*. Online means that a new constraint can be added to the set of existing constraints at any time, and the algorithm must be able to take it into account with only incremental amount of work. When the set of constraints is determined to be unsatisfiable, the algorithm must also produce a proof of this fact.

Additionally, the theory of linear arithmetic is extended with the predicate $\mathsf{int}(t)$, which evaluates to true on a real-valued arithmetic term t, when t evaluates to an integer value. The decision procedure must be able to handle constraints of the form $\mathsf{int}(t)$ and $\neg\mathsf{int}(t)$ in addition to the usual linear arithmetic constraints.

The reasons for the above requirements can be better understood from the architecture of CVC. At a very high level, CVC can be viewed as a SAT solver [DP60] which solves the satisfiability problem of the Boolean skeleton of the first-order formulas. Each time the SAT solver makes a decision, a new constraint is produced, which is submitted to the appropriate decision procedure. Since decisions are dynamic, decision procedures must be able to receive a new constraint and process it efficiently (the *online* requirement).

Modern SAT solvers [SS96,MSS99,MMZ$^+$01] use *conflict clauses* and *intelligent backtracking* to enhance performance. Backtracking implies that the solver may retract some constraints dynamically, hence, the decision procedure must support this operation. From an implemenation point of view, this means that all the internal data structures in the decision procedure must be backtrackable.

To construct a conflict clause, one needs to identify those decisions made by the SAT solver which lead to a contradiction. One way of identifying such decisions is to extract them from the proof that the decision procedure constructs when it derives a contradiction. This explains the need for proof production in the decision procedures in CVC. As a bonus, the proofs can be checked by an external proof checker [Stu02] to increase the confidence in the results produced by CVC.

Finally, the $\mathsf{int}(t)$ predicate arises from the translation of CVC's logic with partial functions [Far90] to classical logic. However, this topic is out of scope of this paper.

In summary, a decision procedure is much more useful to the research community when it can be combined with other decision procedures. This combination imposes additional requirements on the decision procedure. Namely, it must be *online* and *proof producing*, and all of its internal data structures must be backtrackable.

This paper describes a decision procedure for mixed-integer linear arithmetic designed to meet the above requirements, and thus, fit the CVC framework. The decision procedure is based on an existing algorithm called *Omega-test* [Pug91], which is extended to be online, proof producing, and handle the $\mathsf{int}()$ predicate. Additionally, this implementation supports arbitrary precision arithmetic based on the GMP library [GMP].

The choice of Omega-test over other algorithms for solving mixed-integer linear arithmetic problems (simplex, interior point method [BT97], earlier versions of Fourier-Motzkin elimination [Wil76], etc.) is driven by its simplicity and practical efficiency for a large class of verification problems. In particular, proof production is relatively easy to implement for the Omega-test.

The rest of the paper is organized as follows. Sections 2 and 3 remind the reader the original Fourier-Motzkin elimination method for real variables and its extension to integers (Omega-test), respectively. Both versions are then redesigned to make the algorithm online as described in sections 4 and 5. Section 4 also gives a brief overview of CVC. Proof production is described in section 6, and we conclude in section 7.

2 Fourier-Motzkin Elimination for Inequalities over Real Variables

The problem. Given a system of linear inequality constraints over real-valued variables of the form

$$\sum_{i=1}^{n} a_i x_i + c < 0,$$

where a_i's and c are rational constants, determine if this system is satisfiable. We only consider strict inequalities, since $\alpha \leq 0$ can be expanded into $\alpha < 0 \vee \alpha = 0$ and solved for each case in the disjunction. Here and in the rest of the paper, linear arithmetic terms are often denoted as α, β, γ, or t, possibly with subscripts. In this section, we do not consider equalities, since any equality can be solved for some variable and instantiated into the other constraints, thus obtaining an equivalent system without the equality.

Terminology. For the sake of terminological clarity, we say that a variable is ***eliminated*** if it is eliminated by solving an equality constraint for it and substituting the result in all the other constraints. When the variable is eliminated using the Fourier-Motzkin reasoning on inequalities, we say that such a variable is ***projected***.

Throughout the paper we assume that all the constants and coefficients are *rational*. Although we often refer to variables as real-valued, it is well-known that under the

above conditions, the system of linear constraints is satisfiable in reals if and only if it is satisfiable in rationals.

Intuitively, Fourier-Motzkin elimination procedure [DE73] iteratively projects one variable x by rewriting the system of inequalities into a new system without x which has a solution if and only if the original system has a solution (i.e. the two systems are *equisatisfiable*). This process repeats until no variables are left, at which point all of the constraints become inequalities over numerical constants and can be directly checked for satisfiability.

More formally, the projection procedure is the following. First, pick a variable present in at least one inequality, say x_n. All the inequalities containing this variable are then rewritten in the form $\beta < x_n$ or $x_n < \alpha$, depending on the sign of the coefficient a_n. This creates three types of constraints: those with x on the right, with x on the left, and those without x:

$$\left\{\begin{array}{c}\beta_1 < x_n \\ \vdots \\ \beta_{k_1} < x_n\end{array}\right. \qquad \left\{\begin{array}{c}x_n < \alpha_1 \\ \vdots \\ x_n < \alpha_{k_2}\end{array}\right. \qquad \left\{\begin{array}{c}\gamma_1 < 0 \\ \vdots \\ \gamma_{k_3} < 0.\end{array}\right. \tag{1}$$

If this system of constraints has a solution, then x_n must satisfy

$$\max(\beta_1, \ldots, \beta_{k_1}) < x_n < \min(\alpha_1, \ldots, \alpha_{k_2}).$$

Since real numbers are dense, such x_n exists if and only if the following constraint holds:

$$\max(\beta_1, \ldots, \beta_{k_1}) < \min(\alpha_1, \ldots, \alpha_{k_2}).$$

This constraint can be equivalently rewritten as

$$\left\{\begin{array}{ll}\beta_i < \alpha_j & \text{for all } i = 1 \ldots k_1,\ j = 1 \ldots k_2,\end{array}\right. \tag{2}$$

which is again a system of linear inequalities. We call them the *shadow constraints*, because they define an $n-1$-dimensional shadow of the n-dimensional shape defined by the original constraints (1). The shadow constraints (2) combined together with $\gamma_l < 0$ comprise a new system of constraints which is equisatisfiable with (1), but does not contain the variable x_n. This process can now be repeated for x_{n-1}, and so on, until all the variables are projected.

Observe, that for a system of m constraints each elimination step may produce a new system with up to $(m/2)^2$ constraints. Therefore, eliminating n variables may, in the worst case, create a system of $4 \cdot (m/4)^{2^n}$ constraints. Thus, the decision procedure for linear inequalities based on Fourier-Motzkin even in the case of real variables has a doubly exponential worst case complexity in the number of variables.

3 Extension of Fourier-Motzkin Elimination to Integer Variables (Omega-Test)

Our version of the extension is largely based on the Omega approach [Pug91] with a few differences. First, we consider the system of *mixed integer linear constraints*

which, in addition to linear equalities and (strict) inequalities may also contain $\mathsf{int}(t)$ or $\neg\mathsf{int}(t)$ for any linear term t, meaning that the linear term t is restricted to only integer (respectively, fractional) values.

If the term t is not a variable, the constraint $\mathsf{int}(t)$ is satisfiable iff $\mathsf{int}(z) \wedge z = t$ is satisfiable, where z is a new variable. Furthermore, $\neg\mathsf{int}(t)$ is satisfiable for any term t iff $\mathsf{int}(y) \wedge y < t < y + 1$ is satisfiable for a new variable y. Hence, any system of mixed integer linear constraints may be converted to an equisatisfiable system of constraints with only equalities, inequalities, and predicates of the form $\mathsf{int}(x)$, where x is a variable.

3.1 Elimination of Equalities

As in the case of reals, all the equalities are eliminated first. If an equality contains a variable x that is not an integer, then we solve the equality for this variable and eliminate x from the system. Since this is the most efficient way of reducing the dimensionality of the problem, all such equalities are eliminated first.

Now suppose that an equality contains only integer variables:

$$\sum_{i=1}^{n} a_i x_i + c = 0. \tag{3}$$

Here we use the same variable elimination algorithm as in the Omega-test [Pug91]. If x is the only variable in (3), then there is only one value of x which can satisfy this equality constraint, namely $x = -(c/a)$. If this value is integer, we substitute it for x, and otherwise the system is unsatisfiable.

If there is more than one variable in (3), the equality is normalized such that all the coefficients a_i and the free constant c are relatively prime integers. It can always be done when the coefficients are rational numbers. If, after the normalization, there is a variable x_k whose coefficient is $|a_k| = 1$, then we simply solve for x_k and eliminate it from the rest of the system. Otherwise pick a variable x_k whose coefficient a_k is the smallest by the absolute value and define

$$m = |a_k| + 1.$$

Define also a modulus operation with the range $\left[-\frac{m}{2}, \frac{m}{2}\right)$ as follows:

$$a \,\mathbf{mod}\, m = a - m \left\lfloor \frac{a}{m} + \frac{1}{2} \right\rfloor.$$

The important properties of $\mathbf{mod}$ are that $a_k \,\mathbf{mod}\, m = -\mathrm{sign}(a_k)$, and that it distributes over addition and multiplication.

For a new integer variable σ, introduce two new constraints into the system:

$$\mathsf{int}(\sigma) \quad \text{and} \quad \sum_{i=1}^{n} (a_i \,\mathbf{mod}\, m) x_i + (c \,\mathbf{mod}\, m) = m\sigma. \tag{4}$$

The second constraint is derivable from (3) by applying $\cdot \bmod m$ on both sides of (3) and propagating $\bmod$ over addition and multiplication to the coefficients. Hence, the system remains equisatisfiable with the original.

Since $a_k \bmod m = -\text{sign}(a_k)$, the equation (4) can be solved for x_k and x_k is eliminated from the system:

$$x_k = -\text{sign}(a_k)m\sigma + \sum_{i\in[1..n]-\{k\}} \text{sign}(a_k)(a_i \bmod m)x_i + \text{sign}(a_k)(c \bmod m). \quad (5)$$

Substituting the result into the original equation (3) yields the following:

$$-|a_k|m\sigma + \sum_{i\in[1..n]-\{k\}} (a_i + |a_k|(a_i \bmod m))x_i + c + |a_k|(c \bmod m) = 0.$$

Since $|a_k| = m - 1$, it is the same as

$$-|a_k|m\sigma + \sum_{i\in[1..n]-\{k\}} ((a_i - (a_i \bmod m)) + m(a_i \bmod m))x_i + (c - (c \bmod m)) + m(c \bmod m) = 0.$$

From the definition of $\bmod$,

$$a - (a \bmod m) = m \left\lfloor \frac{a}{m} + \frac{1}{2} \right\rfloor.$$

Instantiating it in the above equation and dividing by m produces a new normalized constraint:

$$-|a_k|\sigma + \sum_{i\in[1..n]-\{k\}} a'_i x_i + c' = 0, \quad (6)$$

where $a'_i = \lfloor \frac{a_i}{m} + \frac{1}{2} \rfloor + (a_i \bmod m)$ and $c' = \lfloor \frac{c}{m} + \frac{1}{2} \rfloor + (c \bmod m)$. The new system (which is the original system with x_k eliminated using (5), and (3) rewritten as (6)) contains the same number of variables as the original one. Moreover, the new coefficients a' in (6) are guaranteed to decrease by absolute value compared to (3), namely $|a'_i| \leq \frac{2}{3}|a_i|$, except for the coefficient of σ which remains as large as that of x_k. This ensures that repeating the process above will eventually result in equality (6) having some variable with a coefficient 1 or -1. That variable can then be eliminated, reducing the overall dimensionality.

3.2 Projecting Variables from Inequalities

After eliminating all of the equalities, we are left with the system of (strict) inequalities over real and integer variables. Similar to the equality case, all the remaining real variables are projected first with the standard Fourier-Motzkin elimination procedure, resulting in a system of inequalities with only integer variables.

At this point, all the inequalities are normalized to make the coefficients be relatively prime integers, and a variable x_n is chosen for projection. Since x_n has an additional

integral constraint, we cannot simply divide the inequality by the coefficient a_n unless it is 1 or -1, and in general, the system of inequalities is rewritten in the equivalent form, very much like in (1), only the coefficients of x_n are preserved:

$$\begin{cases} \beta_1 < b_n^1 x_n \\ \vdots \\ \beta_{k_1} < b_n^{k_1} x_n \end{cases} \qquad \begin{cases} a_n^1 x_n < \alpha_1 \\ \vdots \\ a_n^{k_2} x_n < \alpha_{k_2} \end{cases} \qquad \begin{cases} \gamma_1 < 0 \\ \vdots \\ \gamma_{k_3} < 0, \end{cases} \tag{7}$$

where the coefficients a_i^j and b_i^j are positive integers. Similar to the original Fourier-Motzkin construction, for each pair of inequalities $\beta < bx_n$ and $ax_n < \alpha$, which is equivalent to

$$a\beta < abx_n < b\alpha, \tag{8}$$

the *real shadow constraint* is constructed:

$$a\beta < b\alpha. \tag{9}$$

However, the real shadow is a necessary but not a sufficient condition for the satisfiability of (8), since there might not be an integer value abx_n between $a\beta$ and $b\alpha$, even if there is a real one. In addition to the real shadow, at least one point $ab \cdot i$ must exist between $a\beta$ and $b\alpha$ for some integer i. A sufficient (but not necessary) condition is to demand that the gap between $a\beta$ and $b\alpha$ is at least $ab + 1$ wide:

$$\mathbf{D} \equiv b\alpha - a\beta > ab. \tag{10}$$

This constraint is called the *dark shadow constraint* (the object is "thick enough" to contain an integer point, and therefore casts a darker shadow; the term *dark shadow* is from [Pug91]). The dark shadow constraint is sufficient, but not necessary for an integer solution of x_n to exist. Therefore, if equation (10) makes the system unsatisfiable, we have to look for an integer solution outside of $\mathbf{D}$, i.e. in the *gray shadow*:

$$b\alpha - a\beta \leq ab, \tag{11}$$

or equivalently:

$$b\alpha \leq a\beta + ab.$$

Following the construction in the Omega-test [Pug91], $b\alpha$ on the right-hand side of (8) is replaced by the larger $a\beta + ab$, and dividing the result by a yields the following:

$$\beta < bx_n < \beta + b.$$

This means that if there is an integer solution to x_n, it must satisfy $bx_n = \beta + i$ for some $0 < i < b$, since β contains only integer variables with integer coefficients. We then try each such i in succession until a solution is found. In other words, the *gray shadow constraint* is:

$$\mathbf{G}_b \equiv \bigvee_{i=1}^{b-1} bx_n = \beta + i.$$

Alternatively, the equation (11) can be written as follows:

$$b\alpha - ab \leq a\beta.$$

Replacing $a\beta$ with the smaller value $b\alpha - ab$ on the left-hand side of (8) yields the following:

$$b\alpha - ab < abx_n < b\alpha,$$

which simplifies to

$$\alpha - a < ax_n < \alpha.$$

The gray shadow constraint in this case becomes

$$\mathbf{G}_a \equiv \bigvee_{i=1}^{a-1} ax_n = \alpha - i.$$

Either one of $\mathbf{G}_a$ or $\mathbf{G}_b$ is sufficient to guarantee integrality of x_n and the completeness of the procedure. Therefore, we can pick the one which requires fewer cases to consider. Namely, pick $\mathbf{G}_a$ if $a < b$, and $\mathbf{G}_b$ otherwise.

This is, obviously, the most expensive step of the algorithm, since it involves a lot of backtracking, but according to [Pug91], the dark shadow constraint almost always suffices in practice, and the gray shadow is often empty. Therefore, as a practical heuristic, the dark shadow constraint $\mathbf{D}$ is always tried first, and only if it fails, then a solution is searched for in the gray shadow $\mathbf{G}$.

4 Online Version of Fourier-Motzkin for Reals

In CVC, decision procedures are most effective when they are *online*, that is, the constraints are not given all at once but are fed to the decision procedure one at a time, and for each constraint the algorithm performs some relatively small amout of work to take that constraint into account and derive new constraints that follow from it.

In order to understand the reasons for being online and to clarify the important interface features that the decision procedure relies on, we give a brief introduction to the CVC framework.

4.1 Brief Introduction to CVC Framework

The goal of this subsection is to provide just enough information about the interface and underlying structure of the CVC framework to understand the requirements for the online version of the decision procedure for mixed integer linear arithmetic. Therefore, some of the features are greatly simplified or omitted. For more details on CVC framework the reader is referred to [BDS00,BDS02b,BDS02a].

At a very high level, CVC can be viewed as a SAT solver for the Boolean skeleton of the quantifier-free first-order formulas (figure 1). The SAT solver treats the atomic constraints from different theories as Boolean variables. It solves the satisfiability problem by *splitting cases* on each variable; that is, picking a variable, assigning it values

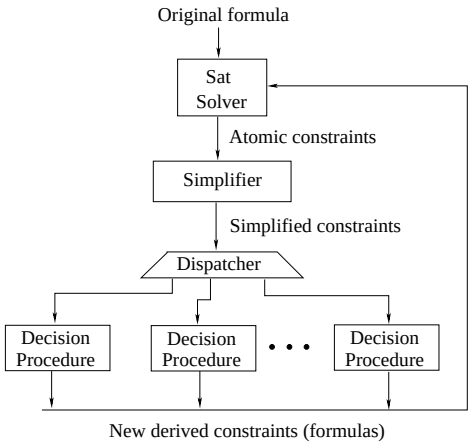

Fig. 1. Flow of constraints in CVC

true and false (making a *decision*), and solving the rest of the formula for each case recursively. If it finds a *satisfying assignment* to variables, then the original formula is satisfiable. When a particular set of decisions results in a contradiction, the SAT solver backtracks and tries a different decision. If in all branches it derives a contradiction, then the formula is unsatisfiable.

Since the variables represent constraints from various theories, each time the SAT solver makes a decision, a new constraint is produced, which is simplified and dispatched to the appropriate decision procedure. These decisions are dynamic, which requires decision procedures to be able to receive a new constraint and process it efficiently, deriving a contradiction as soon as possible to cut off the search early. This explains the *online* requirement. In some cases, however, a simplified constraint may be returned directly to the SAT solver without going through a decision procedure.

When a decision procedure receives a constraint, it derives new constraints from the current and previously seen constraints, and asserts them back to the SAT solver. If a contradiction is detected, then the decision procedure asserts false as a new constraint. Note, that the new constraints may contain arbitrary Boolean combinations of atomic formulas, but the decision procedure always receives atomic constraints (equalities and theory-specific predicates over terms). In other words, the decision procedure can assume it always solves the satisfiability problem for a *conjunction of atomic constraints*. However, it is allowed to infer Boolean combinations of new constraints from the input set of constraints.

To boost the efficiency of the SAT solver, the *intelligent backtracking* technique is utilized in conjunction with *conflict clauses* [SS96,MSS99]. To construct a conflict clause, one needs to identify a (preferrably small) set of decisions made by the SAT solver that lead to the contradiction. One way of identifying such decisions is to extract them from the *proof* that the decision procedure constructs when it derives false. This explains the need for proof production in the decision procedures in CVC. As a

bonus, the proofs can be checked by an external proof checker [Stu02] to increase the confidence in the results produced by CVC.

As mentioned earlier, each decision procedure assumes that it solves the satisfiability of a conjunction of constraints, and when new constraints arrive, they are added to the conjunction. However, when the SAT solver backtracks, some of the constraints are effectively removed. Therefore, if a decision procedure stores some information about previously received constraints, it must be able to roll back to the appropriate state when the SAT solver backtracks. In other words, all the data structures which persist across calls to the decision procedure must be *backtrackable*. Below, in the description of the algorithm, we always assume that such backtracking mechanism is properly implemented and is completely transparent to the decision procedure.

4.2 Online Fourier-Motzkin Elimination for Reals

In this and the subsequent sections the term *decision procedure* will refer to the decision procedure component from figure 1. In particular, we can always assume that the constraints dispatched to the decision procedure are *atomic* (no Boolean connectives) and *simplified*. The simplification step consists of theory-specific rewrites such as normalization of arithmetic constraints and elimination of equalities, so that only normalized inequality constraints reach the decision procedure.

Hence, the description of the algorithm consists of two parts: the simplifier, which is a set of equivalent transformations, and the decision procedure itself. The latter is presented as a function that takes a simplified atomic constraint and returns (possibly a Boolean combination of) new constraints back to the framework.

In the algorithm below, we assume a total ordering $\lessdot$ on all variables which defines the order in which the variables are projected from inequalities. In particular, x is said to be the *maximal* variable from a set of variables when it is the highest in this set w.r.t. $\lessdot$. In this section, we only consider equalities and inequalities over real-valued variables. Handling the constraints with $\mathsf{int}(t)$ predicate and integer variables will be described later in section 5.

Simplification step. Each equality constraint $t_1 = t_2$ is first rewritten as $t_1 - t_2 = 0$ and simplified by grouping like terms. If the resulting equality contains no variables (meaning $t_1 - t_2$ simplifies to a numerical constant), then it is checked for satisfiability, and the result is reported directly to the top-level SAT solver. Otherwise, it is rewritten in the form $x = \alpha$ for some variable x, and then x is replaced by α everywhere in the system, completely eliminating the variable x.

Similarly, an inequality $t_1 < t_2$ is rewritten as $t_1 - t_2 < 0$ and simplified. If the left-hand side simplifies to a constant, the inequality is evaluated to true or false and submitted back to the solver. Otherwise, it is rewritten as $\beta < x$ or $x < \alpha$ for the maximum variable x in $t_1 - t_2$ w.r.t. $\lessdot$, and forwarded to the decision procedure.

Decision procedure. Due to the simplification step above, the decision procedure receives only inequalities of the form $\beta < x$ or $x < \alpha$, where x is the maximal variable in α and β w.r.t. $\lessdot$. We say that the variable x in such inequalities is *isolated*.

The decision procedure maintains a backtrackable database $\mathrm{DB}_<$ of inequalities indexed by the isolated variable. Whenever a new inequality $x < \alpha$ arrives, this database is searched for the opposite inequalities $\beta < x$ and for each such inequality the new shadow constraint $\beta < \alpha$ is constructed and asserted back to the framework. The received constraint $x < \alpha$ is then added to the database. The inequalities of the form $\beta < x$ are handled similarly.

The newly generated constraint $\beta < \alpha$ is eventually simplified and submitted back to the decision procedure with a smaller variable (w.r.t. $\lessdot$) isolated, and this process repeats until no variables remain in the constraint.

The ordering $\lessdot$ on the variables guarantees that all the intermediate constraints that would be constructed by the offline version of the procedure are eventually constructed and processed by this online version, provided both algorithms project variables according to the same ordering. Assuming that the original offline version of Fourier-Motzkin elimination is complete and terminating implies that the online procedure is also complete and terminating. We formulate the completeness part of this statement more precisely as a lemma.

Lemma 1. ***(Local Completeness.)*** *Let $\mathbf{C} = \{C_1, \ldots, C_k\}$ be a set of linear arithmetic inequalities over real-valued variables $x_1, \ldots, x_n$, and $\lessdot$ be a total ordering on the variables. Let $\mathcal{C}_{\mathrm{all}} \supseteq \mathbf{C}$ be the set of constraints processed by the offline algorithm while solving the original set of constraints $\mathbf{C}$. Then any constraint $C \in \mathcal{C}_{\mathrm{all}}$ is also processed by the online algorithm, regardless of the order in which the original constraints from $\mathbf{C}$ are submitted to the online algorithm.*

Proof. (*Sketch*) For simplicity, assume that there are no equality constraints in the original system $\mathbf{C}$.

Consider an arbitrary constraint $t < 0 \in \mathcal{C}_{\mathrm{all}}$, where t is a linear arithmetic term, and show that this constraint is either in $\mathbf{C}$, or is generated by the online version of the algorithm.

The argument is by induction on the number of projected variables at the time the constraint is generated. The base case is when all the variables are still present; then the constraint $t < 0$ must be in the original system of constraints: $t < 0 \in \mathbf{C}$, which trivially satisfies the claim.

For the inductive step, suppose that the constraint $t < 0$ is generated while projecting the variable x_i. Then $t < 0$ is a rewritten form of some shadow constraint $\beta < \alpha$ generated from the pair of constraints $\beta < x_i < \alpha$. Since these two constraints contain x_i, the induction hypothesis applies, and we can assume that both $\beta < x_i$ and $x_i < \alpha$ are eventually constructed by the online procedure. Since we project variables in a fixed order, the variable x_i is greater (w.r.t. $\lessdot$) than any of the variables in α and β. Hence, the solver is guaranteed to isolate x_i in both inequalities when they are processed, and they are indeed rewritten as $\beta < x_i$ and $x_i < \alpha$ before they are submitted to the decision procedure.

Suppose that $\beta < x_i$ arrives first and is recorded in the local database. When $x_i < \alpha$ eventually arrives, the procedure finds the constraint $\beta < x_i$ in the database and $\beta < \alpha$ is generated. Similarly, the same constraint is generated when the constraints arrive in the opposite order. This completes the inductive step and the proof of the claim.

When the original system $\mathbf{C}$ contains equations, each time an equality is eliminated from the system, the affected constraints are resubmitted to the algorithm by the top-level SAT solver engine. Again, for simplicity, we can assume that *all* the remaining equations and all the original inequalities are updated accordingly and resubmitted to the algorithm. This guarantees that when the last equality is eliminated, the resulting system of inequalities will be exactly the same as in the offline algorithm after eliminating all equations. □

In the absense of equality constraints we can even show that the dual claim holds, that is, all the constraints generated by the online procedure are also generated by the offline one (provided that it does not terminate immediately after finding a contradiction, but proceeds until all the constraints are processed), which would prove the two procedures to be functionally identical. However, it is not needed for completeness. As the soundness go, it is sufficient to show that every constraint we generate does indeed follow from the previously generated constraints, which is easy to see by inspection. Thus, the following theorem holds.

Theorem 1. ***(Soundness and completeness).*** *The online version of the algorithm is sound and complete for the system of linear arithmetic constraints over real-valued variables.*

If we allow the equality constraints, then the online procedure will actually generate more constraints than the offline one, as can be seen from the proof sketch of Lemma .1 The additional constraints, however, do not affect soundness, they are simply redundant. Once the variable x is eliminated, the recorded constraints of the form $\beta < x$ and $x < \alpha$ are rewritten with the new value of x and the updated constraints are resubmitted to the decision procedure again. This time another variable is isolated and projected. Once all of the equalities have been processed, further projections are done exactly as in the offline procedure. However, more constraints may still be generated from the extra shadow constraints.

5 Online Fourier-Motzkin Elimination for Mixed Integer Constraints

The online version of the decision procedure for integers cannot have such a direct correspondence to the offline version, since the order of the projection of variables depends on the integrality constraints, $\mathsf{int}(x)$ and $\neg\mathsf{int}(x)$, and the variable may become known to be integer only after it has already been projected or eliminated. A naive solution would be to backtrack and redo the projection and elimination steps. This could be a very costly operation.

Fortunately, there is a simple and elegant solution to this problem. Whenever a constraint $\mathsf{int}(x)$ arrives, a new constraint $x - \sigma = 0$ is added to the system, where σ is a new integer variable, and the fact $\mathsf{int}(\sigma)$ is recorded into a local database $\mathrm{DB}_{\mathrm{int}}$ indexed by σ. The resulting system is equisatisfiable with the original one (which includes $\mathsf{int}(x)$), but the variable x remains real-valued in the new system. Therefore,

the projections and eliminations of x do not have to be redone. At the same time, the integrality of x is enforced by the integrality of σ.

In addition, for any integer constraint $\mathsf{int}(t)$ in $\mathrm{DB}_{\mathrm{int}}$, whenever the term t is rewritten to t' (because of some variable elimination), and t' simplifies to a constant term c, one must check that c is indeed an integer and assert unsatisfiability if it is not.

Just like the algorithm for only real variables, the online algorithm for deciding mixed-integer linear constraints consists of two parts: simplification and decision procedure.

Simplification step. This version of the simplifier performs the same transformations as the one for real variables (section 4.2). An equality constraint is first rewritten as $\gamma = 0$ and γ is checked for being a constant. If it is, the constraint is immediately checked for satisfiability and the result is returned directly to the SAT solver. Otherwise, if γ contains real-valued variables, then one such variable x is isolated and eliminated. If only integer variables remain in γ, then the iterative equality elimination algorithm is performed, as described in section 3.1. At the end of this process the equality $\gamma = 0$ is rewritten into an equisatisfiable system of equations

$$\begin{cases} x_1 = \beta_1 \\ \quad \vdots \quad \vdots \\ x_k = \beta_k, \end{cases}$$

where each equation $x_i = \beta_i$ corresponds to the equation (5) in section 3.1 from each iteration of the algorithm. All the variables x_i are then eliminated by replacing them with their right-hand sides. Thus, equations are handled in the simplification step and never submitted to the actual decision procedure.

Inequalities are also transformed and simplified into $\gamma < 0$, then evaluated if γ is a constant. If γ containts variables, the inequality is rewritten to the form $\beta < ax$ or $ax < \alpha$ for some positive integer coefficient a, where the variable x is the maximal w.r.t. $\lessdot$. The new inequality is then forwarded to the decision procedure. Similar to the offline version of the algorithm, it is important to project real variables first. Therefore, we define $\lessdot$ such that real-valued variables are always higher in the ordering than the integer ones.

In the constraints of the form $\mathsf{int}(t)$ and $\neg\mathsf{int}(t)$ only the term t is simplified by combining like terms, and otherwise these constraints are passed to the decision procedure unmodified.

5.1 The Decision Procedure

First, observe that, due to the simplification step, only inequalities of the form $\beta < bx$ and $ax < \alpha$ and integer constraints $\mathsf{int}(t)$ and $\neg\mathsf{int}(t)$ are submitted to the decision procedure. Notice that inequality constraints always have the maximal variable isolated w.r.t. $\lessdot$. These inequalities are stored in the local database $\mathrm{DB}_<$. Additionally, whenever a term t in any constraint $\mathsf{int}(t) \in \mathrm{DB}_{\mathrm{int}}$ is rewritten to t' by the simplifier, $\mathsf{int}(t)$ is automatically replaced by $\mathsf{int}(t')$ in $\mathrm{DB}_{\mathrm{int}}$. Both local databases are also backtrackable.

The decision procedure receives a constraint C from the simplifier and returns, or *asserts*, new constraints back to the framework. We describe it as a case-by-case analysis of the constraint C.

1. $C \equiv \mathsf{int}(t)$:
   (a) If t is a constant, then evaluate $\mathsf{int}(t)$, assert the result to the framework, and return. If t is not a constant, go to step 1b.
   (b) If $t \notin \mathrm{DB}_{\mathrm{int}}$, then create a new integer variable z, add t and z into $\mathrm{DB}_{\mathrm{int}}$, assert the new facts:
   $$\mathsf{int}(z) \quad \text{and} \quad t - z = 0.$$
   Otherwise, if $\mathrm{t} \in \mathrm{DB}_{\mathrm{int}}$, then ignore C and return.
2. $C \equiv \neg\mathsf{int}(t)$:
   Introduce a new integer variable z, add z to $\mathrm{DB}_{\mathrm{int}}$ and assert the new constraints:
   $$\mathsf{int}(z) \quad \text{and} \quad z < t < z + 1.$$
3. $C \equiv a \cdot x < \alpha$ (or $C \equiv \beta < b \cdot x$):
   (a) Find all inequalities of the form $\beta < b \cdot x$ (respectively, $a \cdot x < \alpha$) in the database $\mathrm{DB}_{<}$, and for each such inequality perform the following steps:
      i. Generate and assert the real shadow constraint $R \equiv a \cdot \beta < b \cdot \alpha$.
      ii. If $x \in \mathrm{DB}_{\mathrm{int}}$ (in which case all the variables in α and β must also be in $\mathrm{DB}_{\mathrm{int}}$), then generate the integrality constraint $\mathbf{D} \vee \mathbf{G}$ (dark and gray shadows), where $\mathbf{D}$ and $\mathbf{G}$ are defined as in section 3.2:
      $$\mathbf{D} \equiv b \cdot \alpha - a \cdot \beta > ab$$
      and the gray shadow constraint G is
      $$\mathbf{G} = \bigvee_{i=1}^{a-1} a \cdot x = \alpha - i, \quad \textbf{if } a < b,$$
      or
      $$\mathbf{G} = \bigvee_{i=1}^{b-1} b \cdot x = \beta + i, \quad \textbf{if } a \geq b.$$
      Following the heuristic of Pugh [Pug91], the top-level SAT solver should first search for a solution in $\mathbf{D}$ before trying $\mathbf{G}$.
   (b) Add the received constraint C to $\mathrm{DB}_{<}$ and return.

It is not hard to see that each step of the algorithm above corresponds very closely to the similar steps in the offline version of the algorithm. The soundness and completeness of the procedure follow from the fact that the set of constraints asserted by the decision procedure at each step is always equisatisfiable with the given constraint C.

The experiments shown in table 1 indicate that most of the time the overhead of the CVC framework and arbitrary precision arithmetic slows down the algorithm only by a constant factor. Since this implementation is not yet tuned for efficiency, there are a

#experiments in each suite	#formulas / #vars in each experiment	CVC completed	Omega completed	avg. slow-down factor
5996	1-4/1-5	5990 (99.9%)	5568 (92%)	13.4
395	1-10/1-20	387 (98%)	322 (81%)	192
65	10-50/10-50	61 (95%)	8 (12%)	7.8

Table 1. Experimental comparisons of CVC vs. Omega on suites of randomly generated examples. CVC is generally slower than Omega approximately by a factor of 10.

few exceptional cases when CVC performs much worse than Omega, which explains the large slow-down factor in the second line of table 1. In any case, this is a very reasonable price to pay for having the arithmetic decision procedure combined with other theories of CVC. Additionally, the resulting implementation proves to be much more stable than the original Omega-test.

Lemma 2. ***(Local soundness and completeness).*** *Let C be any linear arithmetic or integrality constraint, $\mathcal{C}$ be the set of constraints previously processed by the algorithm, and let $\mathcal{C}'$ be the set of new constraints asserted during the run of the simplifier and the decision procedure on C. Then $C \wedge \bigwedge \mathcal{C}$ and $\bigwedge \mathcal{C}'$ are equisatisfiable formulas.*

Proof. By inspecton of all the cases. □

Theorem 2. ***(Soundness, completeness and termination).*** *Given a finite set of linear arithmetic and integral constraints $\mathbf{C}$, the algorithm above as a part of the CVC framework satisfies the following properties:*

1. *If it terminates, it returns SAT if $\mathbf{C}$ is satisfiable;*
2. *If it terminates, it returns UNSAT if $\mathbf{C}$ is unsatisfiable;*
3. *It terminates.*

Proof. (Rough sketch.) Soundness (1) and completeness (2) follow from Lemma 2.

The proof of termination (3) can be done by well-founded induction over the number of constraints submitted to the top-level SAT solver, which can be thought of as the number of the recursive calls to the SAT solver. The measure for such induction is

$$M = (N_{\neg \mathsf{int}()}, N_{x_1}, \ldots, N_{x_n}),$$

where $N_{\neg \mathsf{int}()}$ is the number of $\neg \mathsf{int}(t)$ constraints in the set of original input constraints $\mathbf{C}$, and N_{x_i} is the number of inequalities that contain the variable x_i. The variables x_i are all the variables that are present in the original set of constraints and those that will be introduces by the decision procedure. The variables in the sequence $x_1, \ldots, x_n$ are sorted in descending order w.r.t. $\lessdot$, the projection ordering.

This measure M strictly decreases, w.r.t. the lexicographic ordering on the values of M, for each recursive call of the top level SAT solver, assuming that the contstraints of the form $\neg \mathsf{int}(t)$ are processed only once.[1] Since the branching factor is finite, the original set of constraints is finite, and the number of newly generated constraints in each

[1] Other constraints have to be re-processed every time a variable is eliminated from an equality.

call to the simplifier and the decision procedure is finite, the recursion is guaranteed to terminate. □

6 Proof Production

When the algorithm in section 5 reports that the system of constraints is unsatisfiable, it produces a proof of this fact which can be verified independently by an external proof checker. This increases our confidence in the soundness of the implementation.

Additionally, proof production mechanism allows CVC framework to extract logical dependencies that drive the backtracking mechanism of the built-in SAT solver (see section 4.1). The details are out of the scope of this paper, but intuitively, if the proof of unsatisfiability depends only on a small subset of decisions made by the SAT solver, then the SAT solver memorizes this combination of decisions and avoids it in the future. This can dramatically reduce the size of the decision tree.

6.1 Natural Deduction

Since the algorithm in section 5 consists of the set of relatively simple transformations which take existing constraints and assert new ones, it is natural to construct the proofs for the new assertions, taking the given facts as assumptions.

The proof production interface, therefore, is implemented as follows. Whenever our algorithm receives a new constraint, it also receives a proof that this constraint holds (or a *proof of this constraint*). Whenever the algorithm asserts a new constraint, it also builds and supplies a proof of this new constraint, which may include the available proofs of the previously submitted constraints. In other words, the algorithm maintains the invariant, that every constraint appearing in the algorithm has an associated proof with it.

The proofs are represented as derivations in natural deduction extended with arithmetic and with specialized derived or admissible rules. The specialized rules can then be proven sound externally, either by manual inspection, or with the help of automated theorem provers. Ideally, every step of the algorithm should correspond to one such proof rule, to minimize the proof size. However, some steps of the algorithm have conditional branches or substeps, and it is often convenient to break the proof for those into smaller subproofs, resulting in additional but simpler proof rules.

Definitions. A *judgement* (or a *statement*) in our version of natural deduction is just a formula in the quantifier-free first-order logic with arithmetic. Most often, the formula has the following forms:

1. Equivalence of two arithmetic constraints: $A \equiv B$;
2. Equality or inequality of two arithmetic terms: $t_1 = t_2$ or $t_1 < t_2$;
3. Boolean combination of the above.

The *inference rules*, or *proof rules*, are normally of the form

$$\frac{P_1 \quad \cdots P_n \quad S_1 \quad \cdots S_m}{C}\ \text{rule name}$$

where judgements P_i are the *premisses* (they are assumed to have proofs), S_i are side conditions (the rule is applicable only if all S_i are true), and the judgement C is the *conclusion* of the rule. The semantics of a proof rule is that if all P_i are valid, then C is also valid, provided that all the side conditions S_j are true. Typical rules of this form are "and introduction" and "modus ponens:"

$$\frac{A \quad B}{A \wedge B} \wedge I \qquad \frac{A \quad A \to B}{B} \mathrm{MP}.$$

The other type of proof rules introduces *assumptions*. For instance, in order to show that an implication $A \to B$ is valid, one needs to show that, given a proof of A, it is possible to construct a proof of B. But A does not have to be valid in order for the implication to be valid. Therefore, the "implication introduction" rule has the form:

$$\frac{\begin{array}{c}\overline{}\,u \\ A \\ \vdots \\ B\end{array}}{A \to B} \to I^u$$

Here u is a *parametric derivation*, an assumed proof of A which can only be used above the line of this rule in the proof of B. The dots between A and B must be filled in with an actual proof of B.

A *derivation* of a formula A is a tree, whose nodes are formulas, A is the root, and the children of each node F are premisses in some rule application in which F is a conclusion. For example, a proof of a formula $A \wedge (A \to B) \to B$ is the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (A \to B)}\,u}{A} \wedge E_1 \qquad \dfrac{\dfrac{}{A \wedge (A \to B)}\,u}{A \to B} \wedge E_2}{B}\,\mathrm{MP}}{A \wedge (A \to B) \to B} \to I^u$$

Notice, that the assumption u is used twice in the proof of B.

Proof rules for equivalent transformations. It is often necessary to rewrite a term to a different but an equivalent term. For instance, our algorithm assumes that all the linear arithmetic terms are reduced by the framework to a canonical form: $\sum_{i=1}^{n} a_i \cdot x_i$, even though the new asserted facts may include non-canonical forms.

To justify such transformations in the proofs, specialized proof rules are introduced of the form

$$\frac{S_1 \cdots S_n}{t_1 = t_2},$$

where S_i are side conditions, and t_i are arithmetic terms. The intended meaning of such a rule is that the term t_1 can be equivalently rewritten to t_2.

Similarly, arithmetic constraints are often rewritten in a similar way:

$$\frac{S_1 \cdots S_n}{c_1 \equiv c_2}.$$

This way constraints can be modified without having to prove them. Sequences of such transformations are glued together with the *transitivity rule*:

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\text{trans},$$

and these equalities can be used to derive new equalities using *substitution*:

$$\frac{t_1 = t'_1 \quad \cdots \quad t_n = t'_n}{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}\text{subst}.$$

Proof rules for the algorithm. We divide the proof rules for our decision procedure into *transformations* and *inferences*. Generally, all the transformations in the simplifier (canonization of terms, elimination of equalities, and variable isolation) are equivalent transformations. In this paper, we do not consider canonization of terms and refer the reader to [Stu02].

Fourier-Motzkin projection of a variable and handling the $\mathsf{int}()$ predicate involves deriving new constraints from the existing ones. Therefore, their proof rules have a form of *inference*, that is, a general rule which derives a free-form constraint, not necessarily an equality, from already proven premisses.

6.2 Proof Rules for Equivalent Transformations

Normalization. Multiplying an (in)equality by a positive number preserves the constraint. These rules assume that the right-hand side is always 0, which is the property of the normal form for the arithmetic constraints.

$$\frac{f \in \mathcal{R},\ f > 0}{\sum_{i=1}^{n} a_i \cdot x + c < 0 \quad \equiv \quad f \cdot (\sum_{i=1}^{n} a_i \cdot x + c) < 0}\text{norm}_<$$

$$\frac{f \in \mathcal{R}}{\sum_{i=1}^{n} a_i \cdot x + c = 0 \quad \equiv \quad f \cdot (\sum_{i=1}^{n} a_i \cdot x + c) = 0}\text{norm}_=$$

Variable Isolation. Given an (in)equality, pick a variable to isolate and transform the constraint in such a way that the variable with some positive coefficient is solely on one side, and the rest of the term is on the other. For equalities, the isolated variable must always be on the left-hand side. For inqualities, it depends on the sign of the coefficient: if it is positive, the variable stays on the left-hand side, and all the other terms are moved to the right-hand side; otherwise the variable is isolated on the right-hand side.

The four rules below implement the four cases of variable elimination: for equality and inequality, and for positive and negative coefficients:

$$\frac{a_i > 0}{\begin{array}{l} c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n = 0 \\ \equiv a_i \cdot x_i = -(c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n) \end{array}} \mathrm{VI}^+_=$$

$$\frac{a_i < 0}{\begin{array}{l} c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n = 0 \\ \equiv -a_i \cdot x_i = c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n \end{array}} \mathrm{VI}^-_=$$

$$\frac{a_i > 0}{\begin{array}{l} c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n < 0 \\ \equiv a_i \cdot x_i < -(c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n) \end{array}} \mathrm{VI}^+_<$$

$$\frac{a_i < 0}{\begin{array}{l} c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n < 0 \\ \equiv c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n < -a_i \cdot x_i \end{array}} \mathrm{VI}^-_<$$

Elimination of Equalities. Most of the rules in this subsection implement the low-level details of the variable elimination in equalities. They are divided into two groups: elimination of real variables and elimination of integer variables.

Elimination of real variable. The first two rules are axioms of multiplication by 1:

$$\frac{}{1 \cdot t = t}\text{mult drop one} \qquad \frac{}{t = 1 \cdot t}\text{mult one.}$$

When the isolated variable x is real, its coefficient is normalized to 1:

$$\frac{a \neq 0}{a \cdot x = \alpha \quad \equiv \quad x = \alpha/a}\text{Eq Elim.}$$

After this transformation, when the new fact $x = \alpha/a$ propagates to the global database of facts, the canonical form of x becomes α/a. This fact is recorded into the global database, and all the existing constraints containing x will be automatically updated with the new value of x and re-canonized. This effectively eliminates x from the system of constraints, both current and future ones.

Elimination of integer variable. When all the variables in the constraint are integers, including the isolated variable x, the elimination of the variable has to go through a number of iterative transformations, as described in section 3.1. One step of each iteration is formalized as the following proof rule:

$$\frac{\begin{array}{r} (C \bmod m + \sum_{i=1}^n (a_i \bmod m) \cdot x_i + x)/m = t \\ -(C \bmod m + \sum_{i=1}^n (a_i \bmod m) \cdot x_i - m \cdot \sigma(t)) = t_2 \\ \mathbf{f}(C) + \sum_{i=1}^n \mathbf{f}(a_i) \cdot x_i - a \cdot \sigma(t) = t_3 \\ \mathsf{int}(x) \quad \mathsf{int}(x_i) \quad a : \mathcal{N} \quad C : \mathcal{N} \quad a_i : \mathcal{N} \text{ for } i = 1..n \quad a \geq 2 \end{array}}{a \cdot x = C + \sum_{i=1}^n a_i \cdot x_i \quad \equiv \quad x = t_2 \wedge t_3 = 0}\text{Eq Elim int}$$

where $m = a + 1$, and

$$i \bmod m = i - m \left\lfloor \frac{i}{m} + \frac{1}{2} \right\rfloor,$$

$$\mathbf{f}(i) = \left\lfloor \frac{i}{m} + \frac{1}{2} \right\rfloor + i \bmod m.$$

The first three premisses of this rule can be thought of as definitions of t, t_2, and t_3 for the use in the rest of the formulas. In the actual implementation, however, t, t_2, and t_3 are the canonized versions of their left-hand sides. Thus, for example, the new "variable" $\sigma(t)$ is constructed from the canonical term t, and not the left-hand side expression in the first premiss.

Note, that instead of introducing a new variable σ, we use a single *function* $\sigma(t)$, which is axiomatized as $\sigma(t) = \lfloor t \rfloor$ in section 6.3. This way we avoid the need to generate new variable names (which is not directly expressible in natural deduction) and guarantee that exactly the same "variable" will be introduced for exactly the same equality, even if it is processed several times.

System of equalites in solved form. Note, that the rule above transforms an equality into a conjunction of two equalities; one is in a *solved form* (the variable on the left-hand side has coefficient 1), and the other is not. The equality elimination algorithm guarantees that eventually the second equality can be transformed into solved form directly by isolating a variable, and at that moment the original equality will correspond to a conjunction of n equalities in solved form:

$$\begin{array}{r} x_1 = t_1 \\ \wedge x_2 = t_2 \\ \vdots \quad \vdots \\ \wedge x_n = t_n. \end{array}$$

For technical reasons (since we implement the algorithm as a Shostak decision procedure [Sho77,BDS02a]), such conjunction of equalities needs to be transformed into a *system of equalities in solved form*. The additional requirement is that no isolated variable on the left-hand side can appear on the right-hand side of any equality. Since no variable x_j appears in t_i for $j \leq i$, this can be achieved by substituting x_i's in all the previous right-hand sides t_j for $j < i$, starting from x_n.

If done directly, the first equality $x_1 = t_1$ will be transformed $n - 1$ times. This procedure can be optimized to transform each t_i only once by substituting all $x_{i+1}, \ldots, x_n$ *simultaneously* for the new (already transformed) right-hand sides $t'_{i+1}, \ldots, t'_n$ respectively, obtaining a new i-th right-hand side t'_i.

The following proof rule is designed for one step of such substitution:

$$\frac{B_1 \equiv C \wedge B_2 \quad B_2 \equiv A_1 \wedge \cdots \wedge A_n \qquad \begin{array}{c} \overline{A_1 \wedge \cdots \wedge A_n}\, u \\ \vdots \\ C \equiv A \end{array}}{B_1 \equiv A_1 \wedge \cdots \wedge A_n \wedge A} \equiv^u_\wedge$$

Here B_1 is the original equality $a \cdot x_i = t_i$ which was transformed into $C \wedge B_2$ by the Eq Elim Int rule, where B_2 is the intermediate constraint $t_3 = 0$ which is later transformed into the conjunction of solved equalities. A_i is a "new," already transformed, equality $x_i = t'_i$, and C is the current equality in solved form, $x_i = t_i$, to be transformed at the current step by substituting t'_j for x_j, $j > i$.

Assuming that the first $n = i - 1$ equalities are already transformed into the system of solved equalities ($B_2 \equiv A_1 \wedge \cdots \wedge A_n$), prove that the transformation of the next equality C into A makes the larger set of equalities B_1 also a system of solved equalities: $B_1 \equiv A_1 \wedge \cdots \wedge A_n \wedge A$. This new system of equalities will be used in the next iteration as an assumption to this same rule as $B_2 \equiv \cdots$, and so on.

This rule is derivable from the existing propositional rules and the substitution rules already implemented in CVC, and we provide it here only to "cut the corner" and simplify the proof. The derivation is quite straightforward but rather tedious, and we do not provide it in this paper.

6.3 Elimination of Inequalities: Inference Rules

The proof rules in this section derive new constraints from already existing ones. This type of rules is needed for the actual Fourier-Motzkin elimination, or *projection* of variables from inequalities, and for handling the $\mathsf{int}()$ predicate.

Handling the $\mathsf{int}()$ Predicate. When a non-integer constant is constrained to be an integer, such constraint is unsatisfiable, and the following rule applies:

$$\frac{\mathsf{int}(n : \mathcal{R}) \quad n \notin \mathcal{N}}{\mathsf{false}}\text{Const\_int.}$$

The rules below are axiomatization of $\sigma(t)$ as $\lfloor t \rfloor$. The last two rules are used to handle the negation of the $\mathsf{int}()$ predicate.

$$\frac{}{\mathsf{int}(\sigma(t))}\text{int}_\sigma \qquad \frac{\mathsf{int}(t)}{t - \sigma(t) = 0}\text{Term\_int}$$

$$\frac{\mathsf{int}(t) \equiv \mathsf{false}}{\sigma(t) < t}\neg\text{int}_{<t} \qquad \frac{\mathsf{int}(t) \equiv \mathsf{false}}{t < \sigma(t) + 1}\neg\text{int}_{\text{t}<}$$

Projecting the Isolated Variable. The remaining rules implement the projection steps from the algorithm in section 5.1 with some optimizations.

Real shadow. Deriving real shadow from two opposing constraints makes a simple and obvious proof rule:

$$\frac{\beta < b \cdot x \quad a \cdot x < \alpha}{a \cdot \beta < b \cdot \alpha}\text{Real Shadow.}$$

Dark and gray shadows. Instead of asserting the disjunction of dark and gray shadow in the fully expanded form as given in the algorithm, we "hide" the constraints under special predicates $\mathbf{D}(t_1,\, t_2)$ and $\mathbf{G}(t_1,\, t_2,\, n)$ and later expand them "lazily." The intended semantics for these predicates is the following:

$$\begin{aligned}
\mathbf{D}(t_1,\, t_2) &\equiv t_1 < t_2 \\
\mathbf{G}(t_1,\, t_2,\, 0) &\equiv \mathsf{false} \\
\mathbf{G}(t_1,\, t_2,\, i) &\equiv t_1 = t_2 + i \vee \mathbf{G}(t_1,\, t_2,\, i-1) \\
\mathbf{G}(t_1,\, t_2,\, -i) &\equiv t_1 = t_2 - i \vee \mathbf{G}(t_1,\, t_2,\, -i+1),
\end{aligned}$$

where $i > 0$. Thus, the type of the gray shadow (for $a < b$ and $a \geq b$) is encoded into the sign of the bound n of $\mathbf{G}(t_1,\, t_2,\, n)$, and the large disjunction is expanded one disjunct at a time.

The following rules infer the new constraint $\mathbf{D} \vee \mathbf{G}$ in our special encoding. They also include an optimization for the case when either a or b is 1, so the gray shadow is not necessary. The seeming complexity of the rules comes from the fact that they are only sound when all the variables and coefficients are integer. Such premises and side conditions are the primary reason for the complexity. However, the key premises are always the two opposing inequalities, $\beta < b \cdot x$ and $a \cdot x < \alpha$, as in the Real Shadow rule.

$$\frac{\begin{array}{l}\beta < b \cdot x \quad a \cdot x < \alpha \\ \mathsf{int}(x) \quad 2 \leq a \quad 2 \leq b \\ \mathsf{int}(x_i) \quad b_i : \mathcal{N} \text{ for } i \in 1..n \quad \mathsf{int}(y_j) \quad a_j : \mathcal{N} \text{ for } j \in 1..m\end{array}}{\mathbf{D}(ab,\, b\alpha - a\beta) \vee \mathbf{G}(a \cdot x,\, \alpha,\, -a+1)}\, \mathbf{D} \text{ or } \mathbf{G}_{2 \leq a \leq b}$$

$$\frac{\begin{array}{l}\beta < b \cdot x \quad a \cdot x < \alpha \\ \mathsf{int}(x) \quad 2 \leq a \quad 2 \leq b \\ \mathsf{int}(x_i) \quad b_i : \mathcal{N} \text{ for } i \in 1..n \quad \mathsf{int}(y_j) \quad a_j : \mathcal{N} \text{ for } j \in 1..m\end{array}}{\mathbf{D}(ab,\, b\alpha - a\beta) \vee \mathbf{G}(b \cdot x,\, \beta,\, b-1)}\, \mathbf{D} \text{ or } \mathbf{G}_{2 \leq b < a}$$

$$\frac{\begin{array}{l}\beta < b \cdot x \quad a \cdot x < \alpha \\ \mathsf{int}(x) \quad (a = 1 \vee b = 1) \\ \mathsf{int}(x_i) \quad b_i : \mathcal{N} \text{ for } i \in 1..n \quad \mathsf{int}(y_j) \quad a_j : \mathcal{N} \text{ for } j \in 1..m\end{array}}{\mathbf{D}(ab,\, b\alpha - a\beta)}\, \mathbf{D} \text{ or } \mathbf{G}_{a,b \leq 1}$$

Expanding dark shadow. When the dark shadow constraint arrives as a new fact to our decision procedure, it is time to expose its interpretation:

$$\frac{\mathbf{D}(t_1,\, t_2)}{t_1 < t_2}\, \text{Expand } \mathbf{D}.$$

In principle, the built-in SAT solver may also decide to assert the negation of the dark shadow in some other decision branch, but then it will have to assert the gray shadow. Therefore, there is no rule for the negation of $\mathbf{D}(t_1,\, t_2)$, and it is simply ignored.

Expanding gray shadow. Similarly, when $\mathbf{G}(t_1,\, t_2,\, n)$ arrives as a new fact, we would like to expose its interpretation to the framework (and ignore its negation, similar to the dark shadow). However, when n is large, the resulting formula is a big disjunction, and asserting it all at once may cause too many unnecessary decision points for the SAT solver. Therefore, the disjuncts are expanded one at a time.

Additionally, when t_2 is a constant term c, the expansion of $\mathbf{G}(t_1,\, c,\, n)$ creates disjuncts of the form $a \cdot x = c'$ for some constant c' from a finite range. Such disjunct is satisfiable only if $c' \equiv 0 \pmod{a}$. If $a > 1$, then most of the disjuncts will be unsatisfiable and can be safely eliminated from the expansion, significantly reducing the number of disjuncts. The following three rules are designed for this particular optimization.

Constant RHS: First, define the following abbreviations:

$$
\begin{aligned}
|t| &:= \mathbf{if}\ t \geq 0\ \mathbf{then}\ t\ \mathbf{else}\ -t \\
\mathrm{sign}(x) \cdot y &:= \mathbf{if}\ x > 0\ \mathbf{then}\ y\ \mathbf{elsif}\ x = 0\ \mathbf{then}\ 0\ \mathbf{else}\ -y \\
j(c,\, b,\, a) &:= \begin{cases} b > 0 : (c + b) \bmod a \\ b < 0 : (a - (c + b)) \bmod a \end{cases}
\end{aligned}
$$

Using the above notation, the proof rules for the case when t_2 is a constant in $\mathbf{G}(t_1,\, t_2,\, n)$ are written as follows:

$$\frac{\mathbf{G}(a \cdot x,\, c : \mathcal{N},\, b : \mathcal{N}) \quad j(c,\, b,\, a) \geq |b|}{\mathsf{false}}\,\mathrm{Expand}\,\mathbf{G}\,\mathrm{const}_0$$

$$\frac{\mathbf{G}(a \cdot x,\, c : \mathcal{N},\, b : \mathcal{N}) \quad j(c,\, b,\, a) < |b| \leq a + j(c,\, b,\, a)}{a \cdot x = c + b - \mathrm{sign}(b) \cdot j(c,\, b,\, a)}\,\mathrm{Expand}\,\mathbf{G}\,\mathrm{const}_1$$

$$\frac{\mathbf{G}(a \cdot x,\, c : \mathcal{N},\, b : \mathcal{N}) \quad a + j(c,\, b,\, a) < |b|}{\begin{array}{l} a \cdot x = c + b - \mathrm{sign}(b) \cdot j(c,\, b,\, a) \\ \vee\ \mathbf{G}(a \cdot x,\, \mathrm{term}(c),\, b - \mathrm{sign}(b) \cdot (a + j(c,\, b,\, a))) \end{array}}\,\mathrm{Expand}\,\mathbf{G}\,\mathrm{const}.$$

Non-constant RHS: When t_2 is not a constant, the following rules expand $\mathbf{G}(t_1,\, t_2,\, n)$ exactly as prescribed by the intended semantics:

$$\frac{\mathbf{G}(t_1,\, t_2,\, i : \mathcal{N}) \quad i \geq 2}{\mathbf{G}(t_1,\, t_2,\, i - 1) \vee t_1 = t_2 + i}\,\mathrm{Expand}\,\mathbf{G}_{\geq 2}$$

$$\frac{\mathbf{G}(t_1,\, t_2,\, i : \mathcal{N}) \quad i \leq -2}{\mathbf{G}(t_1,\, t_2,\, i + 1) \vee t_1 = t_2 + i}\,\mathrm{Expand}\,\mathbf{G}_{\leq -2}$$

$$\frac{\mathbf{G}(t_1,\, t_2,\, i : \mathcal{N}) \quad |i| = 1}{t_1 = t_2 + i}\,\mathrm{Expand}\,\mathbf{G}_{=1}.$$

This completes the set of proof rules used in our implementation of Omega test in CVC. These rules can be thought of as a practical axiomatization of linear arithmetic, where every step in the decision procedure has a corresponding proof rule justifying that step.

7 Conclusion

This paper presents the theory and some implementation detail of an *online* and *proof producing* decision procedure for a theory of mixed-integer linear arithmetic extended with the `int()` predicate. Additionally, the decision procedure supports arbitrary precision arithmetic.

A decision procedure is much more useful to the research community when it can be combined with other decision procedures. Therefore, designing a stand-alone decision procedure is only the very first step in the design process. The next and more difficult task is to enhance the algorithm with additional properties which enable it to communicate with other decision procedures. Namely, the decision procedure must be *online* and *proof producing*, and must support *backtracking*.

In our experience, conceptually the most difficult is the online property. Just adapting the original Omega-test to an online algorithm required significant efforts (before any implementation!). Proof production is the next difficult problem in the design process. It could have easily been the hardest one if CVC did not already have a thoroughly worked-out methodology for adding proof production to existing decision procedures. Nevertheless, the implementation and especially debugging of proof production still presents a challenge. Finally, backtracking is relatively easy to design and implement in the context of CVC, since the framework provides all the necessary data structures.

Since our algorithm is largely based on Omega-test, its performance is comparable with that of the original implementation of Omega-test. The overhead of the CVC framework and the additional requirements on the decision procedure slow it down by about a factor of 10. This is a very reasonable price to pay for having an arithmetic decision procedure be combined with other powerful decision procedures of CVC.

This reimplementation adds arbitrary precision arithmetic, and generally is much more stable than the Omega library code. The arbitrary precision arithmetic is crucial for solving sizable systems of mixed-integer constraints using Fourier-Motzkin approach, since repeatedly generating shadow constraints produces large coefficients even if the coefficients in the original input are relatively small.

Acknowledgements. The authors would like to thank the principal developers of CVC Aaron Stump and Clark Barrett for many insightful discussions about the theory and implementation of this decision procedure.

References

[BDS00] C. Barrett, D. Dill, and A. Stump. A Framework for Cooperating Decision Procedures. In David McAllester, editor, *17th International Conference on Computer Aided Deduction*, volume 1831 of *LNAI*, pages 79–97. Springer-Verlag, 2000.

[BDS02a] C. Barrett, D. Dill, and A. Stump. A Generalization of Shostak's Method for Combining Decision Procedures. In *4th International Workshop on Frontiers of Combining Systems (FroCos)*, 2002.

[BDS02b] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verifi cation*, 2002.

[BT97] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Far90] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–1291, 1990.

[GMP] GMP library for arbitrary precision arithmetic. URL: `http://swox.com/gmp`.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[Möl98] M. Oliver Möller. Solving bit-vector equations - a decision procedure for hardware verification, 1998. Diploma Thesis, available at `http://www.informatik.uni-ulm.de/ki/Bitvector/`.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[RS01] H. Ruess and N. Shankar. Deconstructing Shostak. In *16th IEEE Symposium on Logic in Computer Science*, 2001.

[SBD02] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verifi cation*, 2002.

[SBDL01] A. Stump, C. Barrett, D. Dill, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Computer Society, 2001.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, 1977.

[Sho84] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP – A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, 11 1996.

[Stu02] A. Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002. In preparation: check http://verify.stanford.edu/~stump/ for a draft.

[Wil76] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory (A)*, 21:118–123, 1976.