Formalization of Incremental Simplex Algorithm by Stepwise Refinement

Mirko Spasić and Filip Marić

Faculty of Mathematics, University of Belgrade

Abstract. We present an Isabelle/HOL formalization and total correctness proof for incremental version of Simplex algorithm which is used in most state-of-the-art SMT solvers. Formalization relies on stepwise program and data refinement, starting from a simple specification, going trough a number of fine refinement steps, and ending up in a fully executable functional implementation. Symmetries present in the algorithm are handled with special care.

1 Introduction

Linear arithmetic solvers that decide satisfiability of linear constraint problems have many practical uses (e.g., modeling finite sets, program arithmetic, manipulation of pointers and memory, real-time constraints, physical properties of the environment) and are very important modules of many automated reasoning tools (e.g., theorem provers, SMT solvers). Throughout history, many different algorithms have been developed and, due to their importance, many have been formally verified with machine checkable proofs [7,14,15,16].

The quantifier-free fragment of linear arithmetic is very important for many applications (especially in SMT solving [4]). Most efficient decision procedures for this fragment are based on an *incremental versions of Simplex algorithm* [10,9]. They are specially adapted for use within SMT solvers and used in many industrial applications. The basic procedure is formulated for linear rational arithmetic, but its extensions use branch-and-bound and Gomory cuts techniques [10] and can also handle integer constraints. We are not aware that any Simplex based algorithm has been formally verified within a proof assistant (and literature [9] shows only a sketch of its termination, but no partial correctness).

We present an Isabelle/HOL [18] formalization and total correctness proof of the Simplex-based satisfiability checking procedure for linear rational arithmetic given by Dutertre and de Moura [9,10]. Only the central case of deciding conjunctions of constraints is considered — handling richer propositional structure of the formula is left for future integrations with verified SAT solvers [13].

Our formalization is highly modular and based on the *stepwise program and data refinement* — a well-studied technique due to Dijkstra [8] and Wirth [19], and given mathematical rigor by Back [2]. Our formalization exploits several different refinement techniques described for Isabelle/HOL [11,17]. Simplex algorithm exhibits several symmetric cases that are handled with special care,

significantly simplifying the proofs. The importance of treating symmetries carefully has already been suggested in the literature [12,16].

Although unverified SMT solving procedures can be successfully used within theorem provers using the *certificate checking* technique [1,5], we advocate that the formal verification here presented has its own merits.

- The formalization offers clear explanations for subtle procedure details.
- By strictly applying the refinement techniques, the procedure can be analyzed and understood on different levels of abstraction.
- Abstract layers in the formalization allow easy porting of our formalization to other theorem provers and verification systems.
- Executable code can be generated from the formalization and, by means of *reflection* [7,15], the procedure can be used to decide validity of universally quantified linear arithmetic constraints.
- The refinement approach makes this formalization suitable for a case study for teaching formal methods.
- The formalization is a contribution to the growing body of verified theorem proving algorithms.

The paper contains precise specifications (preconditions and effects) for all functions and aims to describe how these are obtained from decisions made during the algorithm development. All proofs are omitted from the presentation. Stepwise refinement is pursued down to several simple functions, that can be easily implemented. Their implementation is omitted from the present text, but is given in the Isabelle/HOL formalization[1] (containing a fully executable code).

Outline. The rest of the paper is structured as follows. In Section 2 we give a brief overview of linear arithmetic and Simplex algorithm, Isabelle/HOL and techniques for program and data refinement in Isabelle/HOL. In Section 3 we present our formalization of the Simplex-based LRA solver and show all refinement steps. In Section 4 we discuss the related work and give some experimental comaparisons. In Section 5 we draw some conclusions and discuss further work.

2 Background

Linear arithmetic. Linear arithmetic is a decidable fragment of arithmetic involving addition and multiplication by constants. Constraints are usually formulated either over reals or rationals (linear rational arithmetic, or *LRA*) or over integers (linear integer arithmetic, or *LIA*). A quantifier-free linear arithmetic formula is a first-order formula with atoms of the form: $a_1x_1 + \ldots + a_nx_n \bowtie c$, where a_i and c are rational numbers, x_i are (rational or integer) variables, and $\bowtie$ is one of the operators $=$, $\leq$, $<$, $>$, $\geq$, or $\neq$. Most popular methods for deciding satisfiability of LA formulae are the Fourier-Motzkin procedure and the Simplex algorithm.

Simplex algorithm. Simplex algorithm (invented by George Dantzig in 1947.) is listed among the top 10 algorithms of the 20th century, and it is originally

[1] Available online http://argo.matf.bg.ac.rs

constructed to solve linear programming optimization problem (to maximize objective function on a convex polytope, specified by the set of linear constraints). The decision procedure for linear arithmetic does not have to maximize anything, but have to find a single feasible solution of input constraints. The variant of Simplex method that can be used for this purpose is the *dual Simplex algorithm*, that is quite effective when constraints are added incrementally.

Isabelle/HOL. Isabelle/HOL[18] is a proof assistant for *Higher-order logic (HOL)*. HOL conforms largely to everyday mathematical notation. Terms are built using function applications (e.g., $f\ x$) and λ-abstractions (e.g., $\lambda x.x$). $let\ x\ =\ t\ in\ P\ x$ reduces to $P\ t$. *if-then-else* and *case* expressions are also supported. Basic types we use are Booleans (*bool*), naturals (*nat*), and rationals (*rat*). Type variables are denoted by $'a$, $'b$, ... Sets over type $'a$ (denoted by $'a\ set$) follow usual conventions. Lists over type $'a$ (denoted by $'a\ list$) come with the empty list [], the infix constructor #, and standard higher-order functionals *map* and *foldl*. Missing values are modeled by options over type $'a$ (denoted by $'a\ option$) that are either *None* or *Some* $'a$, and *the* is the function such that $the(Some\ x) = x$. Finite mappings from type $'a$ to type $'b$ (denoted by $('a,'b)mapping$) come with a lookup (here denoted by *look*), and update (here denoted by *upd*) operators. Algebraic datatypes (using the keyword **datatype**) and compound types (using the keyword **record**) are supported. For each record field, there is a selector function of the same name (e.g., accessing the field x in record r is denoted by $x\,r$). Equality is polymorphic and is denoted by either $=$, $\equiv$ or $\longleftrightarrow$ (on type *bool*). Functions are defined by both primitive and general recursion. From the specifications, executable code in several functional languages can be generated.

Refinement in Isabelle/HOL. Since Isabelle/HOL is a general proof-assistant, there are many ways to express refinement. Several frameworks for refinement (e.g., by Proteasa and Back or by Lammich) are available at Archive of Formal Proofs (http://afp.sf.net). However, our formalization uses only the following two (rather leightweight) approaches.

One approach for data and program refinement, based on code-generation facilities of Isabelle/HOL, is described by Haftmann and Nipkow [11]. To replace one function implementation by another, a proof of their equivalence must be made and the code generator must be instructed to use the desired implementation. Note that no axiomatic specification is used in this case. For data refinement, the first step requires defining an abstract data type representation and functions operating on this representation. Further steps require defining concrete data type representation, defining the conversion from the concrete to the abstract representation, and defining functions that operate on the concrete type. Formalizations should rely only on the abstract representation, while concrete representations are used only during code generation. For more details see [11].

Another approach for program refinement is based on *locales* [3] — Isabelle's version of parametrized theories. A locale is a named context of functions f_1, ..., f_n and assumptions P_1, ..., P_m about them that is introduced roughly like **locale** loc = **fixes** f_1, ..., f_n **assumes** P_1, ..., P_m. Locales can be hierarchical

as in **locale** $loc = loc_1 + loc_2 +$ **fixes** .... In the context of a locale, definitions can be made and theorems can be proved. Locales can be interpreted by concrete instances of $f_1, \ldots, f_n$, and then it must be shown that these satisfy assumptions $P_1, \ldots, P_m$. A locale *loc* is a *sublocale* of a locale loc' if all functions of loc' can be defined using the functions of *loc* and all assumptions of loc' can be proved using the assumptions of *loc*. Then every interpretation for *loc* can be automatically converted to an interpretation of loc'.

In the context of program refinement, locales are used to define specifications, i.e., abstract interfaces of functions (e.g., **locale** $F =$ **fixes** f **assumes** P). A refinement step can consist of changing the interface by adding stronger premises (e.g., **locale** $F' =$ **fixes** f **assumes** P'). Then a sublocale relation between F and F' must be proved. A slightly more complicated case is when the function f can be implemented using several functions g_i, each specified in its own locale (e.g., **locale** $G_i =$ **fixes** g_i **assumes** Q_i). Then, a joint locale can be defined (e.g., **locale** $F' = G_1 + \ldots + G_k$) and f can be defined in it. To prove the refinement, the sublocale relation between F and F' must be proved. A similar technique is described by Nipkow [17].

3 Formalization

Next, we present our formalization of the incremental Simplex procedure.

3.1 Linear Polynomials and Constraints

Linear polynomials are of the form $a_1 \cdot x_1 + ... + a_n \cdot x_n$. Their formalization follows the data-refinement approach of Isabelle/HOL [11]. Abstract representation of polynomials are functions mapping variables to their coefficients, where only finitely many variables have non-zero coefficients. Operations on polynomials are defined as operations on functions. For example, the sum of p_1 and p_2 is defined by $\lambda\ v.\ p_1\ v + p_2\ v$ and the value of a polynomial p for a valuation v (denoted by $p\{\!|v|\!\}$), is defined by $\sum x \in \{x.\ p\ x \neq 0\}.\ p\ x \cdot v\ x$. Executable representation of polynomials uses RBT mappings instead of functions.

Linear constraints are of the form $p \bowtie c$ or $p_1 \bowtie p_2$, where p, p_1, and p_2, are linear polynomials, c is a rational constant and $\bowtie \in \{<, >, \leq, \geq, =\}$. Their abstract syntax is given by the *constraint* type, and semantics is given by the relation $\models_c$, defined straightforwardly by primitive recursion over the *constraint* type. The list of contraints is satisfied, denoted by $\models_{cs}$, if all constraints are.

datatype *constraint* = *LT linear-poly rat* | *GT linear-poly rat* | ...
$v \models_c LT\ l\ r \longleftrightarrow l\{\!|v|\!\} < r$ | $v \models_c GT\ l\ r \longleftrightarrow l\{\!|v|\!\} > r$ | ...
$v \models_{cs} cs \equiv \forall\ c \in set\ cs.\ v \models_c c$

3.2 Procedure Specification

The specification for the satisfiability check procedure is given by:

locale *Solve* =
— Decide if the given list of constraints is satisfiable. Return the satisfiability status and, in the satisfiable case, one satisfying valuation.
fixes *solve* :: *constraint list* ⇒ *bool* × *rat valuation option*
— If the status *True* is returned, then returned valuation satisfies all constraints.
assumes *let* (*sat*, *v*) = *solve cs in sat* ⟶ *the v* ⊨$_{cs}$ *cs*
— If the status *False* is returned, then constraints are unsatisfiable.
assumes *let* (*sat*, *v*) = *solve cs in* ¬ *sat* ⟶ ¬ (∃ *v*. *v* ⊨$_{cs}$ *cs*)

Note that the above specification requires returning a valuation (defined as a HOL function), which is not efficiently executable. In order to enable more efficient data structures for representing valuations, a refinement of this specification is needed and the function *solve* is replaced by the function *solve-exec* returning optional (*var*, *rat*) *mapping* instead of *var* ⇒ *rat* function. This way, efficient data structures for representing mappings can be easily plugged-in during code generation [11]. A conversion from the *mapping* datatype to HOL function is denoted by ⟨-⟩ and given by: ⟨*v*⟩ *x* ≡ *case look v x of Some y* ⇒ *y*.

3.3 Handling Strict Inequalities

The first step of the procedure is removing all equalities and strict inequalities. Equalities can be easily rewritten to non-strict inequalities. Removing strict inequalities can be done by replacing the list of constraints by a new one, formulated over an extension $\mathbb{Q}'$ of the space of rationals $\mathbb{Q}$. $\mathbb{Q}'$ must have a structure of a linearly ordered vector space over $\mathbb{Q}$ (represented by the type class *lrv*) and must guarantee that if some non-strict constraints are satisfied in $\mathbb{Q}'$, then there is a satisfying valuation for the original constraints in $\mathbb{Q}$. Our final implementation uses the $\mathbb{Q}_\delta$ space, defined in [10] (basic idea is to replace $p < c$ by $p \leq c - \delta$ and $p > c$ by $p \geq c + \delta$ for a symbolic parameter δ). So, all constraints are reduced to the form $p \bowtie b$, where p is a linear polynomial (still over $\mathbb{Q}$), b is constant from $\mathbb{Q}'$ and $\bowtie \in \{\leq, \geq\}$. The non-strict constraints are represented by the type *'a ns-constraint*, and their semantics is denoted by ⊨$_{ns}$ and ⊨$_{nss}$.

datatype *'a ns-constraint* = *LEQ*$_{ns}$ *linear-poly 'a* | *GEQ*$_{ns}$ *linear-poly 'a*
v ⊨$_{ns}$ *LEQ*$_{ns}$ *l r* ⟷ *l*⟦*v*⟧ ≤ *r* | *v* ⊨$_{ns}$ *GEQ*$_{ns}$ *l r* ⟷ *l*⟦*v*⟧ ≥ *r*
v ⊨$_{nss}$ *cs* ≡ ∀ *c* ∈ *set cs*. *v* ⊨$_{ns}$ *c*

Specification of reduction of constraints to non-strict form is given by:

locale *To-ns* =
— Convert a constraint list to an equisatisfiable non-strict constraint list.
fixes *to-ns* :: *constraint list* ⇒ *'a*::*lrv ns-constraint list*
assumes *v* ⊨$_{cs}$ *cs* ⟹ ∃ *v'*. *v'* ⊨$_{nss}$ *to-ns cs*
— Convert the valuation that satisfies all non-strict constraints to the valuation that satisfies all initial constraints.
fixes *from-ns* :: (*var*, *'a*) *mapping* ⇒ *'a ns-constraint list* ⇒ (*var*, *rat*) *mapping*
assumes ⟨*v'*⟩ ⊨$_{nss}$ *to-ns cs* ⟹ ⟨*from-ns v'* (*to-ns cs*)⟩ ⊨$_{cs}$ *cs*

After the transformation, the procedure is reduced to solving only the non-strict constraints, implemented in the *solve-exec-ns* function having an analogous

specification to the *solve* function. If *to-ns*, *from-ns* and *solve-exec-ns* are available, the *solve-exec* function can be easily defined and it can be easily shown that this definition satisfies its specification (also analogous to *solve*).

solve-exec cs ≡ *let cs′* = *to-ns cs*; (*sat*, *v*) = *solve-exec-ns cs′ in*
if sat then (*True*, *Some* (*from-ns* (*the v*) *cs′*)) *else* (*False*, *None*)

3.4 Preprocessing

The next step in the procedure rewrites a list of non-strict constraints into an equisatisfiable form consisting of a list of linear equations (called the *tableau*) and of a list of *atoms* of the form $x_i \bowtie b_i$ where x_i is a variable and b_i is a constant (from the extension field). The transformation is straightforward and introduces auxiliary variables for linear polynomials occurring in the initial formula. For example, $[x_1 + x_2 \leq b_1,\ x_1 + x_2 \geq b_2,\ x_2 \geq b_3]$ can be transformed to the tableau $[x_3 = x_1 + x_2]$ and atoms $[x_3 \leq b_1,\ x_3 \geq b_2,\ x_2 \geq b_3]$.

Equations are of the form $x = p$, where x is a variable and p is a polynomial, and are represented by the type *eq* = *var* × *linear-poly*. Semantics of equations is given by $v \models_e (x,\ p) \equiv v\ x = p\ \{\!|\ v\ |\!\}$. Tableau is represented as a list of equations, by the type *tableau* = *eq list*. Semantics for a tableau is given by $v \models_t t \equiv \forall\, e{\in}set\ t.\ v \models_e e$. Functions *lvars* and *rvars* return sets of variables appearing on the left hand side (lhs) and the right hand side (rhs) of a tableau. Lhs variables are called *basic* while rhs variables are called *non-basic* variables. A tableau t is *normalized* (denoted by $\triangle\ t$) iff no variable occurs on the lhs of two equations in a tableau and if sets of lhs and rhs variables are distinct.

Elementary atoms are represented by the type *′a atom* and semantics for atoms and sets of atoms is denoted by $\models_a$ and $\models_{as}$ and given by:

datatype *′a atom* = *Leq var ′a* | *Geq var ′a*
$v \models_a$ *Leq x c* ⟷ $v\ x \leq c$ | $v \models_a$ *Geq x c* ⟷ $v\ x \geq c$
$v \models_{as}$ *as* ≡ ∀ *a* ∈ *as*. $v \models_a a$

The specification of the preprocessing function is given by:

locale *Preprocess* = **fixes** *preprocess*::*′a*::*lrv ns-constraint list* ⇒ *tableau*×*′a atom list*
assumes
— The returned tableau is always normalized.
let (*t*, *as*) = *preprocess cs in* △ *t*
— Tableau and atoms are equisatisfiable with starting non-strict constraints.
let (*t*, *as*) = *preprocess cs in* $v \models_{as}$ *set as* ∧ $v \models_t t \longrightarrow v \models_{nss}$ *cs*
let (*t*, *as*) = *preprocess cs in* $v \models_{nss}$ *cs* ⟶ (∃ *v′*. $v' \models_{as}$ *set as* ∧ $v' \models_t t$)

Once the preprocessing is done and tableau and atoms are obtained, their satisfiability is checked by the *assert-all* function. Its precondition is that the starting tableau is normalized, and its specification is analogue to the one for the *solve* function. If *preprocess* and *assert-all* are available, the *solve-exec-ns* can be defined, and it can easily be shown that this definition satisfies the specification.

solve-exec-ns s ≡ *let* (*t*, *as*) = *preprocess s in assert-all t as*

3.5 Incrementally Asserting Atoms

The function *assert-all* can be implemented by iteratively asserting one by one atom from the given list of atoms.

Asserted atoms will be stored in a form of *bounds* for a given variable. Bounds are of the form $l_i \leq x_i \leq u_i$, where l_i and u_i and are either scalars or $\pm\infty$. Each time a new atom is asserted, a bound for the corresponding variable is updated (checking for conflict with the previous bounds). Since bounds for a variable can be either finite or $\pm\infty$, they are represented by (partial) maps from variables to values ($'a$ *bounds* = *var* $\rightharpoonup$ $'a$). Upper and lower bounds are represented separately. Infinite bounds map to *None* and this is reflected in the semantics:

$c \geq_{ub} b \longleftrightarrow$ *case* b *of None* $\Rightarrow$ *False* | *Some* $b' \Rightarrow c \geq b'$

$c \leq_{ub} b \longleftrightarrow$ *case* b *of None* $\Rightarrow$ *True* | *Some* $b' \Rightarrow c \leq b'$

Strict comparisons, and comparisons with lower bounds are performed similiarly.

A valuation satisfies bounds iff the value of each variable respects both its lower and upper bound, i.e, $v \models_b (lb,\ ub) \equiv \forall x.\ v\ x \geq_{lb} lb\ x \wedge v\ x \leq_{ub} ub\ x$. Asserted atoms are precisely encoded by the current bounds in a state (denoted by $\doteq$) if every valuation satisfies them iff it satisfies the bounds, i.e., $as \doteq (lb,\ ub) \equiv \forall v.\ v \models_{as} as \longleftrightarrow v \models_b (lb,\ ub)$.

The procedure also keeps track of a valuation that is a candidate solution. Whenever a new atom is asserted, it is checked whether the valuation is still satisfying. If not, the procedure tries to fix that by changing it and changing the tableau if necessary (but so that it remains equivalent to the initial tableau).

Therefore, the state of the procedure stores the tableau (denoted by $\mathcal{T}$), lower and upper bounds (denoted by $\mathcal{B}_l$ and $\mathcal{B}_u$, and ordered pair of lower and upper bounds denoted by $\mathcal{B}$), candidate solution (denoted by $\mathcal{V}$) and a flag (denoted by $\mathcal{U}$) indicating if unsatisfiability has been detected so far:

record $'a$ *state* =
$\mathcal{T}$ :: *tableau* $\quad \mathcal{B}_l$ :: $'a$ *bounds* $\quad \mathcal{B}_u$ :: $'a$ *bounds* $\quad \mathcal{V}$ :: (*var*, $'a$) *mapping* $\quad \mathcal{U}$ :: *bool*

To be a solution of the initial problem, a valuation should satisfy the initial tableau and list of atoms. Since tableau is changed only by equivalency preserving transformations and asserted atoms are encoded in the bounds, a valuation is a solution if it satisfies both the tableau and the bounds in the final state (when all atoms have been asserted). So, a valuation v satisfies a state s (denoted by $\models_s$) if it satisfies the tableau and the bounds, i.e., $v \models_s s \equiv v \models_b \mathcal{B}\ s \wedge v \models_t \mathcal{T}\ s$. Since $\mathcal{V}$ should be a candidate solution, it should satisfy the state (unless the $\mathcal{U}$ flag is raised). This is denoted by $\models s$ and defined by $\models s \equiv \langle \mathcal{V}\ s \rangle \models_s s$. $\bigtriangledown\ s$ will denote that all variables of $\mathcal{T}\ s$ are explicitly valuated in $\mathcal{V}\ s$.

Assuming that the $\mathcal{U}$ flag and the current valuation $\mathcal{V}$ in the final state determine the solution of a problem, the *assert-all* function can be reduced to the *assert-all-state* function that operates on the states:

assert-all t *as* $\equiv$ *let* s = *assert-all-state* t *as in*
if $(\mathcal{U}\ s)$ *then* (*False*, *None*) *else* (*True*, *Some* $(\mathcal{V}\ s)$)

Specification for the *assert-all-state* can be directly obtained from the specification of *assert-all*, and it describes the connection between the valuation in the final state and the initial tableau and atoms. However, we will make an additional refinement step and give stronger assumptions about the *assert-all-state* function that describes the connection between the initial tableau and atoms with the tableau and bounds in the final state.

locale *AssertAllState* = **fixes** *assert-all-state*::*tableau* ⇒ *'a*::*lrv atom list* ⇒ *'a state*
assumes
— The final and the initial tableau are equivalent.
△ $t \Longrightarrow$ *let* s' = *assert-all-state* t *as in* (v::*'a valuation*) $\models_t$ $t \longleftrightarrow v \models_t \mathcal{T}\ s'$
— If $\mathcal{U}$ is not raised, then the valuation in the final state satisfies its tableau and its bounds (that are, in this case, equivalent to the set of all asserted bounds).
△ $t \Longrightarrow$ *let* s' = *assert-all-state* t *as in* $\neg\ \mathcal{U}\ s' \longrightarrow \models s'$
△ $t \Longrightarrow$ *let* s' = *assert-all-state* t *as in* $\neg\ \mathcal{U}\ s' \longrightarrow$ *set as* $\doteq \mathcal{B}\ s'$
— If $\mathcal{U}$ is raised, then there is no valuation satisfying the tableau and the bounds in the final state (that are, in this case, equivalent to a subset of asserted atoms).
△ $t \Longrightarrow$ *let* s' = *assert-all-state* t *as in* $\mathcal{U}\ s' \longrightarrow \neg\ (\exists\ v.\ v \models_s s')$
△ $t \Longrightarrow$ *let* s' = *assert-all-state* t *as in* $\mathcal{U}\ s' \longrightarrow (\exists\ as'.\ as' \subseteq$ *set as* $\wedge\ as' \doteq \mathcal{B}\ s')$

The *assert-all-state* function can be implemented by first applying the *init* function that creates an initial state based on the starting tableau, and then by iteratively applying the *assert* function for each atom in the starting atoms list.

assert-loop as s ≡ *foldl* ($\lambda\ s'\ a.$ *if* ($\mathcal{U}\ s'$) *then* s' *else assert a* s') s *as*
assert-all-state t as ≡ *assert-loop ats* (*init t*)

Specification for *init* can be obtained from the specification of *asser-all-state* since all its assumptions must also hold for *init* (when the list of atoms is empty). Also, since *init* is the first step in the *assert-all-state* implementation, the precondition for *init* the same as for the *assert-all-state*. However, unsatisfiability is never going to be detected during initialization and $\mathcal{U}$ flag is never going to be raised. Also, the tableau in the initial state can just be initialized with the starting tableau. The condition $\{\} \doteq \mathcal{B}$ (*init t*) is equivalent to asking that initial bounds are empty. Therefore, specification for *init* can be refined to:

locale *Init* = **fixes** *init*::*tableau* ⇒ *'a*::*lrv state*
assumes
— Tableau in the initial state for t is t: $\mathcal{T}$ (*init t*) = t
— Since unsatisfiability is not detected, $\mathcal{U}$ flag must not be set: $\neg\ \mathcal{U}$ (*init t*)
— The current valuation must satisfy the tableau: $\langle \mathcal{V}$ (*init t*)$\rangle \models_t t$
— In an inital state no atoms are yet asserted so the bounds must be empty:
$\mathcal{B}_l$ (*init t*) = (λ -. *None*) $\mathcal{B}_u$ (*init t*) = (λ -. *None*)
— All tableau vars are valuated: $\bigtriangledown$ (*init t*)

The *assert* function asserts a single atom. Since the *init* function does not raise the $\mathcal{U}$ flag, from the definition of *assert-loop*, it is clear that the flag is not raised when the *assert* function is called. Moreover, the assumptions about the *assert-all-state* imply that the loop invariant must be that if the $\mathcal{U}$ flag is not raised, then the current valuation must satisfy the state (i.e., $\models s$). The

assert function will be more easily implemented if it is always applied to a state with a normalized and valuated tableau, so we make this another loop invariant. Therefore, the precondition for the *assert a s* function call is that $\neg\ \mathcal{U}\ s$, $\models s$, $\triangle\ (\mathcal{T}\ s)$ and $\bigtriangledown\ s$ hold. The specification for *assert* directly follows from the specification of *assert-all-state* (except that it is additionally required that bounds reflect asserted atoms also when unsatisfiability is detected, and that it is required that *assert* keeps the tableau normalized and valuated).

locale *Assert* = **fixes** *assert*::$'a$::*lrv atom* ⇒ $'a$ *state* ⇒ $'a$ *state*
assumes
— Tableau remains equivalent to the previous one and normalized and valuated.
$[\![\neg\ \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \bigtriangledown\ s]\!] \Longrightarrow$ *let* $s' =$ *assert a s in*
$((v::'a\ \textit{valuation}) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ s') \wedge \triangle\ (\mathcal{T}\ s') \wedge \bigtriangledown\ s'$
— If the $\mathcal{U}$ flag is not raised, then the current valuation is updated so that it satisfies the current tableau and the current bounds.
$[\![\neg\ \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \bigtriangledown\ s]\!] \Longrightarrow \neg\ \mathcal{U}\ (\textit{assert}\ a\ s) \longrightarrow \models (\textit{assert}\ a\ s)$
— The set of asserted atoms remains equivalent to the bounds in the state.
$[\![\neg\ \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \bigtriangledown\ s]\!] \Longrightarrow \textit{ats} \doteq \mathcal{B}\ s \longrightarrow (\textit{ats} \cup \{a\}) \doteq \mathcal{B}\ (\textit{assert}\ a\ s)$
— If the $\mathcal{U}$ flag is raised, then there is no valuation that satisfies both the current tableau and the current bounds.
$[\![\neg\ \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \bigtriangledown\ s]\!] \Longrightarrow \mathcal{U}\ (\textit{assert}\ a\ s) \longrightarrow\ \neg\ (\exists\ v.\ v \models_s (\textit{assert}\ a\ s))$

Under these assumptions, it can easily be shown (mainly by induction) that the previously shown implementation of *assert-all-state* satisfies its specification.

3.6 Aserting Single Atoms

The *assert* function is split in two phases. First, *assert-bound* updates the bounds and checks only for conflicts cheap to detect. Next, *check* performs the full simplex algorithm. The *assert* function can be implemented as *assert a s* = *check* (*assert-bound a s*). Note that it is also possible to do the first phase for several asserted atoms, and only then to let the expensive second phase work.

Asserting an atom $x \bowtie b$ begins with the function *assert-bound*. If the atom is subsumed by the current bounds, then no changes are performed. Otherwise, bounds for x are changed to incorporate the atom. If the atom is inconsistent with the previous bounds for x, the $\mathcal{U}$ flag is raised. If x is not a lhs variable in the current tableau and if the value for x in the current valuation violates the new bound b, the value for x can be updated and set to b, meanwhile updating the values for lhs variables of the tableau so that it remains satisfied. Otherwise, no changes to the current valuation are performed.

So, the *assert-bound* function must ensure that the given atom is included in the bounds, that the tableau remains satisfied by the valuation and that all variables except the lhs variables in the tableau are within their bounds. To formalize this, we introduce the notation $v \models_b (lb, ub) \parallel S$, and define $v \models_b (lb, ub) \parallel S \equiv \forall x{\in}S.\ v\ x \geq_{lb} lb\ x \wedge v\ x \leq_{ub} ub\ x$, and $\models_{nolhs} s \equiv \langle \mathcal{V}\ s\rangle \models_t \mathcal{T}\ s \wedge \langle \mathcal{V}\ s\rangle \models_b \mathcal{B}\ s \parallel - \textit{lvars}\ (\mathcal{T}\ s)$. The *assert-bound* function raises the $\mathcal{U}$ flag if

and only if lower and upper bounds overlap. This is formalized as $\Diamond\ s \equiv \forall\, x.$ *if* $\mathcal{B}_l\ s\ x = None \vee \mathcal{B}_u\ s\ x = None$ *then True else the* $(\mathcal{B}_l\ s\ x) \leq$ *the* $(\mathcal{B}_u\ s\ x)$.

Since the *assert-bound* is the first step in the *assert* function implementation, the preconditions for *assert-bound* are the same as preconditions for the *assert* function. The specifiction for the *assert-bound* is:

locale *AssertBound* = **fixes** *assert-bound*::$'a$::*lrv atom* $\Rightarrow$ $'a$ *state* $\Rightarrow$ $'a$ *state*
assumes
— The tableau remains unchanged and valuated.
$[\![\neg\, \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow$ *let* $s' =$ *assert-bound a s in* $\mathcal{T}\ s' = \mathcal{T}\ s \wedge \triangledown\ s'$
— If the $\mathcal{U}$ flag is not set, all but the lhs variables in the tableau remain within their bounds, the new valuation satisfies the tableau, and bounds do not overlap. $[\![\neg\, \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow$
let $s' =$ *assert-bound a s in* $\neg\, \mathcal{U}\ s' \longrightarrow \models_{nolhs} s' \wedge \Diamond\ s'$
— The set of asserted atoms remains equivalent to the bounds in the state.
$[\![\neg\, \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow$ *ats* $\doteq \mathcal{B}\ s \longrightarrow$ (*ats* $\cup \{a\}) \doteq \mathcal{B}$ (*assert-bound a s*)
— $\mathcal{U}$ flag is raised, only if the bounds became inconsistent:
$[\![\neg\, \mathcal{U}\ s; \models s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow$ *let* $s' =$ *assert-bound a s in* $\mathcal{U}\ s' \longrightarrow \neg(\exists\, v.\ v \models_s s')$

The second phase of *assert*, the *check* function, is the heart of the Simplex algorithm. It is always called after *assert-bound*, but in two different situations. In the first case *assert-bound* raised the $\mathcal{U}$ flag and then *check* should retain the flag and should not perform any changes. In the second case *assert-bound* did not raise the $\mathcal{U}$ flag, so $\models_{nolhs} s$, $\Diamond\ s$, $\triangle\ (\mathcal{T}\ s)$, and $\triangledown\ s$ hold.

locale *Check* = **fixes** *check*::$'a$::*lrv state* $\Rightarrow$ $'a$ *state*
assumes
— If *check* is called from an inconsistent state, the state is unchanged.
$[\![\ \mathcal{U}\ s\ ]\!] \Longrightarrow$ *check* $s = s$
— The tableau remains equivalent to the previous one, normalized and valuated.
$[\![\neg\, \mathcal{U}\ s; \models_{nolhs} s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow$
let $s' =$ *check s in* $((v::'a$ *valuation*$) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ s') \wedge \triangle\ (\mathcal{T}\ s') \wedge \triangledown\ s'$
— The bounds remain unchanged.
$[\![\neg\, \mathcal{U}\ s; \models_{nolhs} s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow \mathcal{B}$ (*check s*) $= \mathcal{B}\ s$
— If $\mathcal{U}$ flag is not raised, the current valuation $\mathcal{V}$ satisfies both the tableau and the bounds and if it is raised, there is no valuation that satisfies them.
$[\![\neg\, \mathcal{U}\ s; \models_{nolhs} s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow \neg\, \mathcal{U}$ (*check s*) $\longrightarrow \models$ (*check s*)
$[\![\neg\, \mathcal{U}\ s; \models_{nolhs} s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \triangledown\ s]\!] \Longrightarrow \mathcal{U}$ (*check s*) $\longrightarrow \neg\ (\exists\ v.\ v \models_s$ (*check s*))

Under these assumptions for *assert-bound* and *check*, it can be easily shown that the implementation of *assert* (previously given) satisfies its specification.

However, for efficiency reasons, we want to allow implementations that delay the *check* function call and call it after several *assert-bound* calls. For example:

assert-bound-loop ats s $\equiv$ *foldl* $(\lambda s'\ a.$ *if* $\mathcal{U}\ s'$ *then* s' *else assert-bound a* $s')$ *s ats*
assert-all-state t ats $\equiv$ *check* (*assert-bound-loop ats* (*init t*))

Then, the loop consists only of *assert-bound* calls, so *assert-bound* postcondition must imply its precondition. This is not the case, since variables on the lhs may be out of their bounds. Therefore, we make a refinement and specify weaker preconditions (replace $\models s$, by $\models_{nolhs} s$ and $\Diamond\ s$) for *assert-bound*, and

show that these preconditions are still good enough to prove the correctnes of this alternative *assert-all-state* definition.

3.7 Update and Pivot

Both *assert-bound* and *check* need to update the valuation so that the tableau remains satisfied. If the value for a variable not on the lhs of the tableau is changed, this can be done rather easily (once the value of that variable is changed, one should recalculate and change the values for all lhs variables of the tableau). The *update* function does this, and it is specified by:

locale *Update* = **fixes** *update*::*var* ⇒ 'a::*lrv* ⇒ 'a *state* ⇒ 'a *state*
assumes
— Tableau, bounds, and the unsatisfiability flag are preserved.
$[\![\triangle\ (\mathcal{T}\ s);\ \triangledown\ s;\ x \notin lvars\ (\mathcal{T}\ s)]\!] \Longrightarrow$
let $s' = update\ x\ c\ s$ *in* $\mathcal{T}\ s' = \mathcal{T}\ s \wedge \mathcal{B}\ s' = \mathcal{B}\ s \wedge \mathcal{U}\ s' = \mathcal{U}\ s$
— Tableau remains valuated.
$[\![\triangle\ (\mathcal{T}\ s);\ \triangledown\ s;\ x \notin lvars\ (\mathcal{T}\ s)]\!] \Longrightarrow \triangledown\ (update\ x\ v\ s)$
— The given variable x in the updated valuation is set to the given value v while all other variables (except those on the lhs of the tableau) are unchanged.
$[\![\triangle\ (\mathcal{T}\ s);\ \triangledown\ s;\ x \notin lvars\ (\mathcal{T}\ s)]\!] \Longrightarrow x' \notin lvars\ (\mathcal{T}\ s) \longrightarrow$
look $(\mathcal{V}\ (update\ x\ v\ s))\ x' = (if\ x = x'\ then\ Some\ v\ else\ look\ (\mathcal{V}\ s)\ x')$
— Updated valuation continues to satisfy the tableau.
$[\![\triangle\ (\mathcal{T}\ s);\ \triangledown\ s;\ x \notin lvars\ (\mathcal{T}\ s)]\!] \Longrightarrow\ \langle\mathcal{V}\ s\rangle \models_t \mathcal{T}\ s \longrightarrow \langle\mathcal{V}\ (update\ x\ c\ s)\rangle \models_t \mathcal{T}\ s$

Given the *update* function, *assert-bound* can be implemented as follows.

assert-bound $(Leq\ x\ c)\ s \equiv$
  if $c \geq_{ub} \mathcal{B}_u\ s\ x$ *then* s
  else let $s' = s\ (\!|\ \mathcal{B}_u := (\mathcal{B}_u\ s)\ (x := Some\ c)\ |\!)$
    in if $c <_{lb} \mathcal{B}_l\ s\ x$ *then* $s'\ (\!|\ \mathcal{U} := True\ |\!)$
    else if $x \notin lvars\ (\mathcal{T}\ s') \wedge c < \langle\mathcal{V}\ s\rangle\ x$ *then update* $x\ c\ s'$ *else* s'

The case of $Geq\ x\ c$ atoms is analogous (a systematic way to avoid symmetries is discussed in Section 3.9). This implementation satisfies both its specifications.

Updating changes the value of x and then updates values of all lhs variables so that the tableau remains satisfied. This can be based on a function that recalculates rhs polynomial values in the changed valuation:

locale *RhsEqVal* = **fixes** *rhs-eq-val*::(*var*, 'a::*lrv*) *mapping* ⇒ *var* ⇒ 'a ⇒ *eq* ⇒ 'a
— *rhs-eq-val* computes the value of the rhs of e in $\langle v\rangle(x := c)$.
assumes $\langle v\rangle \models_e e \Longrightarrow$ *rhs-eq-val* $v\ x\ c\ e = rhs\ e\ \{\!|\ \langle v\rangle\ (x := c)\ |\!\}$

Then, the next implementation of *update* satisfies its specification:

update-eq $v\ x\ c\ v'\ e \equiv upd\ (lhs\ e)\ (rhs\text{-}eq\text{-}val\ v\ x\ c\ e)\ v'$
update $x\ c\ s \equiv s(\!|\mathcal{V} := upd\ x\ c\ (foldl\ (update\text{-}eq\ (\mathcal{V}\ s)\ x\ c)\ (\mathcal{V}\ s)\ (\mathcal{T}\ s))|\!)$

To update the valuation for a variable that is on the lhs of the tableau it should first be swapped with some rhs variable of its equation, in an operation called *pivoting*. Pivoting has the precondition that the tableau is normalized and that it is always called for a lhs variable of the tableau, and a rhs variable

in the equation with that lhs variable. The set of rhs variables for the given lhs variable is found using the *rvars-of-lvar* function (specified in a very simple locale *EqForLVar*, that we do not print).

locale *Pivot* = *EqForLVar* + **fixes** *pivot*::*var* ⇒ *var* ⇒ *'a*::*lrv state* ⇒ *'a state*
assumes
— Valuation, bounds, and the unsatisfiability flag are not changed.
⟦△ (𝒯 *s*); x_i ∈ *lvars* (𝒯 *s*); x_j ∈ *rvars-of-lvar* (𝒯 *s*) x_i⟧ ⟹
let *s'* = *pivot* x_i x_j *s* *in* 𝒱 *s'* = 𝒱 *s* ∧ ℬ *s'* = ℬ *s* ∧ 𝒰 *s'* = 𝒰 *s*
— The tableau remains equivalent to the previous one and normalized.
⟦△ (𝒯 *s*); x_i ∈ *lvars* (𝒯 *s*); x_j ∈ *rvars-of-lvar* (𝒯 *s*) x_i⟧ ⟹
let *s'* = *pivot* x_i x_j *s* *in* ((*v*::*'a valuation*) ⊨$_t$ 𝒯 *s* ⟷ *v* ⊨$_t$ 𝒯 *s'*) ∧ △ (𝒯 *s'*)
— x_i and x_j are swapped, while the other variables do not change sides.
⟦△ (𝒯 *s*); x_i ∈ *lvars* (𝒯 *s*); x_j ∈ *rvars-of-lvar* (𝒯 *s*) x_i⟧ ⟹ *let* *s'* = *pivot* x_i x_j *s* *in*
rvars(𝒯 *s'*) = *rvars*(𝒯 *s*)−{x_j}∪{x_i} ∧ *lvars*(𝒯 *s'*) = *lvars*(𝒯 *s*)−{x_i}∪{x_j}

Functions *pivot* and *update* can be used to implement the *check* function. In its context, *pivot* and *update* functions are always called together, so the following definition can be used: *pivot-and-update* x_i x_j *c* *s* = *update* x_i *c* (*pivot* x_i x_j *s*). It is possible to make a more efficient implementation of *pivot-and-update* that does not use separate implementations of *pivot* and *update*. To allow this, a separate specification for *pivot-and-update* can be given. It can be easily shown that the *pivot-and-update* definition above satisfies this specification.

Pivoting the tableau can be reduced to pivoting single equations, and substituting variable by polynomials. These operations are specified by:

locale *PivotEq* = **fixes** *pivot-eq*::*eq* ⇒ *var* ⇒ *eq*
assumes
— Lhs var of *eq* and x_j are swapped, while the other variables do not change sides.
⟦x_j ∈ *rvars-eq eq*; *lhs eq* ∉ *rvars-eq eq* ⟧ ⟹ *let* *eq'* = *pivot-eq eq* x_j *in*
lhs eq' = x_j ∧ *rvars-eq eq'* = {*lhs eq*} ∪ (*rvars-eq eq* − {x_j})
— Pivoting keeps the equation equisatisfiable.
⟦x_j ∈ *rvars-eq eq*; *lhs eq* ∉ *rvars-eq eq* ⟧ ⟹
(*v*::*'a*::*lrv valuation*) ⊨$_e$ *pivot-eq eq* x_j ⟷ *v* ⊨$_e$ *eq*
locale *SubstVar* = **fixes** *subst-var*::*var* ⇒ *linear-poly* ⇒ *linear-poly* ⇒ *linear-poly*
assumes
— Effect of *subst-var* x_j *lp'* *lp* on *lp* variables.
(*vars lp* − {x_j}) − *vars lp'* ⊆ *vars* (*subst-var* x_j *lp'* *lp*) ⊆ (*vars lp* − {x_j}) ∪ *vars lp'*
— Effect of *subst-var* x_j *lp'* *lp* on *lp* value.
(*v*::*'a*::*lrv valuation*) x_j = *lp'* ⦃*v*⦄ ⟶ *lp* ⦃*v*⦄ = (*subst-var* x_j *lp'* *lp*) ⦃*v*⦄

Then, the next implementation of *pivot* satisfies its specification:

pivot-tableau x_i x_j *t* ≡ *let* *eq* = *eq-for-lvar t* x_i; *eq'* = *pivot-eq eq* x_j *in*
map (λ *e*. *if lhs e* = *lhs eq then eq' else subst-var-eq* x_j (*rhs eq'*) *e*) *t*
pivot x_i x_j *s* ≡ *s*(| 𝒯 := *pivot-tableau* x_i x_j (𝒯 *s*) |)

3.8 Check implementation

The *check* function is called when all rhs variables are in bounds, and it checks if there is a lhs variable that is not. If there is no such variable, then satisfiability

is detected and *check* succeeds. If there is a lhs variable x_i out of its bounds, a rhs variable x_j is sought which allows pivoting with x_i and updating x_i to its violated bound. If x_i is under its lower bound it must be increased, and if x_j has a positive coefficient it must be increased so it must be under its upper bound and if it has a negative coefficient it must be decreased so it must be above its lower bound. The case when x_i is above its upper bound is symmetric (avoiding symmetries is discussed in Section 3.9). If there is no such x_j, unsatisfiability is detected and *check* fails. The procedure is recursively repeated, until it either succeeds or fails. To ensure termination, variables x_i and x_j must be chosen with respect to a fixed variable ordering. For choosing these variables auxiliary functions *min-lvar-not-in-bounds*, *min-rvar-inc* and *min-rvar-dec* are specified (each in its own locale). For, example:

locale *MinLVarNotInBounds* = **fixes** *min-lvar-not-in-bounds*::'*a*::*lrv state* ⇒ *var option*
assumes
min-lvar-not-in-bounds s = *None* ⟶ (∀ *x*∈*lvars* (*T s*). *in-bounds x* ⟨*V s*⟩ (*B s*))
min-lvar-not-in-bounds s = *Some* x_i ⟶ x_i∈*lvars* (*T s*) ∧ ¬*in-bounds* x_i ⟨*V s*⟩ (*B s*)
∧ (∀ *x*∈*lvars* (*T s*). *x* < x_i ⟶ *in-bounds x* ⟨*V s*⟩ (*B s*))

The definition of *check* can be given by:

check s ≡ *if U s then s*
else let x_i' = *min-lvar-not-in-bounds s in*
case x_i' *of None* ⇒ *s*
| *Some* x_i ⇒ *if* ⟨*V s*⟩ x_i <$_{lb}$ $\mathcal{B}_l$ *s* x_i *then check* (*check-inc* x_i *s*)
else check (*check-dec* x_i *s*)

check-inc x_i *s* ≡ *let* l_i = *the* ($\mathcal{B}_l$ *s* x_i); x_j' = *min-rvar-inc s* x_i *in*
case x_j' *of None* ⇒ *s* (| *U* := *True* |) | *Some* x_j ⇒ *pivot-and-update* x_i x_j l_i *s*

The definition of *check-dec* is analogous. It is shown (mainly by induction) that this definition satisfies the *check* specification. Note that this definition uses general recursion, so its termination is non-trivial. It has been shown that it terminates for all states satisfying the check preconditions. The proof is based on the proof outline given in [10]. It is very technically involved, but conceptually uninteresting so we do not discuss it in more details.

3.9 Symmetries

Simplex algorithm exhibits many symmetric cases. For example, *assert-bound* treats atoms *Leq x c* and *Geq x c* in a symmetric manner, *check-inc* and *check-dec* are symmetric, etc. These symmetric cases differ only in several aspects: order relations between numbers ($<$ vs $>$ and $\leq$ vs $\geq$), the role of lower and upper bounds ($\mathcal{B}_l$ vs $\mathcal{B}_u$) and their updating functions, comparisons with bounds (e.g., $\geq_{ub}$ vs $\leq_{lb}$ or $<_{lb}$ vs $>_{ub}$), and atom constructors (*Leq* and *Geq*). These can be attributed to two different orientations (positive and negative) of rational axis. To avoid duplicating definitions and proofs, *assert-bound* definition cases for *Leq* and *Geq* are replaced by a call to a newly introduced function parametrized by

a *Direction* — a record containing minimal set of aspects listed above that differ in two definition cases such that other aspects can be derived from them (e.g., only $<$ need to be stored while $\leq$ can be derived from it). Two constants of the type *Direction* are defined: *Positive* (with $<$, $\leq$ orders, $\mathcal{B}_l$ for lower and $\mathcal{B}_u$ for upper bounds and their corresponding updating functions, and *Leq* constructor) and *Negative* (completely opposite from the previous one). Similarly, *check-inc* and *check-dec* are replaced by a new function *check-incdec* parametrized by a *Direction*. All lemmas, previously repeated for each symmetric instance, were replaced by a more abstract one, again parametrized by a *Direction* parameter.

4 Related Work

The literature on decision procedures for linear arithmetic is vast. Regarding the formally verified algorithms, the closest work to ours is done by Chaieb and Nipkow [7,14,15,16]. They have verifed a number of quantifier-elemination algorithms for both rational and integer case. They cover arbitrary quantifiers and propositional structure (although by a simple DNF-based approach), but restrict atoms only to $<$ and $=$ relations. Our approach has more limited scope since it covers only the quantifier-free case for rational arithmetic, but our experimental results show that, due to the Simplex procedure, it significantly outperforms Fourier-Motzkin procedure verified by Nipkow [16]. We have tested 90 random generated quantifier-free LRA instances with 2-10 variables and 10-100 constraints. Fourier-Motzkin procedure solved only 8 within a 300s time-limit with average time of 66.40s, while Simplex solved all 90 with average time of 0.44s.

5 Conclusions and Further Work

We have presented a formalization of a functional model for the incremental Simplex procedure [10] used in most state-of-the art SMT solvers and proved its total correctness. Only the central case of deciding conjunctions of constraints was discussed, while other important but simpler questions (e.g., explanations, propagations) are left for further work.

The decision to use a stepwise refinement approach enormously simplified reasoning about the procedure. Initially, we did a formalization by formulating the whole algorithm and reasoning about it at once, and our experience shows that this monolith approach required proofs that are several times longer and much harder to understand and follow. Stepwise refinement makes the formalization modular and it is much easier to make changes to the procedure.

Another important decision in our formalization was to pay special attention to symmetric cases in the proof. Pen-and-paper termination proof outline [9] deals only with one of four symmetric cases arising in that context and concludes that other cases are handled „similarly". A direct approach would be to copy-paste the case four times and adapt the proof in each case. However, our generalizations made in basic predicate definitions, completly removed the need for case-analysis in the proof text.

The main obstacle for achieving the maximal efficiency is the lack of imperative data-structures in our formalization. This can be improved if the Imperative/HOL framework [6] is used. However, this does not fit well with our stepwise refinement approach. Imperative/HOL would require redefining the whole code using the monadic approach and proving some kind of equivalence with the current purely functional implementation.

References

1. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT solvers to Coq through Proof Witnesses. In *CPP 2011*, LNCS 7086. Springer, 2011.
2. Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Åbo Akademi, Helsinki, Finland, 1978. Report A–1978–4.
3. Clemens Ballarin. Interpretation of Locales in Isabelle: Theories and Proof Contexts. In *MKM 2006*, LNCS 4108, Springer, 2006.
4. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, IOS Press, 2009.
5. S. Böhme and T. Weber. Fast LCF-style Proof Reconstruction for Z3. In *ITP 2010*, LNCS 6172, 2010.
6. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, J. Matthews. Imperative Functional Programming with Isabelle/HOL. In *TPHOLs*, LNCS 5170, Springer, 2010.
7. A. Chaieb and T. Nipkow. Proof Synthesis and Reflection for Linear Arithmetic. *J. Automated Reasoning*, 41:33–59, 2008.
8. E. W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. *BIT Numerical Mathematics*, 8:174–186, 1968.
9. B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.
10. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV 2006*, LNCS 4144, Springer, 2006.
11. F. Haftmann and T. Nipkow. Code Generation via Higher-Order Rewrite Systems. In *FLOPS 2010*, LNCS 6009, Springer, 2010.
12. John Harrison. Without Loss of Generality. In *TPHOLs 2009*, LNCS 5674, Springer, 2009.
13. Filip Marić. Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.
14. Tobias Nipkow. Linear Quantifier Elimination. In *IJCAR 2008*, LNCS 5195, Springer, 2008.
15. Tobias Nipkow. Reflecting Quantifier Elimination for Linear Arithmetic. In *Formal Logical Methods for System Security and Correctness*, IOS Press, 2008.
16. Tobias Nipkow. Linear Quantifier Elimination. *J. Automated Reasoning*, 45:189–212, 2010.
17. Tobias Nipkow. Verified Efficient Enumeration of Plane Graphs Modulo Isomorphism. In *ITP 2011*, LNCS 6898, Springer, 2011.
18. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS 2283, Springer, 2002.
19. Niklaus Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 26(1):70–74, 1983.