Verifying an Incremental Theory Solver for Linear Arithmetic in Isabelle/HOL

Ralph Bottesch, Max W. Haslbeck, and René Thiemann(✉)

University of Innsbruck, Innsbruck, Austria
rene.thiemann@uibk.ac.at

Abstract. Dutertre and de Moura developed a simplex-based solver for linear rational arithmetic that has an incremental interface and provides unsatisfiable cores. We present a verification of their algorithm in Isabelle/HOL that significantly extends previous work by Spasić and Marić. Based on the simplex algorithm we further formalize Farkas' Lemma. With this result we verify that linear rational constraints are satisfiable over $\mathbb{Q}$ if and only they are satisfiable over $\mathbb{R}$. Hence, our verified simplex algorithm is also able to decide satisfiability in linear real arithmetic.

Keywords: DPLL(T) · Farkas' Lemma · Simplex algorithm · SMT solving

1 Introduction

CeTA [7] is a verified certifier for checking untrusted safety and termination proofs from external tools such as AProVE [12] and T2 [6]. To this end, CeTA also contains a verified SAT-modulo-theories (SMT) solver, since these untrusted proofs contain claims of validity of formulas. It is formalized as a deep embedding and is generated via code generation.

The ultimate aim of this work is the optimization of the existing verified SMT solver, as it is quite basic: The current solver takes as input a quantifier free formula in the theory of linear rational arithmetic, translates it into disjunctive normal form (DNF), and then tries to prove unsatisfiability for each conjunction of literals with the verified simplex implementation of Spasić and Marić [16]. This basic solver has at least two limitations: It only works on small formulas, since the conversion to DNF often leads to an exponential blowup in the formula size; and the procedure is restricted to linear *rational* arithmetic, i.e., the existing formalization only contain results on satisfiability over $\mathbb{Q}$, but not over $\mathbb{R}$.

Clearly, instead of the expensive DNF conversion, the better approach is to verify an SMT solver that is based on DPLL(T) or similar algorithms [4,11].

This research was supported by the Austrian Science Fund (FWF) project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

A. Herzig and A. Popescu (Eds.): FroCoS 2019, LNAI 11715, pp. 223–239, 2019.
https://doi.org/10.1007/978-3-030-29007-8_13

Although there has been recent success in verifying a DPLL-based SAT solver [2], for DPLL(T), a core component is missing, namely a powerful theory solver.

Therefore, in this paper we will extend the formalization of the simplex algorithm due to Spasić and Marić [16]. This will be an important milestone on the way to obtain a fully verified DPLL(T)-based SMT solver. To this end, we change the verified implementation and the existing soundness proofs in such a way that *minimal unsatisfiable cores* are computed instead of the algorithm merely indicating unsatisfiability. Moreover, we provide an *incremental interface* to the simplex method, as required by a DPLL(T) solver, which permits the incremental assertion of constraints, backtracking, etc. Finally, we formalize *Farkas' Lemma*, an important result that is related to duality in linear programming. In our setting, we utilize this lemma to formally verify that unsatisfiability of linear rational constraints over $\mathbb{Q}$ implies unsatisfiability over $\mathbb{R}$. In total, we provide a verified simplex implementation with an incremental interface, that generates minimal unsatisfiable cores over $\mathbb{Q}$ and $\mathbb{R}$.

We base our formalization entirely on the incremental simplex algorithm described by Dutertre and de Moura [10]. This paper was also the basis of the existing implementation by Spasić and Marić, of which the correctness has been formalized in Isabelle/HOL [14].

Although the sizes of the existing simplex formalization and of our new one differ only by a relatively small amount (8143 versus 11167 lines), the amount of modifications is quite significant: 2940 lines have been replaced by 5964 new ones. The verification of Farkas' Lemma and derived lemmas required another 1647 lines. It mainly utilizes facts that are proved in the existing simplex formalization, but it does not require significant modifications thereof.

The remainder of our paper is structured as follows. In Sect. 2 we describe the key parts of the simplex algorithm of Dutertre and de Moura and its formalization by Spasić and Marić. We present the development of the extended simplex algorithm with minimal unsatisfiable cores and incremental interfaces in Sect. 3. We formalize Farkas' Lemma and related results in Sect. 4. Finally, we conclude with Sect. 5.

Our formalization is available in the Archive of Formal Proofs (AFP) for Isabelle 2019 under the entries `Simplex` [13] and `Farkas` [5]. The `Simplex` entry contains the formalization of Spasić and Marić with our modifications and extensions. Our Isabelle formalization can be accessed by downloading the AFP, or by following the hyperlink at the beginning of each Isabelle code listing in Sects. 3 and 4.

Related Work. Allamigeon and Katz [1] formalized and verified an implementation of the simplex algorithm in Coq. Since their goal was to verify theoretical results about convex polyhedra, their formalization is considerably different from ours, as we aim at obtaining a practically efficient algorithm. For instance, we also integrate and verify an optimization of the simplex algorithm, namely the elimination of unused variables, cf. Dutertre and de Moura [10, end of Section 3]. This optimization also has not been covered by Spasić and Marić.

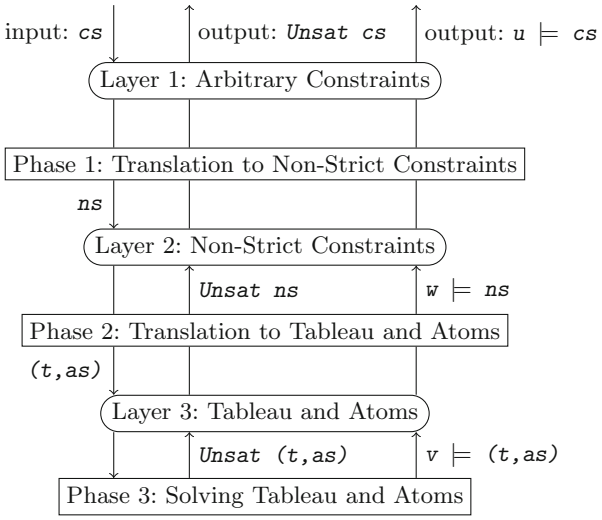

Fig. 1. The layers and phases of the simplex algorithm

Chaieb and Nipkow verified quantifier elimination procedures (QEP) for dense linear orders and integer arithmetic [9], which are more widely applicable than the simplex algorithm. Spasić and Marić compared the QEPs with their implementation on a set of random quantifier-free formulas [16]. In these tests, their (and therefore our) simplex implementation outperforms the QEPs significantly. Hence, neither of the formalizations subsumes the other.

There is also work on verified certification of SMT proofs, where an untrusted SMT solver outputs a certificate that is checked by a verified certifier. This is an alternative to the development of a verified SMT prover, but the corresponding Isabelle implementation of Böhme and Weber [3] is not usable in our setting, as it relies on internal Isabelle tactics, such as `linarith`, which are not accessible in Isabelle-generated code such as CeTA.

2 The Simplex Algorithm and the Existing Formalization

The simplex algorithm as described by Dutertre and de Moura is a decision procedure for the question whether a set of linear constraints is satisfiable over $\mathbb{Q}$. We briefly recall the main steps.

For the sake of the formalization, it is useful to divide the work of the algorithm into *phases*, and to think of the data available at the beginning and end of each phase as a *layer* (see Fig. 1). Thus, Layer 1 consists of the set of input constraints, which are (in)equalities of the form $p \sim c$, for some linear polynomial p, constant $c \in \mathbb{Q}$, and $\sim \in \{<, \leq, =, \geq, >\}$. Phase 1, the first preprocessing phase, transforms all constraints of Layer 1 into non-strict inequalities

involving δ-rationals, i.e. rationals in combination with a symbolic value δ, representing some small positive rational number.[1] In Phase 2, each constraint with exactly one variable is normalized; in all other constraints the linear polynomial is replaced by a new variable (a *slack variable*). Thus, Phase 2 produces a set of inequalities of the form $x \leq c$ or $x \geq c$, where x is a variable (such constraints are called *atoms*). Finally, the equations defining the newly introduced slack variables constitute a *tableau*, and a *valuation* (a function assigning a value to each variable) is taken initially to be the all-zero function.

At this point, the preprocessing phases have been completed. At the end of Phase 2, on Layer 3, we have a tableau of equations of the form $s_j = \sum a_i x_i$, where the s_j are slack variables, together with a set of atoms bounding both original and slack variables. The task now is to find a valuation that satisfies both the tableau and the atoms. This will be done by means of two operations, *assert* and *check*, that provide an incremental interface: assert adds an atom to the set of atoms that should be considered, and check decides the satisfiability of the tableau and currently asserted atoms. Both operations preserve the following invariant: Each variable occurs only on the left-hand or only on the right-hand side of tableau equations, and the valuation satisfies the tableau and the asserted atoms whose variables occur on the right-hand side of tableau equations.

In order to satisfy the invariant, the assert operation has to update the valuation whenever an atom is added whose variable is the right-hand side of the tableau. If this update conflicts with previously asserted atoms in an easily detectable way, assert itself can detect unsatisfiability at this point. Otherwise, it additionally recomputes the valuation of the left-hand side variables according to the equations in the tableau.

The main operation of Phase 3 is check, where the algorithm repeatedly modifies the tableau and valuation, aiming to satisfy all asserted atoms or detect unsatisfiability. The procedure by which the algorithm actually manipulates the tableau and valuation is called *pivoting*, and works as follows: First, it finds a tableau equation where the current valuation does not satisfy an asserted atom, A, involving the left-hand side variable, x. If no such x can be found, the current valuation satisfies the tableau and all asserted atoms. Otherwise, the procedure looks, in the same equation, for a right-hand side variable y for which the valuation can be modified so that the resulting value of x, as given by the equation, exactly matches the bound in A. If no such y can be found, the pivoting procedure concludes unsatisfiability. Otherwise, it updates the valuation for both x and y, and flips the sides of the two variables in the equation, resulting in an equation that defines y. The right-hand side of the new equation replaces all appearances of y on the right-hand side of other equations, ensuring that the invariant is maintained. Since y's updated value may no longer satisfy the asserted atoms involving y, it is not at all clear that repeated applications of pivoting eventually terminate. However, if the choice of variables during pivoting is done correctly, it can be shown that this is indeed the case.

[1] Arithmetic on δ-rationals is defined pointwise, e.g., $(a + b\delta) + (c + d\delta) := (a + c) + (b + d)\delta$, and $a + b\delta < c + d\delta := a < c \vee (a = c \wedge b < d)$ for any $a, b, c, d \in \mathbb{Q}$.

Step	Layer	A	B	C	D	Tableau	Val.
1	1	$x > 5$	$2x + y \leq 12$	$2y \geq 6$	$x - 3y \leq 2$	–	–
2	2	$x \geq 5 + \delta$	$2x + y \leq 12$	$2y \geq 6$	$x - 3y \leq 2$	–	–
3	3	$x \geq 5 + \delta$	$s \leq 12$	$y \geq 3$	$t \leq 2$	$\{s = 2x + y, t = x - 3y\}$	v_0
4	3	$\boldsymbol{x \geq 5 + \delta}$	$\boldsymbol{s \leq 12}$	$y \geq 3$	$\boldsymbol{t \leq 2}$	$\{s = 2x + y, t = x - 3y\}$	v_1
5	3	$\boldsymbol{x \geq 5 + \delta}$	$\boldsymbol{s \leq 12}$	$y \geq 3$	$\boldsymbol{t \leq 2}$	$\{s = \frac{7x-t}{3}, y = \frac{x-t}{3}\}$	v_2
6	3	$\boldsymbol{x \geq 5 + \delta}$	$\boldsymbol{s \leq 12}$	$\boldsymbol{y \geq 3}$	$\boldsymbol{t \leq 2}$	$\{s = \frac{7x-t}{3}, y = \frac{x-t}{3}\}$	v_2
7	3	$\boldsymbol{x \geq 5 + \delta}$	$\boldsymbol{s \leq 12}$	$\boldsymbol{y \geq 3}$	$\boldsymbol{t \leq 2}$	$\{t = \frac{s-7y}{2}, x = \frac{s-y}{2}\}$	v_3

Valuation v	$v(x)$	$v(y)$	$v(s)$	$v(t)$
v_0	0	0	0	0
v_1	$5 + \delta$	0	$10 + 2\delta$	$5 + \delta$
v_2	$5 + \delta$	$1 + \frac{1}{3}\delta$	$11 + \frac{7}{3}\delta$	2
v_3	$\frac{9}{2}$	3	12	$-\frac{9}{2}$

Fig. 2. Example run of the simplex algorithm

Consider the example in Fig. 2. The input constraints A–D are given in step 1 and converted into non-strict inequalities with δ-rationals in the step 2. In step 3, the constraint $2y \geq 6$ is normalized to the atom $y \geq 3$, two slack variables $s = 2x + y$ and $t = x - 3y$ are created, and the constraints $2x + y \leq 12$ and $x - 3y \leq 2$ are simplified accordingly. The equations defining s and t then form the initial tableau, and the initial valuation v_0 is the all-zero function. In step 4, the three atoms A, B and D are asserted (indicated by boldface font) and the valuation is updated accordingly. Next, the algorithm invokes check and performs pivoting to find the valuation v_2 that satisfies A, B, D and the tableau. This valuation on Layer 3 assigns δ-rationals to all variables x, y, s, t and can then be translated to a satisfying valuation over $\mathbb{Q}$ for constraints A, B, D on Layer 1. If the incremental interface is then used to also assert the atom C (step 6), unsatisfiability is detected via check after two further pivoting operations (step 7). Hence, the constraints A–D on Layer 1 are also unsatisfiable.

Spasić and Marić use Isabelle/HOL for the formalization, as do we for the extension. Isabelle/HOL is an interactive theorem prover for higher-order logic. Its syntax conforms to mathematical notation, and Isabelle supports keywords such as `fixes`, `assumes` and `shows`, allowing us to state theorems in Isabelle in a way which is close to mathematical language. Furthermore, all terms in Isabelle have a well-defined type, specified with a double-colon: *`term :: α`*. We use Greek letters for arbitrary types. Isabelle has built-in support for the types of rational numbers (*`rat`*) and real numbers (*`real`*). The type of a function *`f`* from type α to type β is specified as *`f :: α ⇒ β`*. There is a set type (α *`set`*), a list type (α *`list`*), an option type (α *`option`* with constructors *`Some :: α ⇒ α option`* and *`None :: α option`*) and a sum type ($\alpha + \beta$ with constructors *`Inl :: α ⇒ α+β`* and *`Inr :: β ⇒ α+β`*). The syntax for function application is *`f arg1 arg2`*. In this paper we use the terms Isabelle and Isabelle/HOL interchangeably.

Spasić and Marić proved the following main theorem about their simplex implementation `simplex :: rat constraint list ⇒ rat valuation option`.

```
lemma simplex_spasic_maric:
  shows simplex cs = None ⟶ ∄ v :: rat valuation. v ⊨ cs
  shows simplex cs = Some v ⟶ v ⊨ cs
```

The lemma states that if `simplex` returns no valuation, then the constraints `cs` are unsatisfiable. If `simplex` returns a valuation `Some v`, then `v` satisfies `cs`.

To prove the correctness of their algorithm they used a modular approach: Each subalgorithm (e.g. pivoting, incremental assertions) and its properties were specified in a *locale*, a special feature of Isabelle. Locales parameterize definitions and theorems over operations and assumptions. The overall algorithm is then implemented by combining several locales and their verified implementations. Soundness of the whole algorithm is then easily obtained via the locale structure. The modular structure of the formalization allows us to reuse, adapt and extend several parts of their formalization.

3 The New Simplex Formalization

In the following we describe our extension of the formalization of Spasić and Marić through the integration of minimal unsatisfiable cores (Sect. 3.1), the integration of an optimization during Phase 2 (Sect. 3.2) and the development of an incremental interface to the simplex algorithm (Sect. 3.3).

3.1 Minimal Unsatisfiable Cores

Our first extension is the integration of the functionality for producing unsatisfiable cores, i.e., given a set of unsatisfiable constraints, we seek a subset of the constraints which is still unsatisfiable. Small unsatisfiable cores are crucial for a DPLL(T)-based SMT solver in order to derive small conflict clauses, hence it is desirable to obtain *minimal* unsatisfiable cores, of which each proper subset is satisfiable. For example, in Fig. 2, $\{A, B, C\}$ is a minimal unsatisfiable core. We will refer to this example throughout this section.

Internally, the formalized simplex algorithm represents the data available on Layer 3 in a data structure called a *state*, which contains the current tableau, valuation, the set of asserted atoms,[2] and an unsatisfiability flag. Unsatisfiability is detected by the check operation in Phase 3, namely if the current valuation of a state does not satisfy the atoms, and pivoting is not possible.[3] For instance, in step 7 unsatisfiability is detected as follows: The valuation v_3 does not satisfy

[2] In the simplex algorithm [10] and the formalization, the asserted atoms are stored via *bounds*, but this additional data structure is omitted in the presentation here.

[3] Asserting an atom can also detect unsatisfiability, but this gives rise to trivial unsatisfiable cores of the form $\{x \leq c, x \geq d\}$ for constants $d > c$.

the atom $x \geq 5 + \delta$ since $v_3(x) = \frac{9}{2}$. The pivoting procedure looks at the tableau equation for x,

$$x = \frac{1}{2}s - \frac{1}{2}y, \tag{1}$$

and checks whether it is possible to increase the value of x. This is only possible if the valuation of s in increased (since s occurs with positive coefficient in (1)), or if y is decreased (since y occurs with a negative coefficient). Neither is possible, because $v_3(s)$ is already at its maximum ($s \leq 12$) and $v_3(y)$ at its minimum ($y \geq 3$). Hence, in order prove unsatisfiability on Layer 3, it suffices to consider the tableau and the atoms $\{x \geq 5 + \delta, s \leq 12, y \geq 3\}$.

We formally verify that this kind of reasoning works in general: Given the fact that some valuation v of a state does not satisfy an atom $x \geq c$ for some left-hand side variable x, we can obtain the corresponding equation $x = p$ of the tableau T, and take the unsatisfiable core as the set of atoms formed of: $x \geq c$, all atoms $y \geq v(y)$ for variables y of p with coefficient < 0, and all atoms $s \leq v(s)$ for variables s of p with coefficient > 0. The symmetric case $x \leq c$ is handled similarly by flipping signs.

We further prove that the generated cores are minimal w.r.t. the subset relation: Let A be a proper subset of an unsatisfiable core. There are two cases. If A does not contain the atom of the left-hand side variable x, then all atoms in A only contain right-hand side variables. Then by the invariant of the simplex algorithm, the current valuation satisfies both the tableau T and A. In the other case, some atom with a variable z of p is dropped. But then it is possible to apply pivoting for x and z. Let T' be the new tableau and v be the new valuation after pivoting. At this point we use the formalized fact that pivoting maintains the invariant. In particular, $v \models T'$ and $v \models A$, where the latter follows from the fact that A only contains right-hand side variables of the new tableau T' (note that x and z switched sides in the equation following pivoting). Since T and T' are equivalent, we conclude that v satisfies both T and A.

In the formalization, the corresponding lemma looks as follows:

```
lemma check_minimal_unsat_state_core: assumes ⊨_nolhs s and ◇ s and ...
  shows ¬ 𝒰 s ⟶ 𝒰 (check s) ⟶ minimal_unsat_state_core (check s)
```

The assumptions in the lemma express precisely the invariant of the simplex algorithm, and the lemma states that whenever the check operation sets the unsatisfiability flag $\mathcal{U}$, then indeed a minimal unsatisfiable core is stored in the new state `check s`. Whereas the assumptions have been taken unmodified from the existing simplex formalization, we needed to modify the formalized definition of the check operation and the datatype of states, so that `check` can compute and store the unsatisfiable core in the resulting state.

At this point, we have assembled a verified simplex algorithm for Layer 3 that will either return satisfying valuations or minimal unsatisfiable cores. The next task is to propagate the minimal unsatisfiable cores upwards to Layer 2 and 1, since, initially, the unsatisfiable cores are defined in terms of the data available at Layer 3, which is not meaningful when speaking about the first two layers.

A question that arises here is how to represent unsatisfiable cores. Taking the constraints literally is usually not a desirable solution, as then we would have to convert the atoms $\{x \geq 5 + \delta, s \leq 12, y \geq 3\}$ back to the non-strict constraints $\{x \geq 5 + \delta, 2x + y \leq 12, 2y \geq 6\}$ and further into $\{x > 5, 2x + y \leq 12, 2y \geq 6\}$, i.e., we would have to compute the inverses of the transformations in Phases 2 and 1. A far more efficient and simple solution is to use indexed constraints in the same way, as they already occur in the running example. Hence, the unsatisfiable core is just a set of indices ($\{A, B, C\}$ in our example). These indices are then valid for all layers and do not need any conversion.

Since the formalization of Spasić and Marić does not contain indices at all, we modify large parts of the source code so that it now refers to indexed constraints, i.e., we integrate indices into algorithms, data structures, definitions, locales, properties and proofs. For instance, indexed constraints *ics* are just sets of pairs, where each pair consists of an index and a constraint, and satisfiability of indexed constraints is defined as

$$(I, v) \models ics \qquad \text{if and only if} \qquad v \models \{c \mid (i, c) \in ics \wedge i \in I\},$$

where I is an arbitrary set of indices.

In order to be able to lift the unsatisfiable core from Layer 3 to the upper layers, we have to prove that the two transformations (elimination of strict inequalities and introduction of slack variables) maintain minimal unsatisfiable cores. To this end, we modify existing proofs for these transformation, since they are not general enough initially. For instance, the soundness statement for the introduction of slack variables in Phase 2 states that if the transformation on non-strict constraints N produces the tableau T and atoms A, then N and the combination of T and A are equisatisfiable, i.e.,

$$(\exists v.\ v \models N) \longleftrightarrow (\exists v.\ v \models T \wedge v \models A).$$

However, for lifting minimal unsatisfiable cores we need a stronger property, namely that the transformation is also sound for arbitrary indexed subsets I:[4]

$$(\exists v.\ (I, v) \models N) \longleftrightarrow (\exists v.\ v \models T \wedge (I, v) \models A). \tag{2}$$

Here, the indexed subsets in (2) are needed for both directions: given a minimal unsatisfiable core I of T and A, by the left-to-right implication of (2) we conclude that I is an unsatisfiable core of N, and it is minimal because of the right-to-left implication of (2). Note that tableau satisfiability ($v \models T$) is not indexed, since the tableau equations are global.

Our formalization therefore contains several new generalizations, e.g., the following lemma is the formal analogue to (2), where `preprocess` is the function that introduces slack variables. In addition to the tableau `t` and the indexed atoms `ias`, it also provides a computable function `trans_v` to convert satisfying valuations for `t` and `ias` into satisfying valuations for `ics`.

[4] This stronger property is also required, if the preprocessing is performed on the global formula, i.e., including the Boolean structure. The reason is that also there one needs soundness of the preprocessing for arbitrary subsets of the constraints.

```
lemma preprocess: assumes preprocess ics = (t, ias, trans_v)
  shows (I,v) ⊨ ias ⟶ v ⊨ t ⟶ (I, trans_v v) ⊨ ics
  shows (∃ v. (I,v) ⊨ ics) ⟶ (∃ v. (I,v) ⊨ ias ∧ v ⊨ t)
```

After all these modifications we obtain a simplex implementation that indeed provides minimal unsatisfiable cores. The corresponding function *simplex_index* returns a sum type, which is either a satisfying valuation or an unsatisfiable core represented by a set of indices.

```
lemma simplex_index:
  shows simplex_index ics = Inr v ⟶ v ⊨ {c | (i,c) ∈ ics}
  shows simplex_index ics = Inl I ⟶ ∄ v. (I,v) ⊨ ics
  shows simplex_index ics = Inl I ⟶ J ⊂ I ⟶
    distinct_indices ics ⟶ ∃ v. (J,v) ⊨ ics
```

Here, the minimality of the unsatisfiable cores can only be ensured if the indices in the input constraints are distinct. That distinctness is essential can easily be seen: Consider the following indexed constraints $\{(E, x \leq 3), (F, x \leq 5), (F, x \geq 10)\}$ where index F refers to two different constraints. If we invoke the verified simplex algorithm on these constraints, it detects that $x \leq 3$ is in conflict with $x \geq 10$ and hence produces $\{E, F\}$ as an unsatisfiable core. This core is clearly not minimal, however, since $\{F\}$ by itself is already unsatisfiable.

Some technical problems arise, regarding distinctness in combination with constraints involving equality. For example, the Layer 1-constraint $(G, p = c)$ will be translated into the two constraints $(G, p \geq c)$ and $(G, p \leq c)$ on Layer 2,[5] violating distinctness. These problems are solved by weakening the notion of distinct constraints on Layers 2 and 3, and strengthening the notion of a minimal unsatisfiable core for these layers: For each proper subset J of the unsatisfiable subset, each inequality has to be satisfied as if it were an equality, i.e., whenever there is some constraint $(j, p \leq c)$ or $(j, p \geq c)$ with $j \in J$, the satisfying valuation must fulfill $p = c$.

3.2 Elimination of Unused Variables in Phase 2

Directly after creating the tableau and the set of atoms from non-strict constraints in Phase 2, it can happen that there are *unused variables*, i.e., variables in the tableau for which no atoms exist.

Dutertre and de Moura propose to eliminate unused variables by Gaussian elimination [10, end of Section 3] in order to reduce the size of the tableau. We integrate this elimination of variables into our formalization. However, instead of using Gaussian elimination, we implement the elimination via pivoting. To be more precise, for each unused variable x we perform the following steps.

[5] Note that it is not possible to directly add equality constraints on Layer 1 to the tableau: First, this would invalidate the incremental interface, since the tableau constraints are global; second, the tableau forms a homogeneous system of equations, so it does not permit equations such as $x - y = 1$ which have a non-zero constant.

- If x is not already a left-hand side variable of the tableau, find any equation $y = p$ in the tableau that contains x, and perform pivoting of x and y, so that afterwards x is a left-hand side variable of the tableau.
- Drop the unique equation from the tableau that has x on its left-hand side, but remember the equation for reconstructing satisfying valuations.

Example 1. Consider the non-strict constraints $\{x+y \geq 5, x+2y \leq 7, y \geq 2\}$ on Layer 2. These are translated to the atoms $\{s \geq 5, t \leq 7, y \geq 2\}$ in combination with the tableau $\{s = x+y, t = x+2y\}$, so x becomes an unused variable. Since x is not a left-hand side variable, we perform pivoting of x and s and obtain the new tableau $\{x = s-y, t = s+y\}$. Then we drop the equation $x = s-y$ resulting in the smaller tableau $\{t = s+y\}$. Moreover, any satisfying valuation v for the variables $\{y, s, t\}$ will be extended to $\{x, y, s, t\}$ by defining $v(x) := v(s) - v(y)$.

In the formalization, the elimination has been integrated into the `preprocess` function of Sect. 3.1. In fact, `preprocess` just executes both preprocessing steps sequentially: first, the conversion of non-strict constraints into tableau and atoms, and afterwards the elimination of unused variables as described in this section. Interestingly, we had to modify the locale-structure of Spasić and Marić at this point, since preprocessing now depends on pivoting.

3.3 Incremental Simplex

The previous specifications of the simplex algorithm are monolithic: even if two (consecutive) inputs differ only in a single constraint, the functions `simplex` (in Sect. 2) and `simplex_index` (in Sect. 3.1) will start the computation from scratch. Hence, they do not specify an incremental simplex algorithm, despite the fact that an incremental interface is provided on Layer 3 via assert and check.

Since the incrementality of a theory solver is a crucial requirement for developing a DPLL(T)-based SMT solver, we will provide a formalization of the simplex algorithm that provides an incremental interface at each layer. Our design closely follows Dutertre and de Moura, who propose the following operations.

- Initialize the solver by providing the set of all possible constraints. This will return a state where none of these constraints have been asserted.
- Assert a constraint. This invokes a computationally inexpensive deduction algorithm and returns an unsatisfiable core or a new state.
- Check a state. Performs an expensive computation that decides satisfiability of the set of asserted constraints; returns an unsat core or a checked state.
- Extract a solution of a checked state.
- Compute some checkpoint information for a checked state.
- Backtrack to a state with the help of some checkpoint information.

Since a DPLL(T)-based SMT solver basically performs an exhaustive search, its performance can be improved considerably by having it keep track of checked states from which the search can be restarted in a different direction. This is why the checkpointing and backtracking functionality is necessary.

In Isabelle/HOL we specify this informal interface for each layer as a locale, which fixes the operations and the properties of that layer. For instance, the locale *Incremental_Simplex_Ops* is for Layer 1, where the type-variable σ represents the internal state for the layer, and γ is the checkpoint information.

```
locale Incremental_Simplex_Ops =
fixes init :: (ι × constraint) list ⇒ σ
  and assert :: ι ⇒ σ ⇒ ι list + σ
  and check :: σ ⇒ ι list + σ
  and solution :: σ ⇒ rat valuation
  and checkpoint :: σ ⇒ γ
  and backtrack :: γ ⇒ σ ⇒ σ
  and invariant :: (ι × constraint) list ⇒ ι set ⇒ σ ⇒ bool
  and checked :: (ι × constraint) list ⇒ ι set ⇒ σ ⇒ bool
assumes checked cs {} (init cs)
  and checked cs J s ⟶ invariant cs J s
  and invariant cs J s ⟶ assert j s = Inr s' ⟶
    invariant cs ({j} ∪ J) s'
  and invariant cs J s ⟶ assert j s = Inl I ⟶
    I ⊆ {j} ∪ J ∧ minimal_unsat_core I cs
  and invariant cs J s ⟶ check s = Inr s' ⟶ checked cs J s'
  and invariant cs J s ⟶ check s = Inl I ⟶
    I ⊆ J ∧ minimal_unsat_core I cs
  and checked cs J s ⟶ solution s = v ⟶ (J, v) ⊨ cs
  and checked cs J s ⟶ checkpoint s = c ⟶ invariant cs K s' ⟶
        backtrack c s' = s'' ⟶ J ⊆ K ⟶ invariant cs J s''
```

The interface consists of the six operations *init*, ..., *backtrack* to invoke the algorithm, and the two invariants *invariant* and *checked*, the latter of which entails the former.

Both invariants *invariant* and *checked* take the three arguments *cs*, *J* and *s*. Here, *cs* is the global set of indexed constraints that is encoded in the state *s*. It can only be set by invoking *init cs* and is kept constant otherwise. *J* indicates the set of all constraints that have been asserted in the state *s*.

We briefly explain the specification of *assert* and *backtrack* and leave the usage of the remaining functionality to the reader.

For the *assert* operation there are two possible outcomes. If the assertion of index *j* was successful, it returns a new state *s'* which satisfies the same invariant as *s*, and whose set of indices of asserted constraints contains *j*, and is otherwise the same as the corresponding set in *s*. Otherwise, the operation returns a set of indices *I*, which is a subset of the set of indices of asserted constraints (including *j*), such that the set of all *I*-indexed constraints is a minimal unsatisfiable core.

The backtracking facility works as follows. Assume that one has computed the checkpoint information *c* in a state *s*, which is only permitted if *s* satisfies the stronger invariant for some set of indices *J*. Afterwards, one may have

performed arbitrary operations and transitioned to a state s' corresponding to a superset of indices $K \supseteq J$. Then, solely from s' and c, one can compute via `backtrack` a new state s'' that corresponds to the old set of indices J. Of course, the implementation should be done in such a way that the size of c is small in comparison to the size of s; in particular, c should not be s itself. And, indeed, our implementation behaves in the same way as the informally described algorithm by Dutertre and de Moura: for a checkpoint c of state s we store the asserted atoms of the state s, but neither the valuation nor the tableau. These are taken from the state s' when invoking `backtrack c s'`.

In order to implement the incremental interface, we take the same modular approach as Spasić and Marić, namely that for each layer and its corresponding Isabelle locale, we rely upon the existing functionality of the various phases, together with the interface of the lower layers, to implement the locale.

In our case, a significant part of the work has already been done via the results described in Sect. 3.1: most of the generalizations that have been performed in order to support indexed constraints, play a role in proving the soundness of the incremental simplex implementation. In particular, the generalizations for Phases 1 and 2 are vital. For instance, the set of indices I in lemma `preprocess` on page 9 can not only be interpreted as an unsatisfiable core, but also as the set of currently asserted constraints. Therefore, `trans_v` allows us to convert a satisfying valuation on Layer 2 into a satisfying valuation on Layer 1 for the currently asserted constraints that are indexed by I. Consequently, the internal state of the simplex algorithm on Layer 1 not only stores the state of Layer 3 as it is described at the beginning of Sect. 3.1, but additionally stores the function `trans_v`, in order to compute satisfying valuations on Layer 1.

We further integrate and prove the correctness of the functionality of checkpointing and backtracking on all layers, since these features have not been formalized by Spasić and Marić. For instance, when invoking `backtrack c s'` on Layer 3 with `check_point s = c`, we obtain a new state that contains the tableau t' and valuation v' of state s', but the asserted atoms `as` of state s. Hence, we need to show that v' satisfies those asserted atoms of `as` that correspond to right-hand side variables of t'. To this end, we define the invariant on Layer 3 in a way that permits us to conclude that the tableaux t and t' are equivalent. Using this equivalence, we then formalize the desired result for Layer 3. Checkpointing and backtracking on the other layers is just propagated to the next-lower layers, i.e., no further checkpointing information is required on Layers 1 and 2.

Finally, we combine the implementations of all phases and layers to obtain a fully verified implementation of the simplex algorithm w.r.t. the specification defined in the locale `Incremental_Simplex_Ops`.

Note that the incremental interface does not provide a function to assert constraints negatively. However, this limitation is easily circumvented by passing both the positive and the negative constraint with different indices to the `init` function. For example, instead of using $(A, x > 5)$ as in Fig. 2, one can use the two constraints $(+A, x > 5)$ and $(-A, x \leq 5)$. Then one can assert both the original and the negated constraint via indices $+A$ and $-A$, respectively.

Only the negation of equations is not possible in this way, since this would lead to disjunctions. However, each equation can easily be translated into the conjunction of two inequalities on the formula-level, i.e., they can be eliminated within a preprocessing step of the SMT-solver.

4 A Formalized Proof of Farkas' Lemma

Farkas' Lemma states that a system of linear constraints is unsatisfiable if and only if there is a linear combination of the constraints that evaluates to a trivially unsatisfiable inequality (e.g. $0 \leq d$ for a constant $d < 0$). The non-zero coefficients in such a linear combination are referred to as *Farkas coefficients*, and can be thought of as an easy-to-check certificate for the unsatisfiability of a set of linear constraints (given the coefficients, one can simply evaluate the corresponding linear combination and check that the result is indeed unsatisfiable.)

One way to prove Farkas' Lemma is by using the Fundamental Theorem of Linear Inequalities; this theorem can in turn be proved in the same way as the fact that the simplex algorithm terminates (see [15, Chapter 7]). Although Spasić and Marić have formalized a proof of termination for their simplex implementation [16], this is not sufficient to immediately prove Farkas' Lemma. Instead, our formalization of the result begins at the point where the simplex algorithm detects unsatisfiability in Phase 3, because this is the only point in the execution of the algorithm where Farkas coefficients can be computed directly from the available data.[6] Then, these coefficients need to be transferred up to Layer 1. In the following we illustrate how Farkas coefficients are computed and propagated through the various phases of the algorithm, by giving examples and explaining, informally, intermediate statements that have been formalized.

To illustrate how Farkas coefficients are determined at the point where the check-operation detects unsatisfiability in Phase 3, let us return once more to the example in Fig. 2. In step 7, the algorithm detects unsatisfiability via the equation $x = \frac{s-y}{2}$, and generates the unsatisfiable core based on this equation. This equality can also be used to obtain Farkas coefficients. To this end, we rewrite the equation as $-x + \frac{1}{2}s - \frac{1}{2}y = 0$, and use the coefficients in this equation (-1 for x, $\frac{1}{2}$ for s, and $-\frac{1}{2}$ for y) to form a linear combination of the corresponding atoms involving the variables:

$$\begin{aligned} & -(x \geq 5 + \delta) + \frac{1}{2}(s \leq 12) - \frac{1}{2}(y \geq 3) \qquad \text{(FC3)} \\ &= (-x \leq -5 - \delta) + \left(\frac{1}{2}s \leq 6\right) + \left(-\frac{1}{2}y \leq -\frac{3}{2}\right) \\ &= \underbrace{\left(-x + \frac{1}{2}s - \frac{1}{2}y\right)}_{p} \leq \underbrace{\left(-\delta - \frac{1}{2}\right)}_{d}, \end{aligned}$$

[6] Again, we here consider only the check operation, since obtaining Farkas coefficients for a conflict detected by assert is trivial, cf. footnote 3.

where $p = 0$ is a reformulation of an equation of the tableau and d is a negative constant. Consequently, we show in the formalization that whenever unsatisfiability is detected for a given tableau T and set of atoms A, there exist Farkas coefficients r_i, i.e., that there is a linear combination $(\sum r_i a_i) = (p \leq d)$, where $d < 0$, $a_i \in A$ for all i, each $r_i a_i$ is a $\leq$-inequality, and $T \models p = 0$. The second-to-last condition ensures that only inequalities are added which are oriented in the same direction, so that the summation is well-defined. The condition $T \models p = 0$ means that for every valuation that satisfies T, p evaluates to 0.

Recall that before detecting unsatisfiability, several pivoting steps may have been applied, e.g., when going from step 3 to step 7. Hence, it is important to verify that Farkas coefficients are preserved by pivoting. This is easily achieved by using our notion of Farkas coefficients: Spasić and Marić formally proved that pivoting changes the tableau T' into an equivalent tableau T, and, hence, the condition $T \models p = 0$ immediately implies $T' \models p = 0$. In the example, we conclude that $T' \models -x + \frac{1}{2}s - \frac{1}{2}y = 0$ for any tableau T' in steps 3–7. Thus, (FC3) provides Farkas coefficients for the atoms and tableau mentioned in any of these steps.

Layer 2 requires a new definition of Farkas coefficients, since there is no tableau T and set of atoms A at this point, but a set N of non-strict constraints. The new definition is similar to the one on Layer 3, except that the condition $T \models p = 0$ is dropped, and instead we require that $p = 0$. To be precise, r_i are Farkas coefficients for N if there is a linear combination $(\sum r_i c_i) = (0 \leq d)$ where $d < 0$, $c_i \in N$ for all i, and each $r_i c_i$ is a $\leq$-inequality.

We prove that the preprocessing done in Phase 2 allows for the transformation of Farkas coefficients for Layer 3 to Farkas coefficients for Layer 2. In essence, the same coefficients r_i can be used, one just has to replace each atom a_i by the corresponding constraint c_i. The only exception is that if a constraint c_i has been normalized, then one has to multiply the corresponding r_i by the same factor. However, this will not change the constant d, and we formally verify that the polynomial resulting from the summation will indeed be 0.

In the example, we would obtain (FC2) for Layer 2. Here, the third coefficient has been changed from $-\frac{1}{2}$ to $-\frac{1}{2} \cdot \frac{1}{2} = -\frac{1}{4}$, where the latter $\frac{1}{2}$ is the factor used when normalizing the constraint $2y \geq 6$ to obtain the atom $y \geq 3$.

$$-(x \geq 5 + \delta) + \frac{1}{2}(2x + y \leq 12) - \frac{1}{4}(2y \geq 6) = \left(0 \leq -\delta - \frac{1}{2}\right) \tag{FC2}$$

Finally, for Layer 1 the notion of Farkas coefficients must once again be redefined so as to work with a more general constraint type that also allows strict constraints. In particular, we have that either the sum of inequalities is strict and the constant d is non-positive, or the sum of inequalities is non-strict and d is negative. In the example we obtain (again with the same coefficients, but using the original, possibly strict inequalities in the linear combination):

$$-(x > 5) + \frac{1}{2}(2x + y \leq 12) - \frac{1}{4}(2y \geq 6) = \left(0 < -\frac{1}{2}\right). \tag{FC1}$$

Farkas coefficients r_i on Layer 2 are easily translated to Layer 1, since no change is required, i.e., the same coefficients r_i can be used. We just prove that whenever the resulting inequality in Layer 2 is $0 \leq d$ for $d = a + b\delta$ with $a, b \in \mathbb{Q}$, then the sum of inequalities on Layer 1 will be $0 \leq a$ (and $b = 0$), or it will be $0 < a$. In both cases we use the property that $a + b\delta = d$ is negative, to show that the r_i are Farkas coefficients for Layer 1.

We illustrate the results of our formalization of Farkas coefficients by providing the formal statements for two layers. In both lemmas, *cs* is a set of linear constraints of the form $p \sim d$ for a linear polynomial p, constant d and $\sim \in \{\leq, <\}$. Here, the first theorem is an Isabelle statement of [8, Lemma 2], i.e., Farkas' Lemma over δ-rationals. The second theorem is a more general version of Farkas' Lemma which also permits strict inequalities, i.e., our statement on Layer 1. It is known as Motzkin's Transposition Theorem [15, Cor. 7.1k] or the Kuhn–Fourier Theorem [17, Thm. 1.1.9].

```
lemma Farkas'_Lemma_Delta_Rationals: assumes finite cs
    and ∀ c ∈ cs. ∃ p d. c = (p ≤ d)            (* only ≤−constraints *)
  shows (∄ v :: QDelta valuation. v ⊨ cs) ⟷
    (∃ C d. d < 0 ∧ (∀ (r, c) ∈ C. r > 0 ∧ c ∈ cs)
     ∧ (Σ(r,c) ← C. r · c) = (0 ≤ d))
```

```
theorem Motzkin's_transposition_theorem: assumes finite cs
  shows (∄ v :: rat valuation. v ⊨ cs) ⟷
    (∃ C ineq d. (∀ (r, c) ∈ C. r > 0 ∧ c ∈ cs)
     ∧ (Σ (r,c) ← C. r · c) = ineq
     ∧ ((ineq = (0 ≤ d) ∧ d < 0) ∨ (ineq = (0 < d) ∧ d ≤ 0)))
```

The existence of Farkas coefficients not only implies unsatisfiability over $\mathbb{Q}$, but also unsatisfiability over $\mathbb{R}$: lifting the summation of linear inequalities from $\mathbb{Q}$ to $\mathbb{R}$ yields the same conflict $0 \leq d$, with d negative, over the reals. Hence, we formalize the property that satisfiability of linear rational constraints over $\mathbb{Q}$ and over $\mathbb{R}$ are the same. Consequently, the (incremental) simplex algorithm is also able to prove unsatisfiability over $\mathbb{R}$.

```
lemma rat_real_conversion: assumes finite (cs :: rat constraint set)
  shows (∃ v :: rat valuation. v ⊨ cs)
   ⟷ (∃ v :: real valuation. v ⊨ cs)
```

Note that the finiteness condition of the set of constraints in the previous three statements mainly arose from the usage of the simplex algorithm for doing the underlying proofs, since the simplex algorithm only takes finite sets of constraints as input. However, the finiteness of the constraint set is actually a necessary condition, regardless of how the statements are proved: none of the three properties hold for infinite sets of constraints. For instance, the constraint set $\{x \geq c \mid c \in \mathbb{N}\}$ is unsatisfiable over $\mathbb{Q}$, but there are no Farkas coefficients for these constraints. Moreover, the rational constraints $\{x \geq c \mid c \leq \pi, c \in \mathbb{Q}\} \cup \{x \leq c \mid c \geq \pi, c \in \mathbb{Q}\}$ have precisely one real solution, $v(x) = \pi$, but there is no rational solution since $\pi \notin \mathbb{Q}$.

5 Conclusion

We have presented our development of an Isabelle/HOL formalization of a simplex algorithm with minimal unsatisfiable core generation and an incremental interface. Furthermore, we gave a verified proof of Farkas' Lemma, one of the central results in the theory of linear inequalities. Both of these contributions are related to the simplex formalization of Spasić and Marić [16]: the incremental simplex formalization is an extension built on top of their work, and the formal proof of Farkas' Lemma follows their simplex implementation through the phases of the algorithm.

In our formalization we use locales as the main structuring technique for obtaining modular proofs – as was done by Spasić and Marić. Our formal proofs were mainly written interactively, with frequent use of `find_theorems` rather than `sledgehammer` (which only provided a few externally generated proofs).

Both of our contributions form a crucial stepping stone towards our initial goal, the development of a verified SMT solver that is based on the DPLL(T) approach and supports linear arithmetic over $\mathbb{Q}$ and $\mathbb{R}$. The connection of the theory solver and the verified DPLL-based SAT solver [2] remains as future work. Here, we already got in contact with Mathias Fleury to initiate some collaboration. However, he immediately informed us that the connection itself will be a non-trivial task on its own. One issue is that his SAT solver is expressed in the imperative monad, but in our use case we need to apply it outside this monad, i.e., it should have a purely functional type such as *formula* ⇒ *bool*.

Acknowledgments. We thank the reviewers and Mathias Fleury for constructive feedback.

References

1. Allamigeon, X., Katz, R.D.: A formalization of convex polyhedra based on the simplex method. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 28–45. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_3
2. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SATsolver framework with learn, forget, restart, and incrementality. J. Autom. Reasoning **61**(1–4), 333–365 (2018). https://doi.org/10.1007/s10817-018-9455-7
3. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_14
4. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Proofs in conflict-driven theory combination. In: 7th ACM SIGPLAN International Conference Certified Programs and Proofs, CPP 2018, pp. 186–200. ACM (2018). https://doi.org/10.1145/3167096
5. Bottesch, R., Haslbeck, M.W., Thiemann, R.: Farkas' Lemma and Motzkin's Transposition Theorem. Archive of Formal Proofs, January 2019. http://isa-afp.org/entries/Farkas.html. Formal proof development

6. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: temporal property verification. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 387–393. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_22
7. Brockschmidt, M., Joosten, S.J.C., Thiemann, R., Yamada, A.: Certifying safety and termination proofs for integer transition systems. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 454–471. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_28
8. Bromberger, M., Weidenbach, C.: New techniques for linear arithmetic: cubes and equalities. Formal Methods Syst. Des. **51**(3), 433–461 (2017). https://doi.org/10.1007/s10703-017-0278-7
9. Chaieb, A., Nipkow, T.: Proof synthesis and reflection for linear arithmetic. J. Autom. Reasoning **41**(1), 33 (2008). https://doi.org/10.1007/s10817-008-9101-x
10. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
11. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_14
12. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reasoning **58**, 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
13. Marić, F., Spasić, M., Thiemann, R.: An incremental simplex algorithm with unsatisfiable core generation. Archive of Formal Proofs, August 2018. http://isa-afp.org/entries/Simplex.html. Formal proof development
14. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
15. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Hoboken (1999)
16. Spasić, M., Marić, F.: Formalization of incremental simplex algorithm by stepwise refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 434–449. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_35
17. Stoer, J., Witzgall, C.: Convexity and Optimization in Finite Dimensions I. Die Grundlehren der mathematischen Wissenschaften, vol. 163 (1970). https://www.springer.com/gp/book/9783642462184