Integrating the Simplex Algorithm to the Tableau Method

Guillaume Bury*

Ens, Paris, France

The General Context

Automated theorem proving is a fundamental part of program verification, and there are now a lot of automated provers with good capabilities. Handling arithmetic is an important challenge of provers, because of its usage in most programs and theorems, and of the undecidability of Peano arithmetic.

The Research Problem

Pressburger, or linear, arithmetic problems are often encountered in automated proving, whether it be directly for program verification, or asserting coherence of compiler optimizations. There are already many decision procedures for linear arithmetic, some of which integrated to automated theorem provers – the princess prover, for instance, implements the omega procedure – though most of them do not produce proofs In this work I propose to use the general simplex algorithm – a variant of the simplex algorithm for solving linear rational systems (without the optimization problem) – in order to add arithmetic capabilities to the Zenon automated theorem prover. Zenon is a first order automated theorem prover developed at Inria, which uses the tableaux method. The goal was to extend Zenon's calculus so that the linear arithmetic handling interleaves correctly with other parts of Zenon.

Your Contribution

I implemented an extension of Zenon for arithmetic calculus, and extended Zenon's Coq backend to produce correct proofs of arithmetic expressions by the Coq proof assistant. I choose to hide most of the computations done by the simplex, keeping only the core of the explanation returned by the simplex when the given system is unsatisfiable, and only the solution when the system is satisfiable. This led to considering the simplex as a linear system solver black box in some cases, allowing it to be possibly replaced by another algorithm if needed.

Arguments Supporting its Validity

The extended version of Zenon was tested against the TPTP problem library for automated theorem provers, and more particularly, its arithmetic section, called ARI. Problems in ARI cover integer, rational and real arithmetic, and uninterpreted functions and predicates. A large part of these problems were solved, and all the corresponding Coq proofs (except for a few isolated cases) were verified by the Coq compiler.

*Supervised by David Delahaye, Cedric/Cnam/Inria Paris, France

Summary and Future Work

There are still some limitations coming from the simplex. For instance, the simplex does not handle strict inequalities, which is a problem for rational problems. Indeed, for integer problems, a strict inequality can be replaced by an equivalent large inequality (by adding 1 to one side of the inequality), but htat is impossible for rational inequalities. Another limitation if that the simplex cannot do abstract calculations : given the formula $x \leq a$ the simplex cannot solve this system for x (considering a to be a constant). A solution for these limitations would be to use the Fourier-Motzkin projection and/or the omega procedure in addition to the simplex in order to deal with abstract computations.

Furthermore, support of uninterpreted functions and predicates could be added by using a process similar to the purification sometimes used in SMT solvers : replace all expressions within function arguments by variables along with adequate bindings, and then combine a congruence closure algorithm with the arithmetic solver.

Contents

Introduction

Linear arithmetic problems are often encountered in automated proving, whether it be directly for program verification, or asserting coherence of compiler optimizations. Indeed, most programs use built-in arithmetic, and often can be formalized in linear arithmetic. As an example of problems that may be encountered in a compiler, let's consider the following C code :

```
for (i=1;i<=10;i++)
  a[j+i]=a[j];
```

A compiler may be interested in only getting the value of $a[j]$ once before the loop instead of getting it inside the loop each time, since memory access is usually slow, however, to ensure this optimization is safe, the compiler must assert that the value of $a[j]$ does not change within the loop. One way to do so is to prove that there is no index collision in the loop, which can be formalized by proving the following arithmetic formula :

$$\forall i \in \mathbb{Z}, 1 \leq i \leq 10 \Rightarrow j \neq j + 1$$

We worked on adding support for first order linear arithmetic expressions to the Zenon automated theorem prover. Zenon relies on the tableau method, which makes it easy to translate the internal representation of proofs into sequent calculus. We extended Zenon's deduction rules using the simplex algorithm as basis. To do so, we hide most, if not all, the computations done by the simplex, only keeping either the unsatisfiability explanation, or the solution returned, depending on the context. A similar work was done with the omega decision procedure[6] and the automated prover Princess[7].

Two advantages of our approach over simple decision procedures are the interleaving between the arithmetic solving and the logical reasoning of Zenon, allowing to prove more than just pure arithmetic properties, and the proof generated by Zenon (in Coq [9] format) that allows the user to check correctness of the result, while most of the other provers just output the satisfiability of the given formula.

This report will first present the Zenon automated prover and the simplex algorithm, then explain how we dealt with existential quantifiers, and universal quantifiers. Finally, we will present the results of the implementation against the TPTP problem library[8].

1 Zenon

Zenon[1] is an automated first order theorem prover that uses the tableau method. In this section, we will give a quick description of proof search in Zenon.

1.1 The Tableau Method

In the tableau method, starting from the negation of the goal, we apply inference rules to generate a tree. When all branches of the tree are closed, the tree is closed and the tree is a proof the formula at its root. This process produces an MLproof tree, usually presented in a top-down fashion. During the proof search, all formulas encountered are considered to be true, and used to find a contradiction. Intuitively, most MLproof rules can be seen as an implication. Indeed a rule (in top-down presentation) $\frac{\Gamma}{\Gamma'}$ can be seen as $(\Gamma, \Gamma' \vdash \bot) \Rightarrow (\Gamma \vdash \bot)$, and so most of the time, be seen as an implication $\Gamma \Rightarrow \Gamma'$.

The proof search can thus be seen as a process of deducing expressions (according to the MLproof rules) and adding them to a growing environment of expressions, until two contradictory

expressions are found. As such, we will later consider MLproof trees as trees whose nodes are labelled with a set of expressions, which represent the environment.

In order to produce formal proofs in a more usual format, this MLproof tree is then translated into an LLproof tree. The LLproof rules describe a one-side sequent calculus where all sequents are of the form :

$$\frac{\Gamma, \Gamma' \ \vdash \ \bot}{\Gamma \ \vdash \ \bot}$$

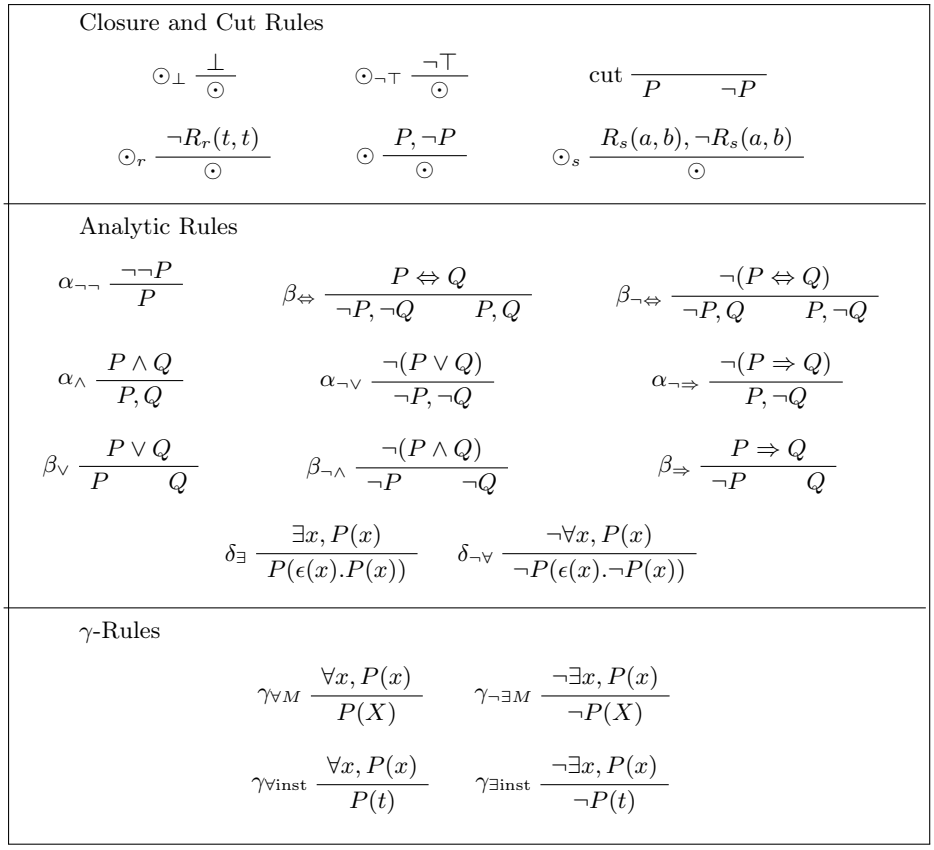


Figure 1: MLproof rules

Figure 1 presents the core of MLproof rules, while the LLproof rules can be found in Appendix A.

1.2 Rules and Quantifiers

Zenon handles quantifiers through the use of metavariables (also called free variables in the literature) for universal quantifiers, and of epsilon terms for existential quantifiers.

Epsilon terms are introduced when Zenon encounters an existential formula (or the negation of a universal formula), and are simply witnesses of the given formula (since in the tableau method, all formulas are considered true).

Metavariables are introduced when Zenon encounters a universal formula (or the negation of an existential formula), and are some kind of wild cards. They can be instantiated with any given expression, and so searching for a good instantiation (i.e one that will create a contradiction with other hypotheses) is one of the main challenges. In the following, we write a variable in lowercase and its metavariables in uppercase to distinguish between them.

Additionally, Zenon has a pruning system, as well as a proof caching system, which we will not describe here, but which is useful in removing unused sections of the tree such as all the nodes pertaining to metavariables before the formulas were instantiated.

For instance, let us consider an untyped example where we have an axiom $\forall x, P(x) \vee Q(x)$ and we want to prove $P(a) \vee Q(a)$ where a is a constant. The search tree would looke like :

$$\alpha_{\neg\vee}\ \frac{\forall x, P(x) \vee Q(x), \neg(P(a) \vee Q(a))}{\gamma_{\forall M}\ \dfrac{\neg P(a), \neg Q(a)}{\beta_{\vee}\ \dfrac{P(X) \vee Q(X)}{P(X) \qquad Q(X)}}}$$

$$\gamma_{\text{inst}}\ \frac{P(a) \vee Q(a)}{\beta_{\vee}\ \dfrac{\odot\ \dfrac{P(a)}{\odot} \qquad \odot\ \dfrac{Q(a)}{\odot}}{}}$$

At which point, Zenon is able to prune the tree by removing the $\gamma_{\forall M}$ node, and get the closed proof tree :

$$\alpha_{\neg\vee}\ \frac{\forall x, P(x) \vee Q(x), \neg(P(a) \vee Q(a))}{\gamma_{\text{inst}}\ \dfrac{\neg P(a), \neg Q(a)}{\beta_{\vee}\ \dfrac{P(a) \vee Q(a)}{\odot\ \dfrac{P(a)}{\odot} \qquad \odot\ \dfrac{Q(a)}{\odot}}}}$$

2 Solving Linear Systems

2.1 Conventions

Here are some naming conventions that we use all along this report :

- Linear arithmetic expressions are built using the addition and multiplication (with the condition, that at least one side of any multiplication is a numeric constant). Subtraction can be seen as syntactic sugar for addition together with multiplication by a negative constant.
- An arithmetic formula is a comparison of two linear arithmetic expressions, for instance $2x + 1 < 7 - \frac{1}{2}y$. There are 5 comparison operators that we consider : $=, <, >, \leq, \geq$.
- An arbitrary comparison operator distinct from the equality may be written $\bowtie$, and its negation $\overline{\bowtie}$, as follows :

$\bowtie$	$\overline{\bowtie}$
$<$	$\geq$
$\leq$	$>$
$>$	$\leq$
$\geq$	$<$

- An expression is a logical proposition that may use arithmetic expressions, for instance :

$$\forall i \in \mathbb{Z}, 1 \leq i \wedge i \leq 10 \Rightarrow j \neq j + 1$$

- The notation $e \neq e'$ is syntactic sugar for $\neg (e = e')$.

2.2 The General Simplex

The general simplex as described in [4, ch 5.2] is a variant of the simplex algorithm, designed to solve the satisfiability problem on linear systems, rather than the optimization of a given objective function under a system of constraints. Given a linear rational system of constraints, the general simplex either provides a solution of the system (i.e an assignment of all the variables such that every constraint is respected), or returns an unsatisfiability certificate for the system (see 2.4), i.e a linear combination from which a contradiction almost immediately follows.

The general simplex accepts only two forms of constraints :

1. Equations of the form : $v = \sum_i a_i x_i$
2. Bounds on variables : $l_i \leq v \leq u_i$

Where the coefficients a_i are in $\mathbb{Q}$, and the bounds l_i, u_i in $\mathbb{Q} \cup \{-\infty, +\infty\}$. A system that contains only formulas of either of the above forms is said to be in general form.

This representation does not restrict expressivity, given that any linear system can be translated to this representation. To do so, two transformations are required :

1. Any equality $e = e'$ is replaced by $e \leq e' \wedge e' \leq e$
2. Any comparison $e \bowtie e'$ is rewritten as $f \bowtie k$ where f is a sum of variables with coefficients and k a numeric constant[1], such that $e - e' = f - k$. The comparison can then be replaced by $x = f \wedge x \bowtie k$, with x a fresh variable.

This transformation to general form adds an interesting property to the system : all variables on the left-hand side of equalities do not appear on the right-hand side of equalities. From now on, we suppose that all general forms satisfy this property i.e we only consider general forms that come from the application of the process described above to a linear systems.

Internal state. The general simplex algorithm maintains an internal state which consists of :

- A set of variables called **basic** variables which represents the left-hand side variables.
- A set of variables called **non-basic** variables which represents the right-hand side variables.
- A matrix, which is a representation of the basic variables as linear functions of the non-basic variables.
- A set of of inequalities on **basic** and **non-basic** variables
- An assignment α of the non-basic variables (a map from the non-basic variables to rationals).

[1] If f is the empty sum, the comparison is either trivially false in which case the system if unsatisfiable, or a tautology in which case it is useless.

In the internal state, the set of inequalities is set at the beginning and will not change during execution of the general simplex, while all other elements of the internal state are mutable. Also, notice that from the assignment of the non-basic variables, one can deduce the complete assignment of all the variables by using the tableau (when considering the value of a given variable at a certain point in the algorithm, we refer to the value given to the variable v in the full assignment by $\alpha(v)$).

Given a linear system S in general form, the simplex algorithm works by iterating the same steps, updating its internal state each time, until it finds a contradiction, or the full assignment deduced from its internal state verifies all the inequalities in its internal state. Furthermore, between each step, a number of invariants are verified :

In-1 The basic and non-basic variables are a partition of the set of variables occurring in S.

In-2 The matrix together with the inequalities represents a linear system logically equivalent to S.

In-3 The assignment value of all non-basic variables is within the bounds of these variable (as constrained by the inequalities).

Initialization. From a given linear system in general form, we can build the initial simplex state as follows :

1. Set the set of basic variables to be the set of variables that appear on the left-hand side of equalities.
2. Set the set of non-basic variables to be the set of all variables that appear on the right-hand side of equalities.
3. The matrix directly follows from the set of equalities in the general form.
4. The set of inequalities is the same as that of the general form.
5. Set the assignment to assign all non-basic variables to 0 if it is within the bounds of the variable, and to the bound that is the closest to 0 otherwise.

For instance, let us consider the linear system to be translated in general form :

$$\begin{cases} x + y & \geq 2 \\ 2 * x - y & \geq 0 \\ -x + 2 * y & \geq 1 \end{cases}$$

During the translation to the general form of this system, we introduce variables s_1, s_2, s_3, and we get the following matrix and its corresponding linear system :

	x	y
s_1	1	1
s_2	2	-1
s_3	-1	2

$$\begin{cases} s_1 = & 1 * x + 1 * y \\ s_2 = & 2 * x - 1 * y \\ s_3 = & -1 * x + 2 * y \end{cases}$$

together with the bounds : $2 \leq s_1 \leq +\infty$, $0 \leq s_2 \leq +\infty$ and $1 \leq s_3 \leq +\infty$.

Pivot The pivot operation on two variables x (basic) and y (non-basic), switches x and y so that x is now a non-basic variable and y a basic variable. To do so, it performs a usual matrix pivot operation on the row of x and column of y in the matrix, then replaces y with x in the set of non-basic variables (and vice-versa in the basic variables), and finally replace in the assignment the binding of y with a binding from x to its value before switching.

For instance, to pivot s_1 and x in the previous example, we need first to express x as a function of s_1 and y, which gives the first row of the new matrix : $x = s_1 - y$. We can then substitute x by $s_1 - y$ in the other rows, and we get the following matrix :

	s_1	y
x	1	-1
s_2	2	-3
s_3	-1	3

Algorithm The simplex algorithm is then as follows :

1. Decide of an arbitrary order on the variables
2. Look for a basic variable x whose value in the current assignment does not respect one of its bounds b. If none exists, the current assignment is a solution.
3. Try to find the smallest suitable (see definition below) non-basic variable y for pivoting. If none exists, the system is unsatisfiable.
4. Do the pivot operation on x and y, set the value of x to b
5. Go to 2

Given a variable x whose value $\alpha(x)$ in the current assignment does not respect one of its bound, for instance its lower bound l_x, and with a tableau expression $x = \sum_i a_i y_i$, a suitable variable is a y_j such that :

- $a_j > 0$ and the current assignment of y_j is strictly lower than its upper bound u_{y_j}
- $a_j < 0$ and the current assignment of y_j is strictly higher than its lower bound l_{y_j}

With a symmetric case for when variable x does not respect its upper bound.

The termination of the general simplex algorithm is ensured by Bland's rule : by choosing the smallest suitable variable (according to the ordering decided at the beginning), we ensure no set of basic variables is repeated. See [2] for a detailed proof. Since the algorithm always terminates and returns the status of the problem (either unsatisfiable or satisfiable), the simplex is a decision procedure for the satisfiability of rational linear systems.

There exists some optimizations for the simplex, such as gomory cuts [3], but we did not implement them in order to keep the unsatisfiability explanation generated as simple as possible.

2.3 Example

Let us look at the system : $x + y \geq 2 \wedge 2x - y \geq 0 \wedge -x + 2y \geq 1$ introduced above. The simplex initial state for this system is :

	x	y
s_1	1	1
s_2	2	-1
s_3	-1	2

$$\begin{cases} x \mapsto 0 \\ y \mapsto 0 \\ s_1 \geq 2 \\ s_2 \geq 0 \\ s_3 \geq 3 \end{cases}$$

We decide of the arbitrary order on the variables : $x < y < s_1 < s_2 < s_3$. In the initial assignment, $\alpha(s_1) = 0$ and so s_1 does not respects its bounds. Since x is suitable and is the smallest variable (according to our arbitrary ordering of the variables), we pivot s_1 and x, and assign 2 to s_1 (which is now a non-basic variable) so that it respects its bounds. We get the following state :

	s_1	y
x	1	-1
s_2	2	-3
s_3	-1	3

$$\begin{cases} s_1 \mapsto 2 \\ y \mapsto 0 \\ s_1 \geq 2 \\ s_2 \geq 0 \\ s_3 \geq 3 \end{cases}$$

Now, s_3 does not respect its bounds, so we pivot it with y, which is suitable, assign 3 to it and we get the following state :

	s_1	s_3
x	$\frac{2}{3}$	$-\frac{1}{3}$
s_2	1	-1
y	$\frac{1}{3}$	$\frac{1}{3}$

$$\begin{cases} s_1 \mapsto 2 \\ s_3 \mapsto 3 \\ s_1 \geq 2 \\ s_2 \geq 0 \\ s_3 \geq 3 \end{cases}$$

We then pivot s_2 with s_1 and map s_2 to 0 :

	s_2	s_3
x	$\frac{2}{3}$	$\frac{1}{3}$
s_1	1	1
y	$\frac{1}{3}$	$\frac{2}{3}$

$$\begin{cases} s_2 \mapsto 0 \\ s_3 \mapsto 3 \\ s_1 \geq 2 \\ s_2 \geq 0 \\ s_3 \geq 3 \end{cases}$$

The assignment now satisfies all inequalities and we have a solution : $x \mapsto 1, y \mapsto 2$.

2.4 Generating Explanations

When the simplex returns a solution, it is easy to see that it is correct solution to the linear system. Let us explain how we can justify the unsatisfiable status of the linear system when the simplex returns the unsatisfiable statement.

The simplex returns an unsatisfiable statement when in presence of a basic variable x whose value $\alpha(x)$ in the current assignment does not respect one of its bound, for instance its lower bound l_x, meaning that $\alpha(x) < l_x$, and that there is no suitable variable for pivoting. If the expression of x is : $x = \sum_i a_i y_i$, then that means that for every non-basic y_i :

- if $a_i > 0$, then $\alpha(y_i) \geq u_{y_i}$, but since y_i is a basic variable, it must respect its bound, and so $\alpha(y_i) = u_{y_i}$
- if $a_i < 0$, then $\alpha(y_i) \leq l_{y_i}$, but since y_i is a basic variable, it must respect its bound, and so $\alpha(y_i) = l_{y_i}$
- A variable without bounds ($l_{y_i} = -\infty$, $u_{y_i} = +\infty$) is always suitable for pivoting.

So, by partitioning the indices in two sets $\mathbb{N}^+ = \{i | a_i > 0\}$ and $\mathbb{N}^- = \{i | a_i < 0\}$, we have that :

$$\begin{aligned} \alpha(x) &= \sum_{i \in \mathbb{N}^+} a_i u_{y_i} + \sum_{i \in \mathbb{N}^-} a_i l_{y_i} \\ \alpha(x) - x &= \sum_{i \in \mathbb{N}^+} a_i (u_{y_i} - y_i) + \sum_{i \in \mathbb{N}^-} a_i (l_{y_i} - y_i) \\ \alpha(x) - x &\geq 0 \end{aligned}$$

From that we deduce that : $x \leq \alpha(x) < l_x$, and so $x < l_x$, which is clearly absurd since we must have that $x \geq l_x$. We will see in Section 3.1 how to formalize these explanations as MLproof rules during the proof search.

2.5 Integer Problems

So far, we have described a decision procedure for rational linear systems. In order to solve integer linear systems, we need to use a strategy, called branch-and-bound.

An integer linear system can be seen as a rational system where all variables are required to have an integer value. For that reason, we can accept rational coefficient in the system : given a constraint with rational coefficients, we multiply it by the greatest common divisor of the denominators of the coefficient in order to get an equivalent constraint.

We call relaxed system of S, and we write relaxed(S), the system S without the condition that the variables must have an integer assignment.

Branch-and-bound The branch-and-bound algorithm for a linear system S is the following :

- Run the simplex algorithm on relaxed(S).
  - If the system is unsatisfiable, return false
  - If the system has a solution :
    - If a non-integer value v is assigned to a variable x, call the branch-and-bound twice, with the systems, $S \cup \{x \leq \lfloor v \rfloor\}$ and $S \cup \{x \geq \lfloor v \rfloor + 1\}$. Return the disjunction of the two return values.
    - If all the variables have an integer assignment, return true.

The choice over which variable to branch does not need to be specified, any variable (whose value in the current assignment is not an integer) can be chosen. However, in our implementation of the branch-and-bound algorithm, we chose to branch over the non-basic variables of the initial problem first, so that the unsatisfiability explanation is easier to understand.

This algorithm is not complete : if we consider the system $1 \leq 3x + 3y \leq 2$, the branch-and bound will loop forever. That will happen for any system with unbounded rational solutions but no integer solution. To solve this problem, we can add a global bounds on the variables at the beginning of the problem.

To ensure termination of the branch-and-bound, we use global bounds found in [5, I.5.4]. Given an $m \times n$ rational matrix $A = (a_i)$ and a vector $b \in \mathbb{Q}^m$, let us call $P = \{x \in \mathbb{R} | Ax \leq b\}$ the set of real solution, then if the set $S = P \cup \mathbb{Z}^n$ of integer solutions is non-empty, then there

exists an integer solution $x \in S$ such that $|x_j| \leq \omega_{A,b}$ for all $1 \leq j \leq n$, with $\omega_{A,b} = \left(2n'^2\theta\right)^{n'}$ where $n' = \max(n, m)$ and $\theta = \max_{ij}(|a_{ij}|)$.

For the system $1 \leq 3x + 3y \leq 2$, the global bound evaluates to $\omega = 576$, while it is $\omega' = 157464$ for the system $1 \leq 3x + 3y + 3z \leq 2$, in which case it is not usable in practice.

2.6 Incrementality

An interesting feature of the general-simplex algorithm, and of the branch-and-bound as well (although a little less), is that both algorithms are incremental. For the simplex, adding an inequality is quite simple :

1. if it is an inequality on a variable, simply add it to the set of inequalities of the internal state of the simplex, and eventually adjust the value of the variable if the variable is non-basic.

2. else, introduce a new basic variable, and add the corresponding equation to the matrix, then add the bound for the newly created variable.

Since multiple additions of variable inequalities may lead to incoherent bounds for a variable, we add a preliminary step to the simplex algorithm that checks the coherence of variables. This allows for very practical use of the simplex, most notably, trying to solve each system after adding an inequality, while other decision procedures such as omega [6] cannot make use of the work done on a subsystem to solve a larger system.

3 Unsat Systems

In this section, we describe how to detect arithmetic contradictions, i.e unsatisfiable systems, and the associated MLproof rules.

3.1 Inference Rules

We present in Fig. 2 the inference rules used to explicit arithmetic contradictions in the MLproof tree, with the following notations :

- A system of constraints may be written with a matrix in order to be clearer : for instance $Ax \leq b$ denotes the system of inequalities $\sum_j a_{i,j}x_j \leq b_i$ for $1 \leq i \leq n$, with A a $n \times m$ rational matrix b an n rational vector, and x an m vector of variables.

- We write $\{\Gamma\}$ for the container that represents the system Γ (see definition below).

The main challenge of these rules if that, in order to benefit from the global bound, one needs to be careful. Indeed, if we use an inequality deduced from the global bound, together with an inequality that is not included in the system that was used to deduce the global bound used, it may lead to false contradiction. For instance, consider the satisfiable system $x \geq 1 \wedge x \geq 3$. The global bound for the subsystem $x \geq 1$, is $\omega = 2$, and a solution such that $|x| \leq 2$ can be found, for instance $x \mapsto 1$. However, the bounds $-2 \leq x \leq 2$, may create a contradiction with the constraint $x \geq 3$ and result in an unsatisfiable system although the initial system was satisfiable.

To avoid using the global bound of a system with constraints outside of the system, we introduce a notion of container : a container is actually a set of constraints, seen as a single

Constant Rules

$$\text{Const}\ \frac{a \bowtie b}{\odot} \qquad \text{Const}\ \frac{a = b}{\odot} \qquad a \bowtie b \text{ (or } a = b\text{) is a trivially false comparison of contants}$$

Normalization rules

$$\text{Eq}\ \frac{e = e'}{e \leq e', e' \leq e} \qquad \text{Neq}\ \frac{e \neq e'}{e < e' \qquad e > e'} \qquad \text{Neg}\ \frac{\neg e \bowtie e'}{e \overline{\bowtie} e'}$$

$$\text{Int-Lt}\ \frac{e < f}{e \leq f - 1} \qquad \text{Int-Gt}\ \frac{e > f}{e \geq f + 1} \qquad e \text{ and } f \text{ are integer expressions}$$

Simplex rules

$$\text{Global}\ \frac{Ax \leq b}{\{Ax - s = 0, s \leq b, -\omega_{A,b} \leq x \leq \omega_{A,b}\}}\ s \text{ fresh}$$

$$\text{Branch}\ \frac{\{\Gamma\}}{\{\Gamma, x \leq k\} \qquad \{\Gamma, x \geq k+1\}}\ x \in \text{var}(\Gamma), k \in \mathbb{Z}$$

$$\text{Conflict}\ \frac{\{\Gamma', x \leq k, x \geq k'\}}{\odot}\ k < k' \text{ numeric constants}$$

$$\text{Simplex-lin}\ \frac{\{\Gamma, e_1 = 0, \ldots, e_n = 0\}}{\left\{\Gamma, e_1 = 0, \ldots, e_n = 0, \sum_{i=1}^{n} a_i e_i = 0\right\}}\ \forall i, a_i \in \mathbb{Q}$$

$$\text{Leq}\ \frac{\left\{\Gamma = \left(\Gamma', x_j \leq u_j | j \in N^+, x_j \geq l_j | j \in N^-, x = \sum_{j \in N^+ \cup N^-} a_j x_j\right)\right\}}{\left\{\Gamma, x \leq \sum_{j \in N^+} a_j u_j + \sum_{j \in N^-} a_j l_j\right\}}\ \begin{pmatrix} a_j > 0, j \in N^+ \\ a_j < 0, j \in N^- \end{pmatrix}$$

$$\text{Geq}\ \frac{\left\{\Gamma = \left(\Gamma', x_j \geq l_j | j \in N^+, x_j \leq u_j | j \in N^-, x = \sum_{j \in N^+ \cup N^-} a_j x_j\right)\right\}}{\left\{\Gamma, x \geq \sum_{j \in N^+} a_j l_j + \sum_{j \in N^-} a_j u_j\right\}}\ \begin{pmatrix} a_j > 0, j \in N^+ \\ a_j < 0, j \in N^- \end{pmatrix}$$

Figure 2: Simplex Rules for Unsatisfiability

object in the tableau rules. The global bounds for the system will be added during the creation of the container, and other rules will add to the container constraints deduced from the formulas already present within the container, until a contradiction is reached and the branch can be closed. That way, the global bounds of a linear system cannot interact with constraints outside of the system. The container is a way to restrict the environment considered when applying a rule.

Implicit rewriting As they are, these rules may appear strange : for instance, the only equalities introduced are of the form $e = 0$ but we also use equalities of the form $x = e'$ in the rules, which may seem absurd since there can be no formula syntactically equal to it in

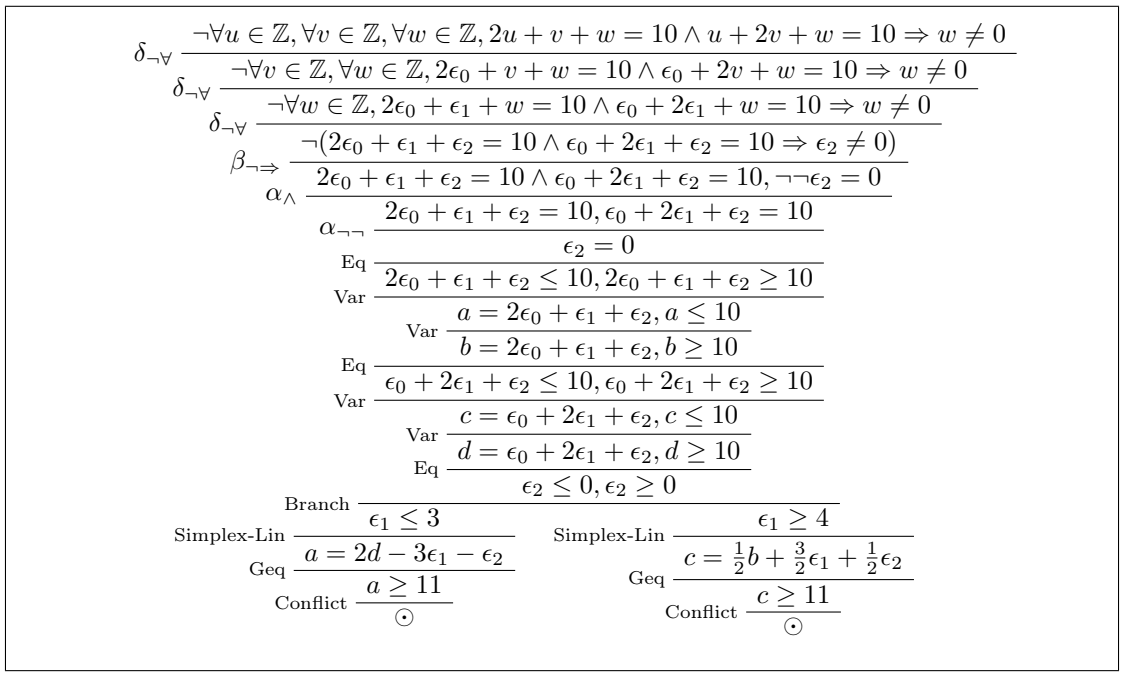


Figure 3: MLproof search tree example for problem ARI178=1.p

a container. In order to keep the proof tree simple we choose not to represent the calculus steps and only keep the logically meaningful transformations. In that setting, the rules are to be understood modulo simple rewriting (factorizing coefficients of variables and computing the result of constant expressions), much like what the 'ring_simplify' Coq tactic does.

While these rules allow for completeness, in practice the global bound is too high to be of much use on any system with at least 3 constraints (or at least 3 variables). In practice, we use a different version of these rules, without the use of containers, since we do not use the global bounds. In the following, we will consider the rules without containers which have been implemented in Zenon. Additionally, the rule Global is now replaced by a local introduction of variable :

$$\text{Var}\ \frac{e \bowtie c}{s = e, s \bowtie c}\ s \text{ fresh}$$

Where c is a numeric constant and e a non-empty sum of variables.

For instance let's consider the following formula[2] :

$$\forall u \in \mathbb{Z}, \forall v \in \mathbb{Z}, \forall w \in \mathbb{Z}, 2u + v + w = 10 \wedge u = 2v + w = 10 \Rightarrow w \neq 0$$

We can derive the MLproof search tree in Fig. 3 from its negation.

3.2 Implementation in Zenon

There are two inference rules that need parameters and thus require the prover to carefully choose those parameters : Branch and Simplex-lin. We describe here how Zenon uses the branch-and-bound algorithm to make correct a use of these rules.

[2]It is the goal of the ARI178=1.p problem found in the ARI section of the TPTP library.

Each time Zenon encounters a formula that it has not seen yet, there are two cases. Either the formula is a bound on a variable, in which case it is simply added to the current simplex state. Or the rule Var is applied and a new variable generated. Additionally, each time a new variable is added to the simplex state, Zenon tries to solve the system present in the simplex state. If this yields an unsatisfiable statement, along with an explanation tree, then the explanation if translated into MLproof rules and introduced in the proof tree, effectively closing the current branch.

In order to translate the explanations given by the simplex into MLproof rules, we use the rule Simplex-lin in order to get the expression used by the simplex in its explanation. We then use either Leq or Geq depending on which one can be used. We can finally use the Conflict rule to close the tree, since the simplex guarantees that the bound we just deduced will conflict with another already present.

Thanks to the fact that the simplex is incremental, we have persistent simplex state that allows to keep all the work already done up to a point when the proof tree branches.

4 Finding Instantiations

In order to find instantiations (for universal quantifiers) that lead to contradictions, we try to find an instantiation of the variables that satisfy a carefully selected set of formulas. To do so, we introduce the notion of cover for trees.

4.1 Arithmetic Constraint Trees

First, we define a few notions that relate to trees of formulas that we will use later to find correct instantiation values for metavariables.

Definition An arithmetic constraint tree is a tree labelled with sets of arithmetic formulas.

In the following, trees will refer to arithmetic constrain trees.

Definition Given a tree $\mathcal{T}$ labelled with set of formulas, and a set of formula $\mathcal{E}$, the set of nodes of $\mathcal{T}$ covered by $\mathcal{E}$ is the least set of nodes n such that :

- Either $\text{label}(n) \cap \mathcal{E} \neq \emptyset$ (we say the node is directly covered)
- Or all children of n are covered by $\mathcal{E}$

Definition A set of formulas $\mathcal{E}$ is said to cover a tree $\mathcal{T}$ if and only if the root of $\mathcal{T}$ belongs to the set of nodes covered by $\mathcal{E}$.

Definition A counter-example of a given tree $\mathcal{T}$ is an assignment of the variables of the formulas of $\mathcal{T}$ such that there exists a set $\mathcal{E}$ that covers $\mathcal{T}$ and that in the assignment, all the negation of the formulas in $\mathcal{E}$ are satisfied.

In order to find a counter-example of a given arithmetic constrain tree $\mathcal{T}$, we simply need to solve the negation of a system (a set of formulas) that cover $\mathcal{T}$. To do so, we enumerate all systems that covers $\mathcal{T}$ and try to solve each of them until we find a counter-example. We can enumerate a sufficient set of covering sets with the following formula :

$$\text{Cover}(\mathcal{T}) = \{\{f\} | f \in \text{label}(\mathcal{T})\} \cup \{ \bigcup_{1 \leq i \leq n} s_i | s_i \in \text{Cover}(\mathcal{T}.[i])\}$$

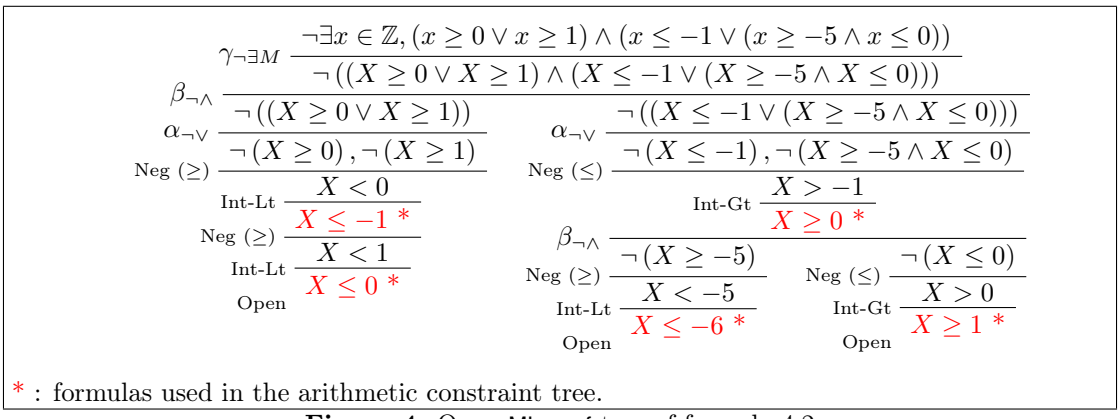


Figure 4: Open MLproof tree of formula 4.2

With label($\mathcal{T}$), the label of the root of $\mathcal{T}$, and $\mathcal{T}.[i]$ the i-th children of the root of $\mathcal{T}$.

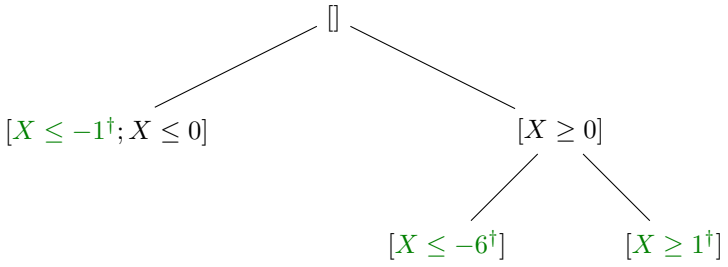
4.2 Interleaving with Zenon

Now, we can make the link with the proof search in Zenon. As stated before, a MLproof tree can be seen as a tree labelled with sets of expressions. To make use of that tree to find instantiations, we have to allow Zenon to return a tree with open branches (actually, these branches are closed by a dummy node), because otherwise Zenon only returns when it finds a closed tree, which is already a proof. Then we filter all the expressions in the tree and keep only the arithmetic constraints. We can finally find a counter-example of the tree, and re-try to prove the formula, this time directly instantiating quantified variables to the values of the counter-example.

Let us see what happens on an example. If we want to prove the formula :

$$\exists x \in \mathbb{Z}, (x \geq 0 \vee x \geq 1) \wedge (x \leq -1 \vee (x \geq -5 \wedge x \leq 0))$$

We first take its negation, then decompose it using the MLproof rules of Zenon, and we get the MLproof tree shown in Fig. 4. From that tree, we can get the following arithmetic constraint tree by keeping the formulas marked in red (followed by an asterisk) in the figure, and collapsing empty nodes :



During enumeration of the covering sets, we get for example to the formulas in green (followed by a † in the arithmetic constraint tree) which form a set that covers the tree, and we can thus try to solve the negation of these formulas, which gives the system :

$$\begin{cases} X \leq -1 \\ X \leq -6 \\ X \geq 1 \end{cases} \Longrightarrow \begin{cases} X \geq 0 \\ X \geq -5 \\ X \leq 0 \end{cases}$$

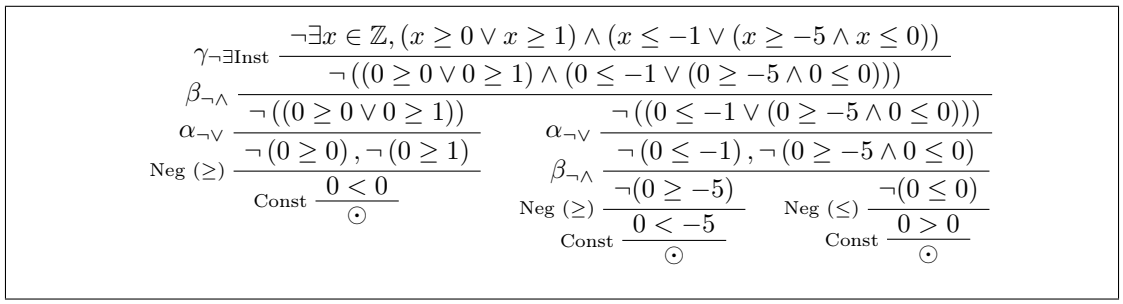


Figure 5: Closed MLproof tree of formula 4.2

Which yields the counter-example : $X \mapsto 0$. We can then rebuild the MLproof tree, this time instantiating x with 0, which allows Zenon to close all the branches of the proof tree. We get the proof tree in Fig. 5.

Zenon now alternates between normal MLproof search with its usual rules, and arithmetic solving on the open tree, remembering the counter-example it finds as hints to instantiating for the next round.

4.3 Limitations of the Simplex

As mentioned previously, the simplex algorithm (and by extension, the branch-and-bound algorithm), has a few limitations.

First, the branch-and-bound is not complete since we abandoned the global bound. However, since it only affects the cases where there are unbounded rational solutions but no integer solutions, it does not affect the search for counter-examples.

The main limitation of the simplex is that it is not able to do abstract computations. More specifically, every variable in formulas is treated in the same way in the simplex algorithm, particularly, the simplex can assign to it any value. Additionally, the simplex always returns assignment from variables to numeric values. This become problematic when there are more than only metavariables in the formulas. For instance, let us suppose we are trying to prove the following formula : $\exists x \in \mathbb{Q}, x \leq a$, where a is a constant (an implicit variable). The problem is that we cannot send the expression $X \leq a$ to the simplex because in this context X and a are fundamentally different : we cannot change the value of a, while we can freely choose the value of X, but the simplex is not able to make that difference. For the simplex, every variable has the same status, therefore it can not solve system with abstract variables. This also prevents from solving system with epsilon variables coming from existentially quantified variables, thus preventing Zenon to solve problems when there is an alternation of quantifiers.

5 Results

5.1 Optimizations

A few other rules have been implemented, in addition to the calculus rules described in Fig 2, in order to alleviate some drawbacks of the simplex algorithm.

First, we have introduced two rules that tightens inequalities when possible. In our calculus, we consider everything to be rational, seeing integer variables as rational variables whose value is constrained to be an integer. As such, we can encounter inequalities such as $x \leq \frac{1}{2}$ where

x is an integer variable. In that case, we would like to deduce the inequality $x \leq 0$. We thus introduce two new calculus rules, with k a non-integral numeric constant, and x an integer variable :

$$\text{Tighten-Leq}\ \frac{x \bowtie k}{x \leq \lfloor k \rfloor} \bowtie \in \{<, \leq\} \quad \text{Tighten-Geq}\ \frac{x \bowtie k}{x \geq \lceil k \rceil} \bowtie \in \{>, \geq\}$$

These two rules are particularly efficient when used together with formula normalization : when rewriting formulas, we try and put them in a normal form, i.e a formula which respects these properties :

- The right side of the comparison operator is a numeric constant
- The left side is a sum of coefficients variables (if there are no variables, the left side is the empty sum, which is 0)
- All the coefficients of the variables are integers, and the greatest common divisor of the coefficients is 1.

Together, these two features allow us to make small reasonings about divisibility, for instance the problem $1 \leq 3x + 3y \leq 2$ will be rewritten in its normal form to a system equivalent to : $\frac{1}{3} \leq x + y \leq \frac{2}{3}$, and then after tightening will yield the system : $1 \leq x + y \leq 0$, which is trivially false.

Additionally, as mentioned Section 2.5, we chose to branch first on the non-basic variables of the initial system during the branch-and-bound algorithm. Since normalization of arithmetic expressions always gives constrains with integer coefficients, the initial tableau of the branch-and-bound only has integer coefficients, so if all non-basic variables have an integer assignment, then the values of the basic variables in that assignment must also be integers. We can then branch only on the non-basic variables of the initial system during our branch-and-bound.

The other optimization is to use Fourier-Motzkin projection on all strict rational inequalities encountered, since those are not taken into account by the simplex. However, integrating it into the dynamic proof search of Zenon requires some changes : instead of taking a full and static system and then eliminating variables one by one, we have to compute projections whenever we encounter a new equation during the proof search. To ensure termination, or rather that only a finite number of formulas is generated, we keep trace of all previous equations encountered and then only project the smallest variable (according to an arbitrary order on variables) of the new formula with the other formulas seen before that contain that particular variable and no smaller variable. This guarantees that the smallest variable in the formulas generated is strictly greater than the variable that we have eliminated.

Fourier-Motzkin projection is mainly used on the strict inequalities produced by the application of the Neq rule. An example of the use of Fourier-Motzkin projection is shown on Fig.6. Both times, the application of the Fourier-Motzkin projection eliminates the variable ϵ_0. The application of the rule on the left branch uses the two constraints : $-\epsilon \leq -10$ and $\epsilon_0 < \frac{10}{1}$, to deduce $10 < \frac{10}{1}$, whose normal form is $0 < 0$.

5.2 Testing over TPTP Problems

The TPTP (Thousands of Problems for Theorem Provers) library[8] contains a section dedicated to arithmetic problems, named ARI. Problems use integer, rational, and real arithmetic, uninterpreted functions and predicates, and first-order logic. Their complexity ranges from simple comparison of numeric constants to complex properties about uninterpreted functions, such as proving 8 is a power of 2, or that 5 is not.

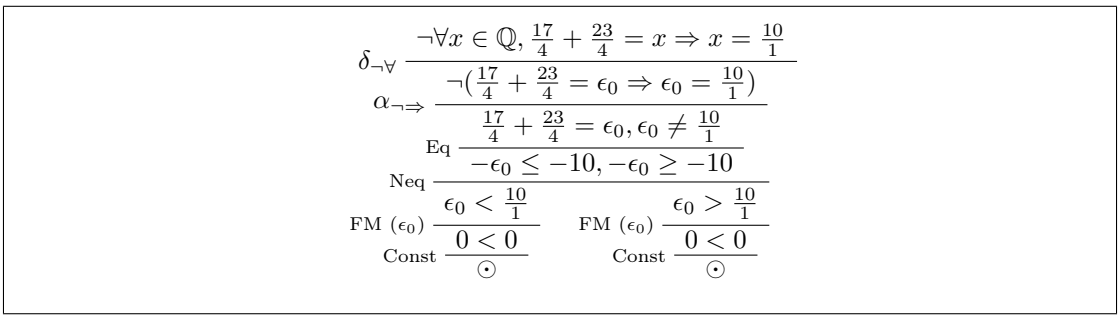


Figure 6: MLproof search tree example for problem ARI257=1.p

Proved	Solved	Problems
302	312	388

Figure 7: Tests Results for Zenon with Arithmetic

Fig. 7 presents the results of the extended version of Zenon on the part of ARI concerning integer and rational arithmetic (and not real arithmetic). The solved problems are the ones for which Zenon has found a proof, while the proved ones are those for which the Coq proof that Zenon outputs has been checked by the Coq proof assistant.

Among the problems not solved by Zenon are problems that uses built-in functions of the TPTP language not yet supported, such as the $to_int function which translates a rational (or a real) to the largest integer lesser or equal to the argument. Other problems are not solved because of an alternation of quantifiers, or because uninterpreted functions and predicates are not yet fully supported. Finally a few problems can not be solved by the simplex because they use strict inequalities : for instance $\exists x \in \mathbb{Q}, \exists y \in \mathbb{Q}, x < y$ is not solved.

Sources for Zenon extended with arithmetic can be found at `http://gbury.eu/public/zenon-arith.tar.gz`. The archive also contains the ARI section of the TPTP library and includes a small script to automatically call Zenon on a list of problems.

5.3 Coq Backend

The main challenge of the Coq backend for Zenon was to lightly integrate the implicit rewriting of arithmetic formulas into the Coq proofs without having to explicit all the rewriting, which would have led to unnecessarily long and verbose proofs.

To do so, we chose to use cuts to introduce the rewritten formulas and then proving the coherence of the rewriting using a normalization tactic. The normalization tactic puts all formulas in the form : $e \bowtie 0$ for comparisons, and or $e = 0$ for equalities, and then uses the ring_simplify tactic to normalize the expression e. Other lemmas such as distributivity over division are also needed since the ring_simplify tactic uses only ring operation rewriting while $\mathbb{Q}$ is also a field.

Only a few proofs are not validated by Coq, mainly for technical reasons. There are three reasons that explain why 10 of the Coq proofs generated by Zenon are invalid :

- 7 proofs use functions and predicates that are not yet correctly translated from Zenon's LLproof to Coq proofs (such as the $is_int predicate, which returns true if a rational is also an integer).

- Zenon automatic factorization of proofs using lemmas incorrectly identifies the variables to be quantified in some of the lemmas, leading to wrongly specified lemmas. This happens in 2 proofs.
- Zenon proofs use the congruence Coq tactic, which works only with the built-in equality of Coq (Leibniz's equality) which is different from the equality over rationals, and the tactic therefore fails in one proof.

Conclusion

The simplex algorithm was successfully integrated to the tableaux method and implemented in Zenon, which is now able to reason about linear rational and integer arithmetic. Additionally, almost all proofs can be checked by the Coq proof assistant, which is an advantage when comparing to other provers (like SMT solvers dealing with arithmetic but without producing any proof). The results that we get from our naive implementation are satisfactory, and can still be greatly improved. For instance, we could add gomory cuts to our implementation of the simplex algorithm, and adapt the MLproof inference rules to take them into account.

The main improvement that could be made would be to use both the simplex and the omega procedure. Indeed, the simplex is able to detect unsatisfiable systems, even when it requires complex and abstract reasoning, however, it is not the case when trying to find instantiations, because it is designed as an algorithm for solving ground linear systems. The Omega procedure works by eliminating variables, each time producing a logically equivalent system with one less variable than the previous. Using omega to simplify the system until the simplex can solve it would allow us to overcome most of the limitations that we encountered when dealing with metavariables.

References

[1] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), October 2007. Springer.

[2] Bruno Dutertre and Leonardo de Moura. Integrating simplex with DPLL(T). Technical report, SRI International, 2006.

[3] Ralph Gomory. An algorithm for integer solutions to linear problems. In *Recent Advances in Mathematical Programming*, pages 269–302, 1963.

[4] Daniel Kroening and Ofer Strichman. *Decision Procedures, An Algorithmic Point of View*. Springer, 2008.

[5] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[6] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *CACM*, 35(8):102–114, August 1992.

[7] Philipp Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. 2008.

[8] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning (JAR)*, 43(4):337–362, December 2009.

[9] The Coq Development Team. Coq, *version 8.1*. INRIA, November 2006.
Available at: `http://coq.inria.fr/`.

A LLproof rules

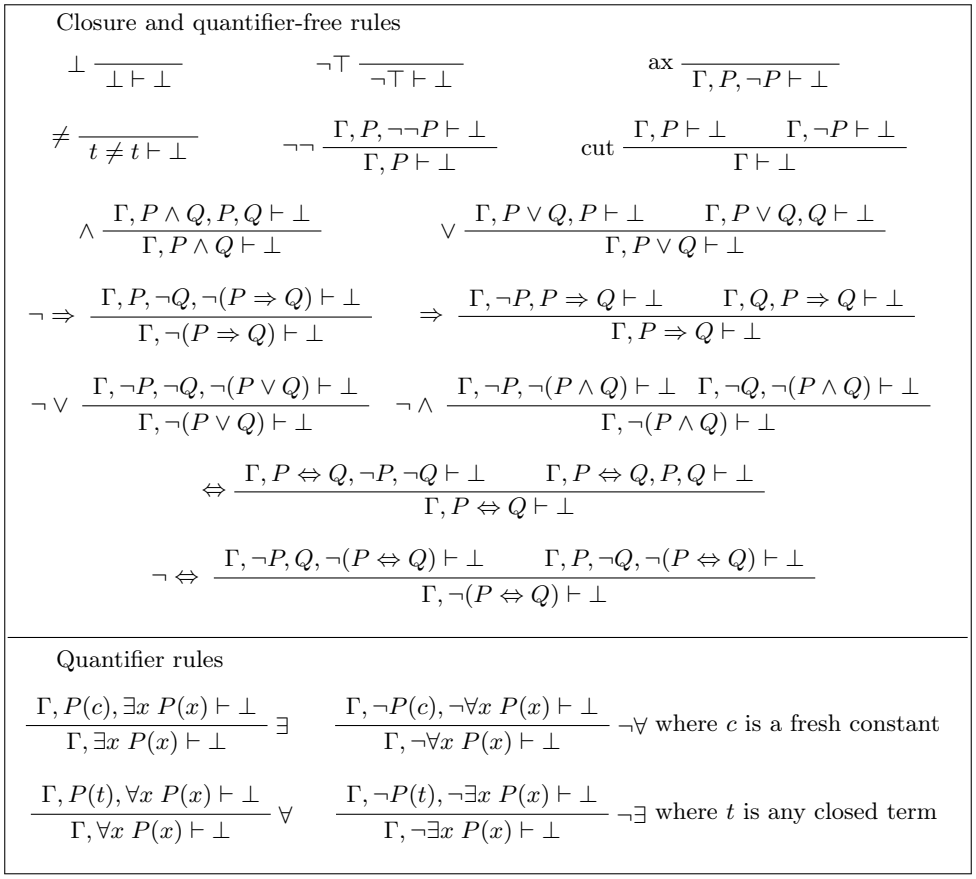


Figure 8: LLproof rules