Farkas' Lemma and Motzkin's Transposition Theorem*

Ralph Bottesch   Max W. Haslbeck   René Thiemann

October 13, 2025

Abstract

We formalize a proof of Motzkin's transposition theorem and Farkas' lemma in Isabelle/HOL. Our proof is based on the formalization of the simplex algorithm which, given a set of linear constraints, either returns a satisfying assignment to the problem or detects unsatisfiability. By reusing facts about the simplex algorithm we show that a set of linear constraints is unsatisfiable if and only if there is a linear combination of the constraints which evaluates to a trivially unsatisfiable inequality.

Contents

*Supported by FWF (Austrian Science Fund) project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

1 Introduction

This formalization augments the existing formalization of the simplex algorithm [3, 5, 7]. Given a system of linear constraints, the simplex implementation in [3] produces either a satisfying assignment or a subset of the given constraints that is itself unsatisfiable. Here we prove some variants of Farkas' Lemma. In essence, it states that if a set of constraints is unsatisfiable, then there is a linear combination of these constraints that evaluates to an unsatisfiable inequality of the form $0 \leq c$, for some negative c.

Our proof of Farkas' Lemma [4, Cor. 7.1e] relies on the formalized simplex algorithm: Under the assumption that the algorithm has detected unsatisfiability, we show that there exist coefficients for the above-mentioned linear combination of the input constraints.

Since the formalized algorithm follows the structure of the simplex-algorithm by Dutertre and de Moura [2], it first goes through a number of preprocessing phases, before starting the simplex procedure in earnest. These are relevant for proving Farkas' Lemma. We distinguish four *layers* of the algorithm; at each layer, it operates on data that is a refinement of the data available at the previous layer.

- *Layer 1. Data*: the input – a set of linear constraints with rational coefficients. These can be equalities or strict/non-strict inequalities. *Preprocessing*: Each equality is split into two non-strict inequalities, strict inequalities are replaced by non-strict inequalities involving δ-rationals.
- *Layer 2. Data*: a set of linear constraints that are non-strict inequalities with δ-rationals. *Preprocessing*: Linear constraints are simplified so that each constraint involves a single variable, by introducing so-called slack variables where necessary. The equations defining the slack variables are collected in a *tableau*. The constraints are normalized so that they are of the form $y \leq c$ or $y \geq c$ (these are called *atoms*).
- *Layer 3. Data*: A tableau and a set of atoms. Here the algorithm initializes the simplex algorithm.
- *Layer 4. Data*: A tableau, a set of atoms and an assignment of the variables. The simplex procedure is run.

At the point in the execution where the simplex algorithm detects unsatisfiability, we can directly obtain coefficients for the desired linear combination. However, these coefficients must then be propagated backwards through the different layers, where the constraints themselves have been modified, in order to obtain coefficients for a linear combination of *input* constraints. These propagation steps make up a large part of the formalized proof, since we must show, at each of the layers 1–3, that the existence of coefficients at the layer below translates into the existence of such coefficients for the current layer. This means, in particular, that we formulate and prove a version of Farkas' Lemma for each of the four layers, in terms of the data available at the respective level. The theorem we obtain at Layer 1 is actually a more general version of Farkas' lemma, in the sense that it allows for strict as well as non-strict inequalities, known as Motzkin's Transposition Theorem [4, Cor. 7.1k] or the Kuhn–Fourier Theorem [6, Thm. 1.1.9].

Since the implementation of the simplex algorithm in [3], which our work relies on, is restricted to systems of constraints over the rationals, this formalization is also subject to the same restriction.

2 Farkas Coefficients via the Simplex Algorithm of Duterte and de Moura

Let $c_1, \ldots, c_n$ be a finite list of linear inequalities. Let C be a list of pairs (r_i, c_i) where r_i is a rational number. We say that C is a list of *Farkas coefficients* if the sum of all products $r_i \cdot c_i$ results in an inequality that is trivially unsatisfiable.

Farkas' Lemma states that a finite set of non-strict linear inequalities is unsatisfiable if and only if Farkas coefficients exist. We will prove this lemma with the help of the simplex algorithm of Dutertre and de Moura's.

Note that the simplex implementation works on four layers, and we will formulate and prove a variant of Farkas' Lemma for each of these layers.

theory *Farkas*
  imports *Simplex.Simplex*
begin

2.1 Linear Inequalities

Both Farkas' Lemma and Motzkin's Transposition Theorem require linear combinations of inequalities. To this end we define one type that permits strict and non-strict inequalities which are always of the form "polynomial R constant" where R is either $\leq$ or $<$. On this type we can then define a commutative monoid.

A type for the two relations: less-or-equal and less-than.

datatype *le-rel* = *Leq-Rel* | *Lt-Rel*

primrec *rel-of* :: *le-rel* ⇒ *'a* :: *lrv* ⇒ *'a* ⇒ *bool* **where**
  rel-of Leq-Rel = (≤)
| *rel-of Lt-Rel* = (<)

instantiation *le-rel* :: *comm-monoid-add* **begin**
definition *zero-le-rel* = *Leq-Rel*

fun *plus-le-rel* **where**
plus-le-rel Leq-Rel Leq-Rel = *Leq-Rel*
| *plus-le-rel* - - = *Lt-Rel*
instance
proof
fix *a b c* :: *le-rel*
show *a* + *b* + *c* = *a* + (*b* + *c*) **by** (*cases a*; *cases b*; *cases c*, *auto*)
show *a* + *b* = *b* + *a* **by** (*cases a*; *cases b*, *auto*)
show *0* + *a* = *a* **unfolding** *zero-le-rel-def* **by** (*cases a*, *auto*)
qed
end

lemma *Leq-Rel-0*: *Leq-Rel* = *0* **unfolding** *zero-le-rel-def* **by** *simp*

datatype *'a le-constraint* = *Le-Constraint* (*lec-rel*: *le-rel*) (*lec-poly*: *linear-poly*) (*lec-const*: *'a*)

abbreviation (*input*) *Leqc* ≡ *Le-Constraint Leq-Rel*

instantiation *le-constraint* :: (*lrv*) *comm-monoid-add* **begin**
fun *plus-le-constraint* :: *'a le-constraint* ⇒ *'a le-constraint* ⇒ *'a le-constraint* **where**
plus-le-constraint (*Le-Constraint r1 p1 c1*) (*Le-Constraint r2 p2 c2*) =
(*Le-Constraint* (*r1* + *r2*) (*p1* + *p2*) (*c1* + *c2*))

definition *zero-le-constraint* :: *'a le-constraint* **where**
zero-le-constraint = *Leqc 0 0*

instance proof
fix *a b c* :: *'a le-constraint*
show *0* + *a* = *a*
by (*cases a*, *auto simp*: *zero-le-constraint-def Leq-Rel-0*)
show *a* + *b* = *b* + *a* **by** (*cases a*; *cases b*, *auto simp*: *ac-simps*)
show *a* + *b* + *c* = *a* + (*b* + *c*) **by** (*cases a*; *cases b*; *cases c*, *auto simp*: *ac-simps*)
qed
end

primrec *satisfiable-le-constraint* :: *'a::lrv valuation* ⇒ *'a le-constraint* ⇒ *bool* (**infixl** ‹$\models_{le}$› *100*) **where**
($v \models_{le}$ (*Le-Constraint rel l r*)) ⟷ (*rel-of rel* (*l*⦃*v*⦄) *r*)

lemma *satisfies-zero-le-constraint*: $v \models_{le}$ *0*
by (*simp add*: *valuate-zero zero-le-constraint-def*)

lemma *satisfies-sum-le-constraints*:
assumes $v \models_{le} c$ $v \models_{le} d$
shows $v \models_{le} (c + d)$
proof −
obtain *lc rc ld rd rel1 rel2* **where** *cd*: *c* = *Le-Constraint rel1 lc rc d* = *Le-Constraint rel2 ld rd*

```
    by (cases c; cases d, auto)
  have 1: rel-of rel1 (lc⦃v⦄) rc using assms cd by auto
  have 2: rel-of rel2 (ld⦃v⦄) rd using assms cd by auto
  from 1 have le1: lc⦃v⦄ ≤ rc by (cases rel1, auto)
  from 2 have le2: ld⦃v⦄ ≤ rd by (cases rel2, auto)
  from 1 2 le1 le2 have rel-of (rel1 + rel2) ((lc⦃v⦄) + (ld⦃v⦄)) (rc + rd)
    apply (cases rel1; cases rel2; simp add: add-mono)
    by (metis add.commute le-less-trans order.strict-iff-order plus-less)+
  thus ?thesis by (auto simp: cd valuate-add)
qed

lemma satisfies-sumlist-le-constraints:
  assumes ⋀ c. c ∈ set (cs :: 'a :: lrv le-constraint list) ⟹ v ⊨_le c
  shows v ⊨_le sum-list cs
  using assms
  by (induct cs, auto intro: satisfies-zero-le-constraint satisfies-sum-le-constraints)

lemma sum-list-lec:
  sum-list ls = Le-Constraint
    (sum-list (map lec-rel ls))
    (sum-list (map lec-poly ls))
    (sum-list (map lec-const ls))
proof (induct ls)
  case Nil
  show ?case by (auto simp: zero-le-constraint-def Leq-Rel-0)
next
  case (Cons l ls)
  show ?case by (cases l, auto simp: Cons)
qed

lemma sum-list-Leq-Rel: ((∑ x←C. lec-rel (f x)) = Leq-Rel) ⟷ (∀ x ∈ set C.
lec-rel (f x) = Leq-Rel)
proof (induct C)
  case (Cons c C)
  show ?case
  proof (cases lec-rel (f c))
    case Leq-Rel
    show ?thesis using Cons by (simp add: Leq-Rel Leq-Rel-0)
  qed simp
qed (simp add: Leq-Rel-0)
```

2.2 Farkas' Lemma on Layer 4

On layer 4 the algorithm works on a state containing a tableau, atoms (or bounds), an assignment and a satisfiability flag. Only non-strict inequalities appear at this level. In order to even state a variant of Farkas' Lemma on layer 4, we need conversions from atoms to non-strict constraints and then further to linear inequalities of type *le-constraint*. The latter conversion is a partial operation, since non-strict constraints of type *ns-constraint* permit greater-or-equal constraints, whereas *le-constraint* allows only less-or-equal.

The advantage of first going via *ns-constraint* is that this type permits a multiplication with arbitrary rational numbers (the direction of the inequality must be flipped when multiplying by a negative number, which is not possible with *le-constraint*).

instantiation *ns-constraint* :: (*scaleRat*) *scaleRat*
begin
fun *scaleRat-ns-constraint* :: *rat* ⇒ *'a ns-constraint* ⇒ *'a ns-constraint* **where**
  scaleRat-ns-constraint r (*LEQ-ns p c*) =
    (*if* (*r* < *0*) *then GEQ-ns* (*r* ∗*R p*) (*r* ∗*R c*) *else LEQ-ns* (*r* ∗*R p*) (*r* ∗*R c*))
| *scaleRat-ns-constraint r* (*GEQ-ns p c*) =
    (*if* (*r* > *0*) *then GEQ-ns* (*r* ∗*R p*) (*r* ∗*R c*) *else LEQ-ns* (*r* ∗*R p*) (*r* ∗*R c*))

instance **..**
end

lemma *sat-scale-rat-ns*: **assumes** $v \models_{ns}$ *ns*
  shows $v \models_{ns}$ (*f* ∗*R ns*)
proof −
  have *f* < *0* | *f* = *0* | *f* > *0* **by** *auto*
  then show *?thesis* **using** *assms* **by** (*cases ns*, *auto simp*: *valuate-scaleRat scaleRat-leq1 scaleRat-leq2*)
qed

lemma *scaleRat-scaleRat-ns-constraint*: **assumes** *a* ≠ *0* ⟹ *b* ≠ *0*
  shows *a* ∗*R* (*b* ∗*R* (*c* :: *'a* :: *lrv ns-constraint*)) = (*a* ∗ *b*) ∗*R c*
proof −
  have *b* > *0* ∨ *b* < *0* ∨ *b* = *0* **by** *linarith*
  moreover have *a* > *0* ∨ *a* < *0* ∨ *a* = *0* **by** *linarith*
  ultimately show *?thesis* **using** *assms*
    by (*elim disjE*; *cases c*, *auto simp add*: *not-le not-less*
      mult-neg-pos mult-neg-neg mult-nonpos-nonneg mult-nonpos-nonpos mult-nonneg-nonpos mult-pos-neg)
qed

fun *lec-of-nsc* **where**
  lec-of-nsc (*LEQ-ns p c*) = (*Leqc p c*)

fun *is-leq-ns* **where**
  is-leq-ns (*LEQ-ns p c*) = *True*
| *is-leq-ns* (*GEQ-ns p c*) = *False*

lemma *lec-of-nsc*:
  assumes *is-leq-ns c*
  shows ($v \models_{le}$ *lec-of-nsc c*) ⟷ ($v \models_{ns}$ *c*)
  using *assms* **by** (*cases c*, *auto*)

fun *nsc-of-atom* **where**

nsc-of-atom (*Leq var b*) = *LEQ-ns* (*lp-monom 1 var*) *b*
| *nsc-of-atom* (*Geq var b*) = *GEQ-ns* (*lp-monom 1 var*) *b*

lemma *nsc-of-atom*: $v \models_{ns}$ *nsc-of-atom a* $\longleftrightarrow v \models_a a$
by (*cases a*, *auto*)

We say that C is a list of Farkas coefficients *for a given tableau t and atom set as*, if it is a list of pairs (r, a) such that $a \in as$, r is non-zero, $r \cdot a$ is a 'less-than-or-equal'-constraint, and the linear combination of inequalities must result in an inequality of the form $p \leq c$, where $c < 0$ and $t \models p = 0$.

definition *farkas-coefficients-atoms-tableau* **where**
farkas-coefficients-atoms-tableau (*as* :: *'a* :: *lrv atom set*) *t C* = ($\exists$ *p c*.
($\forall$ (*r*,*a*) $\in$ *set C*. *a* $\in$ *as* $\wedge$ *is-leq-ns* (*r* **R nsc-of-atom a*) $\wedge$ *r* $\neq$ *0*) $\wedge$
($\sum$ (*r*,*a*) $\leftarrow$ *C*. *lec-of-nsc* (*r* **R nsc-of-atom a*)) = *Leqc p c* $\wedge$
c < *0* $\wedge$
($\forall$ *v* :: *'a valuation*. *v* $\models_t$ *t* $\longrightarrow$(*p*⦃*v*⦄ = *0*)))

We first prove that if the check-function detects a conflict, then Farkas coefficients do exist for the tableau and atom set for which the conflict is detected.

definition *bound-atoms* :: (*'i*, *'a*) *state* $\Rightarrow$ *'a atom set* (‹$\mathcal{B}_A$›) **where**
bound-atoms s = (λ(*v*,*x*). *Geq v x*) ‘ (*set-of-map* ($\mathcal{B}_l$ *s*)) $\cup$
(λ(*v*,*x*). *Leq v x*) ‘ (*set-of-map* ($\mathcal{B}_u$ *s*))

context *PivotUpdateMinVars*
begin

lemma *farkas-check*:
assumes *check*: *check s'* = *s* **and** *U*: $\mathcal{U}$ *s* ¬ $\mathcal{U}$ *s'*
and *inv*: ∇ *s'* $\triangle$ ($\mathcal{T}$ *s'*) $\models_{nolhs}$ *s'* $\Diamond$ *s'*
and *index*: *index-valid as s'*
shows $\exists$ *C*. *farkas-coefficients-atoms-tableau* (*snd* ‘ *as*) ($\mathcal{T}$ *s'*) *C*
proof −
let *?Q* = λ *s f p c C*. *set C* $\subseteq$ $\mathcal{B}_A$ *s* $\wedge$
distinct C $\wedge$
($\forall$ *a* $\in$ *set C*. *is-leq-ns* (*f* (*atom-var a*) **R nsc-of-atom a*) $\wedge$ *f* (*atom-var a*) $\neq$ *0*) $\wedge$
($\sum$ *a* $\leftarrow$ *C*. *lec-of-nsc* (*f* (*atom-var a*) **R nsc-of-atom a*)) = *Leqc p c* $\wedge$
c < *0* $\wedge$
($\forall$ *v* :: *'a valuation*. *v* $\models_t$ $\mathcal{T}$ *s* $\longrightarrow$(*p*⦃*v*⦄ = *0*))
let *?P* = λ *s*. $\mathcal{U}$ *s* $\longrightarrow$ ($\exists$ *f p c C*. *?Q s f p c C*)
have *?P* (*check s'*)
proof (*induct rule*: *check-induct''*[*OF inv*, *of ?P*])
case (*3 s* x_i *dir I*)
have *dir*: *dir* = *Positive* $\vee$ *dir* = *Negative* **by** *fact*
let *?eq* = (*eq-for-lvar* ($\mathcal{T}$ *s*) x_i)
define X_j **where** X_j = *rvars-eq ?eq*
define XL_j **where** XL_j = *Abstract-Linear-Poly.vars-list* (*rhs ?eq*)
have [*simp*]: *set* XL_j = X_j **unfolding** XL_j*-def* X_j*-def*

using *set-vars-list* **by** *blast*
have XL_j*-distinct*: *distinct* XL_j
unfolding XL_j*-def* **using** *distinct-vars-list* **by** *simp*
define *A* **where** *A* = *coeff* (*rhs ?eq*)
have *bounds-id*: $\mathcal{B}_A$ (*set-unsat I s*) = $\mathcal{B}_A$ *s* $\mathcal{B}_u$ (*set-unsat I s*) = $\mathcal{B}_u$ *s* $\mathcal{B}_l$ (*set-unsat I s*) = $\mathcal{B}_l$ *s*
by (*auto simp*: *boundsl-def boundsu-def bound-atoms-def*)
have *t-id*: $\mathcal{T}$ (*set-unsat I s*) = $\mathcal{T}$ *s* **by** *simp*
have *u-id*: $\mathcal{U}$ (*set-unsat I s*) = *True* **by** *simp*
let *?p* = *rhs ?eq* − *lp-monom 1* x_i
have *p-eval-zero*: *?p* ⦃ *v* ⦄ = *0* **if** $v \models_t \mathcal{T}$ *s* **for** *v* :: *'a valuation*
proof −
have *eqT*: *?eq* ∈ *set* ($\mathcal{T}$ *s*)
by (*simp add*: *3(7) eq-for-lvar local.min-lvar-not-in-bounds-lvars*)
have $v \models_e$ *?eq* **using** *that eqT satisfies-tableau-def* **by** *blast*
also have *?eq* = (*lhs ?eq*, *rhs ?eq*) **by** (*cases ?eq*, *auto*)
also have *lhs ?eq* = x_i **by** (*simp add*: *3(7) eq-for-lvar local.min-lvar-not-in-bounds-lvars*)
finally have $v \models_e$ (x_i, *rhs ?eq*) **.**
then show *?thesis* **by** (*auto simp*: *satisfies-eq-iff valuate-minus*)
qed
have *Xj-rvars*: $X_j \subseteq$ *rvars* ($\mathcal{T}$ *s*) **unfolding** X_j*-def*
using *3 min-lvar-not-in-bounds-lvars rvars-of-lvar-rvars* **by** *blast*
have *xi-lvars*: $x_i \in$ *lvars* ($\mathcal{T}$ *s*)
using *3 min-lvar-not-in-bounds-lvars rvars-of-lvar-rvars* **by** *blast*
have *lvars* ($\mathcal{T}$ *s*) ∩ *rvars* ($\mathcal{T}$ *s*) = {}
using *3 normalized-tableau-def* **by** *auto*
with *xi-lvars Xj-rvars* **have** *xi-Xj*: $x_i \notin X_j$
by *blast*
have *rhs-eval-xi*: (*rhs* (*eq-for-lvar* ($\mathcal{T}$ *s*) x_i)) ⦃⟨$\mathcal{V}$ *s*⟩⦄ = ⟨$\mathcal{V}$ *s*⟩ x_i
proof −
have ∗: (*rhs eq*) ⦃ *v* ⦄ = *v* (*lhs eq*) **if** $v \models_e$ *eq* **for** *v* :: *'a valuation* **and** *eq*
using *satisfies-eq-def that* **by** *metis*
moreover have ⟨$\mathcal{V}$ *s*⟩ $\models_e$ *eq-for-lvar* ($\mathcal{T}$ *s*) x_i
using *3 satisfies-tableau-def eq-for-lvar curr-val-satisfies-no-lhs-def xi-lvars*
by *blast*
ultimately show *?thesis*
using *eq-for-lvar xi-lvars* **by** *simp*
qed
let *?*$\mathcal{B}_l$ = *Direction.LB dir*
let *?*$\mathcal{B}_u$ = *Direction.UB dir*
let *?lt* = *Direction.lt dir*
let *?le* = *Simplex.le ?lt*
let *?Geq* = *Direction.GE dir*
let *?Leq* = *Direction.LE dir*

have *0*: (*if A x* < *0 then ?*$\mathcal{B}_l$ *s x* = *Some* (⟨$\mathcal{V}$ *s*⟩ *x*) *else ?*$\mathcal{B}_u$ *s x* = *Some* (⟨$\mathcal{V}$ *s*⟩ *x*)) ∧ *A x* ≠ *0*
if *x*: $x \in X_j$ **for** *x*
proof −

have *Some* (⟨$\mathcal{V}$ *s*⟩ *x*) = (*?*$\mathcal{B}_l$ *s x*) **if** *A x* < *0*
proof −
have *cmp*: ¬ $\rhd_{lb}$ *?lt* (⟨$\mathcal{V}$ *s*⟩ *x*) (*?*$\mathcal{B}_l$ *s x*)
using *x that dir min-rvar-incdec-eq-None*[*OF 3*(*9*)] **unfolding** X_j*-def A-def* **by** *auto*
then obtain *c* **where** *c*: *?*$\mathcal{B}_l$ *s x* = *Some c*
by (*cases ?*$\mathcal{B}_l$ *s x*, *auto simp*: *bound-compare-defs*)
also have *c* = ⟨$\mathcal{V}$ *s*⟩ *x*
proof −
have *x* ∈ *rvars* ($\mathcal{T}$ *s*) **using** *that x Xj-rvars* **by** *blast*
then have *x* ∈ (− *lvars* ($\mathcal{T}$ *s*))
using *3* **unfolding** *normalized-tableau-def* **by** *auto*
moreover have ∀ *x*∈(− *lvars* ($\mathcal{T}$ *s*)). *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}_l$ *s*, $\mathcal{B}_u$ *s*)
using *3* **unfolding** *curr-val-satisfies-no-lhs-def*
by (*simp add*: *satisfies-bounds-set.simps*)
ultimately have *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}_l$ *s*, $\mathcal{B}_u$ *s*)
by *blast*
moreover have *?le* (⟨$\mathcal{V}$ *s*⟩ *x*) *c*
using *cmp c dir* **unfolding** *bound-compare-defs* **by** *auto*
ultimately show *?thesis*
using *c dir* **by** (*auto simp del*: *Simplex.bounds-lg*)
qed
then show *?thesis*
using *c* **by** *simp*
qed
moreover have *Some* (⟨$\mathcal{V}$ *s*⟩ *x*) = (*?*$\mathcal{B}_u$ *s x*) **if** *0* < *A x*
proof −
have *cmp*: ¬ $\lhd_{ub}$ *?lt* (⟨$\mathcal{V}$ *s*⟩ *x*) (*?*$\mathcal{B}_u$ *s x*)
using *x that min-rvar-incdec-eq-None*[*OF 3*(*9*)] **unfolding** X_j*-def A-def* **by** *auto*
then obtain *c* **where** *c*: *?*$\mathcal{B}_u$ *s x* = *Some c*
by (*cases ?*$\mathcal{B}_u$ *s x*, *auto simp*: *bound-compare-defs*)
also have *c* = ⟨$\mathcal{V}$ *s*⟩ *x*
proof −
have *x* ∈ *rvars* ($\mathcal{T}$ *s*) **using** *that x Xj-rvars* **by** *blast*
then have *x* ∈ (− *lvars* ($\mathcal{T}$ *s*))
using *3* **unfolding** *normalized-tableau-def* **by** *auto*
moreover have ∀ *x*∈(− *lvars* ($\mathcal{T}$ *s*)). *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}_l$ *s*, $\mathcal{B}_u$ *s*)
using *3* **unfolding** *curr-val-satisfies-no-lhs-def*
by (*simp add*: *satisfies-bounds-set.simps*)
ultimately have *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}_l$ *s*, $\mathcal{B}_u$ *s*)
by *blast*
moreover have *?le c* (⟨$\mathcal{V}$ *s*⟩ *x*)
using *cmp c dir* **unfolding** *bound-compare-defs* **by** *auto*
ultimately show *?thesis*
using *c dir* **by** (*auto simp del*: *Simplex.bounds-lg*)
qed
then show *?thesis*
using *c* **by** *simp*

qed
moreover have $A\ x \neq 0$
using *that coeff-zero* **unfolding** *A-def* X_j*-def* **by** *auto*
ultimately show *?thesis*
using *that* **by** *auto*
qed

have *l-Ba*: $l \in \mathcal{B}_A\ s$ **if** $l \in \{?Geq\ x_i\ (the\ (?\mathcal{B}_l\ s\ x_i))\}$ **for** l
proof −
from *that* **have** *l*: $l = ?Geq\ x_i\ (the\ (?\mathcal{B}_l\ s\ x_i))$ **by** *simp*
from *3*(*8*) **obtain** c **where** *bl′*: $?\mathcal{B}_l\ s\ x_i = Some\ c$
by (*cases* $?\mathcal{B}_l\ s\ x_i$, *auto simp*: *bound-compare-defs*)
hence *bl*: $(x_i,\ c) \in set\text{-}of\text{-}map\ (?\mathcal{B}_l\ s)$ **unfolding** *set-of-map-def* **by** *auto*
show $l \in \mathcal{B}_A\ s$ **unfolding** *l bound-atoms-def* **using** *bl bl′ dir* **by** *auto*
qed

let $?negA = filter\ (\lambda\ x.\ A\ x < 0)\ XL_j$
let $?posA = filter\ (\lambda\ x.\ \neg\ A\ x < 0)\ XL_j$

define *neg* **where** $neg = (if\ dir = Positive\ then\ (\lambda\ x :: rat.\ x)\ else\ uminus)$
define *negP* **where** $negP = (if\ dir = Positive\ then\ (\lambda\ x :: linear\text{-}poly.\ x)\ else\ uminus)$
define *nega* **where** $nega = (if\ dir = Positive\ then\ (\lambda\ x :: 'a.\ x)\ else\ uminus)$
from *dir* **have** *dirn*: $dir = Positive \wedge neg = (\lambda\ x.\ x) \wedge negP = (\lambda\ x.\ x) \wedge nega = (\lambda\ x.\ x)$
$\vee\ dir = Negative \wedge neg = uminus \wedge negP = uminus \wedge nega = uminus$
unfolding *neg-def negP-def nega-def* **by** *auto*

define C **where** $C = map\ (\lambda x.\ ?Geq\ x\ (the\ (?\mathcal{B}_l\ s\ x)))\ ?negA$
$@\ map\ (\lambda\ x.\ ?Leq\ x\ (the\ (?\mathcal{B}_u\ s\ x)))\ \ ?posA$
$@\ [?Geq\ x_i\ (the\ (?\mathcal{B}_l\ s\ x_i))]$
define f **where** $f = (\lambda x.\ if\ x = x_i\ then\ neg\ (-1)\ else\ neg\ (A\ x))$
define c **where** $c = (\sum x{\leftarrow}C.\ lec\text{-}const\ (lec\text{-}of\text{-}nsc\ (f\ (atom\text{-}var\ x)\ *R\ nsc\text{-}of\text{-}atom\ x)))$
let $?q = negP\ ?p$

show *?case* **unfolding** *bounds-id t-id u-id*
proof (*intro exI impI conjI allI*)
show $v \models_t \mathcal{T}\ s \Longrightarrow ?q\ \{\!|\ v\ |\!\}= 0$ **for** $v :: 'a\ valuation$ **using** *dirn p-eval-zero*[*of v*]
by (*auto simp*: *valuate-minus*)

show $set\ C \subseteq \mathcal{B}_A\ s$
unfolding *C-def set-append set-map set-filter list.simps* **using** *0 l-Ba dir*
by (*intro Un-least subsetI*) (*force simp*: *bound-atoms-def set-of-map-def*)+

show *is-leq*: $\forall\,a{\in}set\ C.\ is\text{-}leq\text{-}ns\ (f\ (atom\text{-}var\ a)\ *R\ nsc\text{-}of\text{-}atom\ a) \wedge f\ (atom\text{-}var\ a) \neq 0$
using *dirn xi-Xj 0* **unfolding** *C-def f-def*

by (*elim disjE*, *auto*)

show ($\sum$ *a* ← *C*. *lec-of-nsc* (*f* (*atom-var a*) ∗*R nsc-of-atom a*)) = *Leqc ?q c*
unfolding *sum-list-lec le-constraint.simps map-map o-def*
proof (*intro conjI*)
define *scale-poly* :: *'a atom* ⇒ *linear-poly* **where**
scale-poly = (λ*x*. *lec-poly* (*lec-of-nsc* (*f* (*atom-var x*) ∗*R nsc-of-atom x*)))
have ($\sum$ *x*←*C*. *scale-poly x*) =
($\sum$ *x*←*?negA*. *scale-poly* (*?Geq x* (*the* ($?\mathcal{B}_l$ *s x*))))
\+ ($\sum$ *x*←*?posA*. *scale-poly* (*?Leq x* (*the* ($?\mathcal{B}_u$ *s x*))))
− *negP* (*lp-monom 1* x_i)
unfolding *C-def* **using** *dirn* **by** (*auto simp add*: *comp-def scale-poly-def f-def*)
also have ($\sum$ *x*←*?negA*. *scale-poly* (*?Geq x* (*the* ($?\mathcal{B}_l$ *s x*))))
= ($\sum$ *x*← *?negA*. *negP* (*A x* ∗*R lp-monom 1 x*))
unfolding *scale-poly-def f-def* **using** *dirn xi-Xj* **by** (*subst map-cong*) *auto*
also have ($\sum$ *x*←*?posA*. *scale-poly* (*?Leq x* (*the* ($?\mathcal{B}_u$ *s x*))))
= ($\sum$ *x*← *?posA*. *negP* (*A x* ∗*R lp-monom 1 x*))
unfolding *scale-poly-def f-def* **using** *dirn xi-Xj* **by** (*subst map-cong*) *auto*
also have ($\sum$ *x*← *?negA*. *negP* (*A x* ∗*R lp-monom 1 x*)) +
($\sum$ *x*← *?posA*. *negP* (*A x* ∗*R lp-monom 1 x*))
= *negP* (*rhs* (*eq-for-lvar* ($\mathcal{T}$ *s*) x_i))
using *dirn* XL_j*-distinct coeff-zero*
by (*elim disjE*; *intro poly-eqI*, *auto intro*!: *poly-eqI simp add*: *coeff-sum-list A-def* X_j*-def*
uminus-sum-list-map[*unfolded o-def*, *symmetric*])
finally show ($\sum$ *x*←*C*. *lec-poly* (*lec-of-nsc* (*f* (*atom-var x*) ∗*R nsc-of-atom x*))) = *?q*
unfolding *scale-poly-def* **using** *dirn* **by** *auto*
show ($\sum$ *x*←*C*. *lec-rel* (*lec-of-nsc* (*f* (*atom-var x*) ∗*R nsc-of-atom x*))) = *Leq-Rel*
unfolding *sum-list-Leq-Rel*
proof
fix *c*
assume *c*: *c* ∈ *set C*
show *lec-rel* (*lec-of-nsc* (*f* (*atom-var c*) ∗*R nsc-of-atom c*)) = *Leq-Rel*
using *is-leq*[*rule-format*, *OF c*] **by** (*cases f* (*atom-var c*) ∗*R nsc-of-atom c*, *auto*)
qed
qed (*simp add*: *c-def*)

show *c* < *0*
proof −
define *scale-const-f* :: *'a atom* ⇒ *'a* **where**
scale-const-f x = *lec-const* (*lec-of-nsc* (*f* (*atom-var x*) ∗*R nsc-of-atom x*)) **for** *x*
obtain *d* **where** *bl'*: $?\mathcal{B}_l$ *s* x_i = *Some d*
using *3* **by** (*cases* $?\mathcal{B}_l$ *s* x_i, *auto simp*: *bound-compare-defs*)
have *c* = ($\sum$ *x*←*map* (λ*x*. *?Geq x* (*the* ($?\mathcal{B}_l$ *s x*))) *?negA*. *scale-const-f x*)

+ ($\sum$ x← *map* (λx. *?Leq x* (*the* (*?*$\mathcal{B}_u$ *s x*))) *?posA*. *scale-const-f x*)
− *nega d*
unfolding *c-def C-def f-def scale-const-f-def* **using** *dirn rhs-eval-xi bl′* **by** *auto*
also have ($\sum$ x←*map* (λx. *?Geq x* (*the* (*?*$\mathcal{B}_l$ *s x*))) *?negA*. *scale-const-f x*) =
($\sum$ x← *?negA*. *nega* (*A x* ∗*R the* (*?*$\mathcal{B}_l$ *s x*)))
using *xi-Xj dirn* **by** (*subst map-cong*) (*auto simp add*: *f-def scale-const-f-def*)
also have . . . = ($\sum$ x←*?negA*. *nega* (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*))
using *0* **by** (*subst map-cong*) *auto*
also have ($\sum$ x←*map* (λx. *?Leq x* (*the* (*?*$\mathcal{B}_u$ *s x*))) *?posA*. *scale-const-f x*) =
($\sum$ x← *?posA*. *nega* (*A x* ∗*R the* (*?*$\mathcal{B}_u$ *s x*)))
using *xi-Xj dirn* **by** (*subst map-cong*) (*auto simp add*: *f-def scale-const-f-def*)
also have . . . = ($\sum$ x← *?posA*. *nega* (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*))
using *0* **by** (*subst map-cong*) *auto*
also have ($\sum$ x←*?negA*. *nega* (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*)) + ($\sum$ x←*?posA*. *nega* (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*))
= ($\sum$ x←*?negA* @ *?posA*. *nega* (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*))
by *auto*
also have . . . = ($\sum$ x∈ X_j. *nega* (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*))
using XL_j*-distinct* **by** (*subst sum-list-distinct-conv-sum-set*) (*auto intro*!: *sum.cong*)
also have . . . = *nega* ($\sum$ x∈ X_j. (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*)) **using** *dirn* **by** (*auto simp*: *sum-negf*)
also have ($\sum$ x∈ X_j. (*A x* ∗*R* ⟨$\mathcal{V}$ *s*⟩ *x*)) = ((*rhs ?eq*) ⦃⟨$\mathcal{V}$ *s*⟩⦄)
unfolding *A-def* X_j*-def* **by** (*subst linear-poly-sum*) (*auto simp add*: *sum-negf*)
also have . . . = ⟨$\mathcal{V}$ *s*⟩ x_i
using *rhs-eval-xi* **by** *blast*
also have *nega* (⟨$\mathcal{V}$ *s*⟩ x_i) − *nega d* < *0*
proof −
have *?lt* (⟨$\mathcal{V}$ *s*⟩ x_i) *d*
using *dirn 3*(*2*−) *bl′* **by** (*elim disjE*, *auto simp*: *bound-compare-defs*)
thus *?thesis* **using** *dirn* **unfolding** *minus-lt*[*symmetric*] **by** *auto*
qed
finally show *?thesis* **.**
qed

show *distinct C*
unfolding *C-def* **using** XL_j*-distinct xi-Xj dirn* **by** (*auto simp add*: *inj-on-def distinct-map*)
qed
qed (*insert U*, *blast*+)
then obtain *f p c C* **where** *Qs*: *?Q s f p c C* **using** *U* **unfolding** *check* **by** *blast*
from *index*[*folded check-tableau-index-valid*[*OF U*(*2*) *inv*(*3*,*4*,*2*,*1*)]] *check*
have *index*: *index-valid as s* **by** *auto*

```
  from check-tableau-equiv[OF U(2) inv(3,4,2,1), unfolded check]
  have id: v ⊨t 𝒯 s = v ⊨t 𝒯 s′ for v :: ′a valuation by auto
  let ?C = map (λ a. (f (atom-var a), a)) C
  have set C ⊆ ℬA s using Qs by blast
  also have ... ⊆ snd ‘ as using index
   unfolding bound-atoms-def index-valid-def set-of-map-def boundsl-def boundsu-def
o-def by force
  finally have sub: snd ‘ set ?C ⊆ snd ‘ as by force
  show ?thesis unfolding farkas-coefficients-atoms-tableau-def
    by (intro exI[of - p] exI[of - c] exI[of - ?C] conjI,
        insert Qs[unfolded id] sub, (force simp: o-def)+)
qed

end
```

Next, we show that a conflict found by the assert-bound function also gives rise to Farkas coefficients.

```
context Update
begin

lemma farkas-assert-bound: assumes inv: ¬ 𝒰 s ⊨nolhs s △ (𝒯 s) ∇ s ◊ s
  and index: index-valid as s
  and U: 𝒰 (assert-bound ia s)
shows ∃ C. farkas-coefficients-atoms-tableau (snd ‘ (insert ia as)) (𝒯 s) C
proof –
  obtain i a where ia[simp]: ia = (i,a) by force
  let ?A = snd ‘ insert ia as
  have ∃ x c d. Leq x c ∈ ?A ∧ Geq x d ∈ ?A ∧ c < d
  proof (cases a)
    case (Geq x d)
    let ?s = updateℬℐ (Direction.UBI-upd (Direction (λx y. y < x) ℬiu ℬil ℬu ℬl
ℐu ℐl ℬil-update Geq Leq (≤)))
                        i x d s
    have id: 𝒰 ?s = 𝒰 s by auto
    have norm: △ (𝒯 ?s) using inv by auto
    have val: ∇ ?s using inv(4) unfolding tableau-valuated-def by simp
    have idd: x ∉ lvars (𝒯 ?s) ⟹ 𝒰 (update x d ?s) = 𝒰 ?s
      by (rule update-unsat-id[OF norm val])
    from U[unfolded ia Geq] inv(1) id idd
    have ◁lb (λx y. y < x) d (ℬu s x) by (auto split: if-splits simp: Let-def)
    then obtain c where Bu: ℬu s x = Some c and lt: c < d
      by (cases ℬu s x, auto simp: bound-compare-defs)
    from Bu obtain j where Mapping.lookup (ℬiu s) x = Some (j,c)
      unfolding boundsu-def by auto
    with index[unfolded index-valid-def] have (j, Leq x c) ∈ as by auto
    hence xc: Leq x c ∈ ?A by force
    have xd: Geq x d ∈ ?A unfolding ia Geq by force
    from xc xd lt show ?thesis by auto
  next
```

case (*Leq x c*)
let *?s* = *updateℬℐ* (*Direction.UBI-upd* (*Direction* (<) $\mathcal{B}_{il}$ $\mathcal{B}_{iu}$ $\mathcal{B}_l$ $\mathcal{B}_u$ $\mathcal{I}_l$ $\mathcal{I}_u$ $\mathcal{B}_{iu}$*-update Leq Geq* (≥))) *i x c s*
have *id*: $\mathcal{U}$ *?s* = $\mathcal{U}$ *s* **by** *auto*
have *norm*: △ ($\mathcal{T}$ *?s*) **using** *inv* **by** *auto*
have *val*: ∇ *?s* **using** *inv*(*4*) **unfolding** *tableau-valuated-def* **by** *simp*
have *idd*: *x* ∉ *lvars* ($\mathcal{T}$ *?s*) ⟹ $\mathcal{U}$ (*update x c ?s*) = $\mathcal{U}$ *?s*
by (*rule update-unsat-id*[*OF norm val*])
from *U*[*unfolded ia Leq*] *inv*(*1*) *id idd*
have $\lhd_{lb}$ (<) *c* ($\mathcal{B}_l$ *s x*) **by** (*auto split*: *if-splits simp*: *Let-def*)
then obtain *d* **where** *Bl*: $\mathcal{B}_l$ *s x* = *Some d* **and** *lt*: *c* < *d*
by (*cases* $\mathcal{B}_l$ *s x*, *auto simp*: *bound-compare-defs*)
from *Bl* **obtain** *j* **where** *Mapping.lookup* ($\mathcal{B}_{il}$ *s*) *x* = *Some* (*j*,*d*)
unfolding *boundsl-def* **by** *auto*
with *index*[*unfolded index-valid-def*] **have** (*j*, *Geq x d*) ∈ *as* **by** *auto*
hence *xd*: *Geq x d* ∈ *?A* **by** *force*
have *xc*: *Leq x c* ∈ *?A* **unfolding** *ia Leq* **by** *force*
from *xc xd lt* **show** *?thesis* **by** *auto*
qed
then obtain *x c d* **where** *c*: *Leq x c* ∈ *?A* **and** *d*: *Geq x d* ∈ *?A* **and** *cd*: *c* < *d* **by** *blast*
show *?thesis* **unfolding** *farkas-coefficients-atoms-tableau-def*
proof (*intro exI conjI allI*)
let *?C* = [(−*1*, *Geq x d*), (*1*,*Leq x c*)]
show ∀ (*r*,*a*)∈*set ?C*. *a* ∈ *?A* ∧ *is-leq-ns* (*r* ∗*R nsc-of-atom a*) ∧ *r* ≠ *0* **using** *c d* **by** *auto*
show *c* − *d* < *0* **using** *cd* **using** *minus-lt* **by** *auto*
qed (*auto simp*: *valuate-zero*)
qed
end

Moreover, we prove that all other steps of the simplex algorithm on layer 4, such as pivoting, asserting bounds without conflict, etc., preserve Farkas coefficients.

lemma *farkas-coefficients-atoms-tableau-mono*: **assumes** *as* ⊆ *bs*
shows *farkas-coefficients-atoms-tableau as t C* ⟹ *farkas-coefficients-atoms-tableau bs t C*
using *assms* **unfolding** *farkas-coefficients-atoms-tableau-def* **by** *force*

locale *AssertAllState‴* = *AssertAllState″ init ass-bnd chk* + *Update update* + *PivotUpdateMinVars eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq pivot-and-update*
for *init* **and** *ass-bnd* :: ′*i* × ′*a* :: *lrv atom* ⇒ - **and** *chk* :: (′*i*, ′*a*) *state* ⇒ (′*i*, ′*a*) *state* **and** *update* :: *nat* ⇒ ′*a* :: *lrv* ⇒ (′*i*, ′*a*) *state* ⇒ (′*i*, ′*a*) *state*
and *eq-idx-for-lvar* :: *tableau* ⇒ *var* ⇒ *nat* **and**
min-lvar-not-in-bounds :: (′*i*,′*a*::*lrv*) *state* ⇒ *var option* **and**
min-rvar-incdec-eq :: (′*i*,′*a*) *Direction* ⇒ (′*i*,′*a*) *state* ⇒ *eq* ⇒ ′*i list* + *var* **and**
pivot-and-update :: *var* ⇒ *var* ⇒ ′*a* ⇒ (′*i*,′*a*) *state* ⇒ (′*i*,′*a*) *state*
+ **assumes** *ass-bnd*: *ass-bnd* = *Update.assert-bound update* **and**

chk: *chk* = *PivotUpdateMinVars.check eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq pivot-and-update*

context *AssertAllState'''*
begin

lemma *farkas-assert-bound-loop*: **assumes** $\mathcal{U}$ (*assert-bound-loop as* (*init t*))
 and *norm*: △ *t*
shows ∃ *C*. *farkas-coefficients-atoms-tableau* (*snd* ‘ *set as*) *t C*
proof −
 let *?P* = λ *as s*. $\mathcal{U}$ *s* ⟶ (∃ *C*. *farkas-coefficients-atoms-tableau* (*snd* ‘ *as*) ($\mathcal{T}$ *s*) *C*)
 let *?s* = *assert-bound-loop as* (*init t*)
 have ¬ $\mathcal{U}$ (*init t*) **by** (*rule init-unsat-flag*)
 have $\mathcal{T}$ (*assert-bound-loop as* (*init t*)) = *t* ∧
  ($\mathcal{U}$ (*assert-bound-loop as* (*init t*)) ⟶ (∃ *C*. *farkas-coefficients-atoms-tableau* (*snd* ‘ *set as*) ($\mathcal{T}$ (*init t*)) *C*))
 proof (*rule AssertAllState''Induct*[*OF norm*], *unfold ass-bnd*, *goal-cases*)
  case *1*
  have ¬ $\mathcal{U}$ (*init t*) **by** (*rule init-unsat-flag*)
  moreover have $\mathcal{T}$ (*init t*) = *t* **by** (*rule init-tableau-id*)
  ultimately show *?case* **by** *auto*
 next
  case (*2 as bs s*)
  hence *snd* ‘ *as* ⊆ *snd* ‘ *bs* **by** *auto*
  from *farkas-coefficients-atoms-tableau-mono*[*OF this*] *2*(*2*) **show** *?case* **by** *auto*
 next
  case (*3 s a ats*)
  let *?s* = *assert-bound a s*
  have *tab*: $\mathcal{T}$ *?s* = $\mathcal{T}$ *s* **unfolding** *ass-bnd* **by** (*rule assert-bound-nolhs-tableau-id*, *insert 3*, *auto*)
  have *t*: *t* = $\mathcal{T}$ *s* **using** *3* **by** *simp*
  show *?case* **unfolding** *t tab*
  proof (*intro conjI impI refl*)
   assume $\mathcal{U}$ *?s*
   from *farkas-assert-bound*[*OF 3*(*1*,*3*−*6*,*8*) *this*]
   show ∃ *C*. *farkas-coefficients-atoms-tableau* (*snd* ‘ *insert a* (*set ats*)) ($\mathcal{T}$ (*init* ($\mathcal{T}$ *s*))) *C*
    unfolding *t*[*symmetric*] *init-tableau-id* **.**
  qed
 qed
 thus *?thesis* **unfolding** *init-tableau-id* **using** *assms* **by** *blast*
qed

Now we get to the main result for layer 4: If the main algorithm returns unsat, then there are Farkas coefficients for the tableau and atom set that were given as input for this layer.

lemma *farkas-assert-all-state*: **assumes** *U*: $\mathcal{U}$ (*assert-all-state t as*)
 and *norm*: △ *t*

shows ∃ *C*. *farkas-coefficients-atoms-tableau* (*snd* ‘ *set as*) *t C*
proof −
  let *?s* = *assert-bound-loop as* (*init t*)
  show *?thesis*
  proof (*cases* 𝒰 (*assert-bound-loop as* (*init t*)))
    case *True*
    from *farkas-assert-bound-loop*[*OF this norm*]
    show *?thesis* **by** *auto*
  next
    case *False*
    from *AssertAllState′′-tableau-id*[*OF norm*]
    have *T*: 𝒯 *?s* = *t* **unfolding** *init-tableau-id* **.**
    from *U* **have** *U*: 𝒰 (*check ?s*) **unfolding** *chk*[*symmetric*] **by** *simp*
    show *?thesis*
    proof (*rule farkas-check*[*OF refl U False*, *unfolded T*, *OF - norm*])
      from *AssertAllState′′-precond*[*OF norm*, *unfolded Let-def*] *False*
      show $\models_{nolhs}$ *?s* ◊ *?s* ∇ *?s* **by** *blast*+
      from *AssertAllState′′-index-valid*[*OF norm*]
      show *index-valid* (*set as*) *?s* **.**
    qed
  qed
qed

2.3 Farkas' Lemma on Layer 3

There is only a small difference between layers 3 and 4, namely that there is no simplex algorithm (*assert-all-state*) on layer 3, but just a tableau and atoms.

Hence, one task is to link the unsatisfiability flag on layer 4 with unsatisfiability of the original tableau and atoms (layer 3). This can be done via the existing soundness results of the simplex algorithm. Moreover, we give an easy proof that the existence of Farkas coefficients for a tableau and set of atoms implies unsatisfiability.

end

lemma *farkas-coefficients-atoms-tableau-unsat*:
  assumes *farkas-coefficients-atoms-tableau as t C*
  shows ∄ *v*. *v* $\models_t$ *t* ∧ *v* $\models_{as}$ *as*
proof
  assume ∃ *v*. *v* $\models_t$ *t* ∧ *v* $\models_{as}$ *as*
  then obtain *v* **where** ∗: *v* $\models_t$ *t* ∧ *v* $\models_{as}$ *as* **by** *auto*
  then obtain *p c* **where** *isleq*: (∀ (*r*,*a*) ∈ *set C*. *a* ∈ *as* ∧ *is-leq-ns* (*r* ∗*R* *nsc-of-atom a*) ∧ *r* ≠ *0*)
    and *leq*: ($\sum$ (*r*,*a*) ← *C*. *lec-of-nsc* (*r* ∗*R* *nsc-of-atom a*)) = *Leqc p c*
    and *cltz*: *c* < *0*
    and *p0*: *p*⦃*v*⦄ = *0*
    using *assms farkas-coefficients-atoms-tableau-def* **by** *blast*
  have *fa*: ∀ (*r*,*a*) ∈ *set C*. *v* $\models_a$ *a* **using** ∗ *isleq leq*

satisfies-atom-set-def **by** *force*
{
fix *r a*
assume *a*: $(r,a) \in set\ C$
from *a fa* **have** *va*: $v \models_a a$ **unfolding** *satisfies-atom-set-def* **by** *auto*
hence *v*: $v \models_{ns} (r *R\ nsc\text{-}of\text{-}atom\ a)$ **by** (*auto simp*: *nsc-of-atom sat-scale-rat-ns*)
from *a isleq* **have** *is-leq-ns* (*r ∗R nsc-of-atom a*) **by** *auto*
from *lec-of-nsc*[*OF this*] *v* **have** $v \models_{le} lec\text{-}of\text{-}nsc\ (r *R\ nsc\text{-}of\text{-}atom\ a)$ **by** *blast*
} **note** *v* = *this*
have $v \models_{le} Leqc\ p\ c$ **unfolding** *leq*[*symmetric*]
by (*rule satisfies-sumlist-le-constraints*, *insert v*, *auto*)
then have $0 \le c$ **using** *p0* **by** *auto*
then show *False* **using** *cltz* **by** *auto*
qed

Next is the main result for layer 3: a tableau and a finite set of atoms are unsatisfiable if and only if there is a list of Farkas coefficients for the set of atoms and the tableau.

lemma *farkas-coefficients-atoms-tableau*: **assumes** *norm*: △ *t*
and *fin*: *finite as*
shows (∃ *C*. *farkas-coefficients-atoms-tableau as t C*) ⟷ $(\nexists\ v.\ v \models_t t \land v \models_{as} as)$
proof
from *finite-list*[*OF fin*] **obtain** *bs* **where** *as*: *as* = *set bs* **by** *auto*
assume *unsat*: $\nexists\ v.\ v \models_t t \land v \models_{as} as$
let *?as* = *map* (λ *x*. ((),*x*)) *bs*
interpret *AssertAllState′′′ init-state assert-bound-code check-code update-code eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq pivot-and-update-code*
by (*unfold-locales*, *auto simp*: *assert-bound-code-def check-code-def*)
let *?call* = *assert-all t ?as*
have *id*: *snd ‘ set ?as* = *as* **unfolding** *as* **by** *force*
from *assert-all-sat*[*OF norm*, *of ?as*, *unfolded id*] *unsat*
obtain *I* **where** *?call* = *Inl I* **by** (*cases ?call*, *auto*)
from *this*[*unfolded assert-all-def Let-def*]
have $\mathcal{U}$ (*assert-all-state-code t ?as*)
by (*auto split*: *if-splits simp*: *assert-all-state-code-def*)
from *farkas-assert-all-state*[*OF this*[*unfolded assert-all-state-code-def*] *norm*, *unfolded id*]
show ∃ *C*. *farkas-coefficients-atoms-tableau as t C* **.**
qed (*insert farkas-coefficients-atoms-tableau-unsat*, *auto*)

2.4 Farkas' Lemma on Layer 2

The main difference between layers 2 and 3 is the introduction of slack-variables in layer 3 via the preprocess-function. Our task here is to show that Farkas coefficients at layer 3 (where slack-variables are used) can be converted into Farkas coefficients for layer 2 (before the preprocessing).

We also need to adapt the previos notion of Farkas coefficients, which was used in *farkas-coefficients-atoms-tableau*, for layer 2. At layer 3, Farkas coefficients are the coefficients in a linear combination of atoms that evaluates to an inequality of the form $p \leq c$, where p is a linear polynomial, $c < 0$, and $t \models p = 0$ holds. At layer 2, the atoms are replaced by non-strict constraints where the left-hand side is a polynomial in the original variables, but the corresponding linear combination (with Farkas coefficients) evaluates directly to the inequality $0 \leq c$, with $c < 0$. The implication $t \models p = 0$ is no longer possible in this layer, since there is no tableau t, nor is it needed, since p is 0. Thus, the statement defining Farkas coefficients must be changed accordingly.

definition *farkas-coefficients-ns* **where**
  farkas-coefficients-ns ns C = (∃ *c*.
    (∀ (*r*, *n*) ∈ *set C*. *n* ∈ *ns* ∧ *is-leq-ns* (*r* ∗*R n*) ∧ *r* ≠ *0*) ∧
    (∑ (*r*, *n*) ← *C*. *lec-of-nsc* (*r* ∗*R n*)) = *Leqc 0 c* ∧
    c < *0*)

The easy part is to prove that Farkas coefficients imply unsatisfiability.

lemma *farkas-coefficients-ns-unsat*:
  assumes *farkas-coefficients-ns ns C*
  shows ∄ *v*. *v* $\models_{nss}$ *ns*
proof
  assume ∃ *v*. *v* $\models_{nss}$ *ns*
  then obtain *v* **where** ∗: *v* $\models_{nss}$ *ns* **by** *auto*
  obtain *c* **where**
    isleq: (∀ (*a*,*n*) ∈ *set C*. *n* ∈ *ns* ∧ *is-leq-ns* (*a* ∗*R n*) ∧ *a* ≠ *0*) **and**
    leq: (∑ (*a*,*n*) ← *C*. *lec-of-nsc* (*a* ∗*R n*)) = *Leqc 0 c* **and**
    cltz: *c* < *0* **using** *assms farkas-coefficients-ns-def* **by** *blast*
  {
    fix *a n*
    assume *n*: (*a*,*n*) ∈ *set C*
    from *n* ∗ *isleq* **have** *v* $\models_{ns}$ *n* **by** *auto*
    hence *v*: *v* $\models_{ns}$ (*a* ∗*R n*) **by** (*rule sat-scale-rat-ns*)
    from *n isleq* **have** *is-leq-ns* (*a* ∗*R n*) **by** *auto*
    from *lec-of-nsc*[*OF this*] *v*
    have *v* $\models_{le}$ *lec-of-nsc* (*a* ∗*R n*) **by** *blast*
  } **note** *v* = *this*
  have *v* $\models_{le}$ *Leqc 0 c* **unfolding** *leq*[*symmetric*]
    by (*rule satisfies-sumlist-le-constraints*, *insert v*, *auto*)
  then show *False* **using** *cltz*
    by (*metis leD satisfiable-le-constraint.simps valuate-zero rel-of.simps*(*1*))
qed

In order to eliminate the need for a tableau, we require the notion of an arbitrary substitution on polynomials, where all variables can be replaced at once. The existing simplex formalization provides only a function to replace one variable at a time.

definition *subst-poly* :: (*var* ⇒ *linear-poly*) ⇒ *linear-poly* ⇒ *linear-poly* **where**

subst-poly σ *p* = ($\sum$ *x* $\in$ *vars p*. *coeff p x* $*R$ σ *x*)

lemma *subst-poly-0*[*simp*]: *subst-poly* σ *0* = *0* **unfolding** *subst-poly-def* **by** *simp*

lemma *valuate-subst-poly*: (*subst-poly* σ *p*) $\{\!|$ *v* $|\!\}$ = (*p* $\{\!|$ (λ *x*. ((σ *x*) $\{\!|$ *v* $|\!\}$)) $|\!\}$)
by (*subst* (*2*) *linear-poly-sum*, *unfold subst-poly-def valuate-sum valuate-scaleRat*, *simp*)

lemma *subst-poly-add*: *subst-poly* σ (*p* + *q*) = *subst-poly* σ *p* + *subst-poly* σ *q*
by (*rule linear-poly-eqI*, *unfold valuate-add valuate-subst-poly*, *simp*)

fun *subst-poly-lec* :: (*var* $\Rightarrow$ *linear-poly*) $\Rightarrow$ *'a le-constraint* $\Rightarrow$ *'a le-constraint*
where
subst-poly-lec σ (*Le-Constraint rel p c*) = *Le-Constraint rel* (*subst-poly* σ *p*) *c*

lemma *subst-poly-lec-0*[*simp*]: *subst-poly-lec* σ *0* = *0* **unfolding** *zero-le-constraint-def*
by *simp*

lemma *subst-poly-lec-add*: *subst-poly-lec* σ (*c1* + *c2*) = *subst-poly-lec* σ *c1* + *subst-poly-lec* σ *c2*
by (*cases c1*; *cases c2*, *auto simp*: *subst-poly-add*)

lemma *subst-poly-lec-sum-list*: *subst-poly-lec* σ (*sum-list ps*) = *sum-list* (*map* (*subst-poly-lec* σ) *ps*)
by (*induct ps*, *auto simp*: *subst-poly-lec-add*)

lemma *subst-poly-lp-monom*[*simp*]: *subst-poly* σ (*lp-monom r x*) = *r* $*R$ σ *x*
unfolding *subst-poly-def* **by** (*simp add*: *vars-lp-monom*)

lemma *subst-poly-scaleRat*: *subst-poly* σ (*r* $*R$ *p*) = *r* $*R$ (*subst-poly* σ *p*)
by (*rule linear-poly-eqI*, *unfold valuate-scaleRat valuate-subst-poly*, *simp*)

We need several auxiliary properties of the preprocess-function which are not present in the simplex formalization.

lemma *Tableau-is-monom-preprocess'*:
assumes (*x*, *p*) $\in$ *set* (*Tableau* (*preprocess' cs start*))
shows $\neg$ *is-monom p*
using *assms*
by(*induction cs start rule*: *preprocess'.induct*)
(*auto simp add*: *Let-def split*: *if-splits option.splits*)

lemma *preprocess'-atoms-to-constraints'*: **assumes** *preprocess' cs start* = *S*
shows *set* (*Atoms S*) $\subseteq$ {(*i*,*qdelta-constraint-to-atom c v*) | *i c v*. (*i*,*c*) $\in$ *set cs* $\wedge$
($\neg$ *is-monom* (*poly c*) $\longrightarrow$ *Poly-Mapping S* (*poly c*) = *Some v*)}
unfolding *assms*(*1*)[*symmetric*]
by (*induct cs start rule*: *preprocess'.induct*, *auto simp*: *Let-def split*: *option.splits*, *force*+)

lemma *monom-of-atom-coeff*:
assumes *is-monom* (*poly ns*) *a* = *qdelta-constraint-to-atom ns v*
shows (*monom-coeff* (*poly ns*)) *∗R nsc-of-atom a* = *ns*
using *assms is-monom-monom-coeff-not-zero*
by(*cases a*; *cases ns*)
(*auto split*: *atom.split ns-constraint.split simp add*: *monom-poly-assemble field-simps*)

The next lemma provides the functionality that is required to convert an atom back to a non-strict constraint, i.e., it is a kind of inverse of the preprocess-function.

lemma *preprocess′-atoms-to-constraints*: **assumes** *S*: *preprocess′ cs start* = *S*
and *start*: *start* = *start-fresh-variable cs*
and *ns*: *ns* = (*case a of Leq v c* ⇒ *LEQ-ns q c* | *Geq v c* ⇒ *GEQ-ns q c*)
and *a* ∈ *snd ‘ set* (*Atoms S*)
shows (*atom-var a* ∉ *fst ‘ set* (*Tableau S*) ⟶ (∃ *r*. *r* ≠ *0* ∧ *r ∗R nsc-of-atom a* ∈ *snd ‘ set cs*))
∧ ((*atom-var a*, *q*) ∈ *set* (*Tableau S*) ⟶ *ns* ∈ *snd ‘ set cs*)
proof −
let *?S* = *preprocess′ cs start*
from *assms*(*4*) **obtain** *i* **where** *ia*: (*i*,*a*) ∈ *set* (*Atoms S*) **by** *auto*
with *preprocess′-atoms-to-constraints′*[*OF assms*(*1*)] **obtain** *c v*
where *a*: *a* = *qdelta-constraint-to-atom c v* **and** *c*: (*i*,*c*) ∈ *set cs*
and *nmonom*: ¬ *is-monom* (*poly c*) ⟹ *Poly-Mapping S* (*poly c*) = *Some v*
by *blast*
hence *c′*: *c* ∈ *snd ‘ set cs* **by** *force*
let *?p* = *poly c*
show *?thesis*
proof (*cases is-monom ?p*)
case *True*
hence *av*: *atom-var a* = *monom-var ?p* **unfolding** *a* **by** (*cases c*, *auto*)
from *Tableau-is-monom-preprocess′*[*of - ?p cs start*] *True*
have (*x*, *?p*) ∉ *set* (*Tableau ?S*) **for** *x* **by** *blast*
{
assume (*atom-var a*, *q*) ∈ *set* (*Tableau S*)
hence (*monom-var ?p*, *q*) ∈ *set* (*Tableau S*) **unfolding** *av* **by** *simp*
hence *monom-var ?p* ∈ *lvars* (*Tableau S*) **unfolding** *lvars-def* **by** *force*
from *lvars-tableau-ge-start*[*rule-format*, *OF this*[*folded S*]]
have *monom-var ?p* ≥ *start* **unfolding** *S* **.**
moreover have *monom-var ?p* ∈ *vars-constraints* (*map snd cs*) **using** *True c*
by (*auto intro*!: *bexI*[*of -* (*i*,*c*)] *simp*: *monom-var-in-vars*)
ultimately have *False* **using** *start-fresh-variable-fresh*[*of cs*, *folded start*] **by** *force*
}
moreover
from *monom-of-atom-coeff*[*OF True a*] *True*
have ∃ *r*. *r* ≠ *0* ∧ *r ∗R nsc-of-atom a* = *c*
by (*intro exI*[*of - monom-coeff ?p*], *auto*, *cases a*, *auto*)
ultimately show *?thesis* **using** *c′* **by** *auto*

```
  next
    case False
    hence av: atom-var a = v unfolding a by (cases c, auto)
    from nmonom[OF False] have Poly-Mapping S ?p = Some v .
    from preprocess'-Tableau-Poly-Mapping-Some[OF this[folded S]]
    have tab: (atom-var a, ?p) ∈ set (Tableau (preprocess' cs start)) unfolding av
by simp
    hence atom-var a ∈ fst ‘ set (Tableau S) unfolding S by force
    moreover
    {
      assume (atom-var a, q) ∈ set (Tableau S)
      from tab this have qp: q = ?p unfolding S using lvars-distinct[of cs start,
unfolded S lhs-def]
        by (simp add: case-prod-beta' eq-key-imp-eq-value)
      have ns = c unfolding ns qp using av a False by (cases c, auto)
      hence ns ∈ snd ‘ set cs using c' by blast
    }
    ultimately show ?thesis by blast
  qed
qed
```

Next follows the major technical lemma of this part, namely that Farkas coefficients on layer 3 for preprocessed constraints can be converted into Farkas coefficients on layer 2.

```
lemma farkas-coefficients-preprocess':
  assumes pp: preprocess' cs (start-fresh-variable cs) = S and
    ft: farkas-coefficients-atoms-tableau (snd ‘ set (Atoms S)) (Tableau S) C
  shows ∃ C. farkas-coefficients-ns (snd ‘ set cs) C
proof −
  note ft[unfolded farkas-coefficients-atoms-tableau-def]
  obtain p c where 0: ∀ (r,a) ∈ set C. a ∈ snd ‘ set (Atoms S) ∧ is-leq-ns (r ∗R
nsc-of-atom a) ∧ r ≠ 0
    (∑(r,a)←C. lec-of-nsc (r ∗R nsc-of-atom a)) = Leqc p c
    c < 0
    ⋀v :: QDelta valuation. v ⊨t Tableau S ⟹ p ⦃ v ⦄ = 0
    using ft unfolding farkas-coefficients-atoms-tableau-def by blast
  note 0 = 0(1)[rule-format, of (a, b) for a b, unfolded split] 0(2−)
  let ?T = Tableau S
  define σ :: var ⇒ linear-poly where σ = (λ x. case map-of ?T x of Some p ⇒
p | None ⇒ lp-monom 1 x)
  let ?P = (λr a s ns. ns ∈ (snd ‘ set cs) ∧ is-leq-ns (s ∗R ns) ∧ s ≠ 0 ∧
      subst-poly-lec σ (lec-of-nsc (r ∗R nsc-of-atom a)) = lec-of-nsc (s ∗R ns))
  have ∃s ns. ?P r a s ns if ra: (r,a) ∈ set C for r a
  proof −
    have a: a ∈ snd ‘ set (Atoms S)
      using ra 0 by force
    from 0 ra have is-leq: is-leq-ns (r ∗R nsc-of-atom a) and r0: r ≠ 0 by auto
    let ?x = atom-var a
    show ?thesis
```

proof (*cases map-of ?T ?x*)
case (*Some q*)
hence *σ*: *σ ?x = q* **unfolding** *σ-def* **by** *auto*
from *Some* **have** *xqT*: (*?x*, *q*) ∈ *set ?T* **by** (*rule map-of-SomeD*)
define *ns* **where** *ns* = (*case a of Leq v c* ⇒ *LEQ-ns q c*
| *Geq v c* ⇒ *GEQ-ns q c*)
from *preprocess′-atoms-to-constraints*[*OF pp refl ns-def a*] *xqT*
have *ns-mem*: *ns* ∈ *snd ‘ set cs* **by** *blast*
have *id*: *subst-poly-lec σ* (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*))
= *lec-of-nsc* (*r* ∗*R ns*) **using** *is-leq σ*
by (*cases a*, *auto simp*: *ns-def subst-poly-scaleRat*)
from *id is-leq σ* **have** *is-leq*: *is-leq-ns* (*r* ∗*R ns*) **by** (*cases a*, *auto simp*: *ns-def*)
show *?thesis* **by** (*intro exI*[*of - r*] *exI*[*of - ns*] *conjI ns-mem id is-leq conjI r0*)
next
case *None*
hence *?x* ∉ *fst ‘ set ?T* **by** (*meson map-of-eq-None-iff*)
from *preprocess′-atoms-to-constraints*[*OF pp refl refl a*] *this*
obtain *rr* **where** *rr*: *rr* ∗*R nsc-of-atom a* ∈ (*snd ‘ set cs*) **and** *rr0*: *rr* ≠ *0*
by *blast*
from *None* **have** *σ*: *σ ?x = lp-monom 1 ?x* **unfolding** *σ-def* **by** *simp*
define *ns* **where** *ns* = *rr* ∗*R nsc-of-atom a*
define *s* **where** *s* = *r* / *rr*
from *rr0 r0* **have** *s0*: *s* ≠ *0* **unfolding** *s-def* **by** *auto*
from *is-leq σ*
have *subst-poly-lec σ* (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*))
= *lec-of-nsc* (*r* ∗*R nsc-of-atom a*)
by (*cases a*, *auto simp*: *subst-poly-scaleRat*)
moreover have *r* ∗*R nsc-of-atom a* = *s* ∗*R ns* **unfolding** *ns-def s-def*
scaleRat-scaleRat-ns-constraint[*OF rr0*] **using** *rr0* **by** *simp*
ultimately have *subst-poly-lec σ* (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*))
= *lec-of-nsc* (*s* ∗*R ns*) *is-leq-ns* (*s* ∗*R ns*) **using** *is-leq* **by** *auto*
then show *?thesis*
unfolding *ns-def* **using** *rr s0* **by** *blast*
qed
qed
hence ∀ *ra*. ∃ *s ns*. (*fst ra*, *snd ra*) ∈ *set C* ⟶ *?P* (*fst ra*) (*snd ra*) *s ns* **by** *blast*
from *choice*[*OF this*] **obtain** *s* **where** *s*: ∀ *ra*. ∃ *ns*. (*fst ra*, *snd ra*) ∈ *set C* ⟶ *?P* (*fst ra*) (*snd ra*) (*s ra*) *ns* **by** *blast*
from *choice*[*OF this*] **obtain** *ns* **where** *ns*: ⋀ *r a*. (*r*,*a*) ∈ *set C* ⟹ *?P r a* (*s* (*r*,*a*)) (*ns* (*r*,*a*)) **by** *force*
define *NC* **where** *NC* = *map* (λ(*r*,*a*). (*s* (*r*,*a*), *ns* (*r*,*a*))) *C*
have (∑(*s*, *ns*)←*map* (λ(*r*,*a*). (*s* (*r*,*a*), *ns* (*r*,*a*))) *C′*. *lec-of-nsc* (*s* ∗*R ns*)) =
(∑(*r*, *a*)←*C′*. *subst-poly-lec σ* (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*)))
if *set C′* ⊆ *set C* **for** *C′*
using *that* **proof** (*induction C′*)
case *Nil*
then show *?case* **by** *simp*

next
case (*Cons a C'*)
have ($\sum$ *x*←*a* # *C'*. *lec-of-nsc* (*s x* ∗*R ns x*)) =
lec-of-nsc (*s a* ∗*R ns a*) + ($\sum$ *x*←*C'*. *lec-of-nsc* (*s x* ∗*R ns x*))
by *simp*
also have ($\sum$ *x*←*C'*. *lec-of-nsc* (*s x* ∗*R ns x*)) = ($\sum$ (*r*, *a*)←*C'*. *subst-poly-lec* σ (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*)))
using *Cons* **by** (*auto simp add*: *case-prod-beta' comp-def*)
also have *lec-of-nsc* (*s a* ∗*R ns a*) = *subst-poly-lec* σ (*lec-of-nsc* (*fst a* ∗*R nsc-of-atom* (*snd a*)))
proof −
have *a* ∈ *set C*
using *Cons* **by** *simp*
then show *?thesis*
using *ns* **by** *auto*
qed
finally show *?case*
by (*auto simp add*: *case-prod-beta' comp-def*)
qed
also have ($\sum$ (*r*, *a*)←*C*. *subst-poly-lec* σ (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*)))
= *subst-poly-lec* σ ($\sum$ (*r*, *a*)←*C*. (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*)))
by (*auto simp add*: *subst-poly-lec-sum-list case-prod-beta' comp-def*)
also have ($\sum$ (*r*, *a*)←*C*. (*lec-of-nsc* (*r* ∗*R nsc-of-atom a*))) = *Leqc p c*
using *0* **by** *blast*
also have *subst-poly-lec* σ (*Leqc p c*) = *Leqc* (*subst-poly* σ *p*) *c* **by** *simp*
also have *subst-poly* σ *p* = *0*
proof (*rule all-valuate-zero*)
fix *v* :: *QDelta valuation*
have (*subst-poly* σ *p*) ⦃ *v* ⦄ = (*p* ⦃ λ*x*. ((σ *x*) ⦃ *v* ⦄) ⦄) **by** (*rule valuate-subst-poly*)
also have . . . = *0*
proof (*rule 0*(*4*))
have (σ *a*) ⦃ *v* ⦄ = (*q* ⦃ λ*x*. ((σ *x*) ⦃ *v* ⦄) ⦄) **if** (*a*, *q*) ∈ *set* (*Tableau S*) **for** *a q*
proof −
have *distinct* (*map fst ?T*)
using *normalized-tableau-preprocess' assms* **unfolding** *normalized-tableau-def lhs-def*
by (*auto simp add*: *case-prod-beta'*)
then have *0*: σ *a* = *q*
unfolding σ*-def* **using** *that* **by** *auto*
have *q* ⦃ *v* ⦄ = (*q* ⦃ λ*x*. ((σ *x*) ⦃ *v* ⦄) ⦄)
proof −
have *vars q* ⊆ *rvars ?T*
unfolding *rvars-def* **using** *that* **by** *force*
moreover have (σ *x*) ⦃ *v* ⦄ = *v x* **if** *x* ∈ *rvars ?T* **for** *x*
proof −
have *x* ∉ *lvars* (*Tableau S*)
using *that normalized-tableau-preprocess' assms*

```
          unfolding normalized-tableau-def by blast
        then have x ∉ fst ` set (Tableau S)
          unfolding lvars-def by force
        then have map-of ?T x = None
          using map-of-eq-None-iff by metis
        then have σ x = lp-monom 1 x
          unfolding σ-def by auto
        also have (lp-monom 1 x) ⦃ v ⦄ = v x
          by auto
        finally show ?thesis .
      qed
      ultimately show ?thesis
        by (auto intro!: valuate-depend)
    qed
    then show ?thesis
      using 0 by blast
  qed
  then show (λx. ((σ x) ⦃ v ⦄)) ⊨t ?T
    using 0 by (auto simp add: satisfies-tableau-def satisfies-eq-def)
 qed
 finally show (subst-poly σ p) ⦃ v ⦄ = 0 .
qed
finally have (∑(s, n)←NC. lec-of-nsc (s *R n)) = Le-Constraint Leq-Rel 0 c
  unfolding NC-def by blast
moreover have ns (r,a) ∈ snd ` set cs is-leq-ns (s (r, a) *R ns (r, a)) s (r, a)
≠ 0 if (r, a) ∈ set C for r a
  using ns that by force+
ultimately have farkas-coefficients-ns (snd ` set cs) NC
  unfolding farkas-coefficients-ns-def NC-def using 0 by force
then show ?thesis
  by blast
qed

lemma preprocess′-unsat-indexD: i ∈ set (UnsatIndices (preprocess′ ns j)) ⟹
  ∃ c. poly c = 0 ∧ ¬ zero-satisfies c ∧ (i,c) ∈ set ns
  by (induct ns j rule: preprocess′.induct, auto simp: Let-def split: if-splits op-
tion.splits)

lemma preprocess′-unsat-index-farkas-coefficients-ns:
  assumes i ∈ set (UnsatIndices (preprocess′ ns j))
  shows ∃ C. farkas-coefficients-ns (snd ` set ns) C
proof −
  from preprocess′-unsat-indexD[OF assms]
  obtain c where contr: poly c = 0 ¬ zero-satisfies c and mem: (i,c) ∈ set ns by
auto
  from mem have mem: c ∈ snd ` set ns by force
  let ?c = ns-constraint-const c
  define r where r = (case c of LEQ-ns - - ⇒ 1 | - ⇒ (−1 :: rat))
  define d where d = (case c of LEQ-ns - - ⇒ ?c | - ⇒ − ?c)
```

have [*simp*]: $(- x < 0) = (0 < x)$ **for** *x* :: *QDelta* **using** *uminus-less-lrv*[*of* - *0*] **by** *simp*
show *?thesis* **unfolding** *farkas-coefficients-ns-def*
by (*intro exI*[*of* - [(*r*,*c*)]] *exI*[*of* - *d*], *insert mem contr*, *cases c*, *auto simp*: *r-def d-def*)
qed

The combination of the previous results easily provides the main result of this section: a finite set of non-strict constraints on layer 2 is unsatisfiable if and only if there are Farkas coefficients. Again, here we use results from the simplex formalization, namely soundness of the preprocess-function.

lemma *farkas-coefficients-ns*: **assumes** *finite* (*ns* :: *QDelta ns-constraint set*)
shows ($\exists$ *C*. *farkas-coefficients-ns ns C*) $\longleftrightarrow$ ($\nexists$ *v*. $v \models_{nss}$ *ns*)
proof
assume $\exists$ *C*. *farkas-coefficients-ns ns C*
from *farkas-coefficients-ns-unsat this* **show** $\nexists$ *v*. $v \models_{nss}$ *ns* **by** *blast*
next
assume *unsat*: $\nexists$ *v*. $v \models_{nss}$ *ns*
from *finite-list*[*OF assms*] **obtain** *nsl* **where** *ns*: *ns* = *set nsl* **by** *auto*
let *?cs* = *map* (*Pair* ()) *nsl*
obtain *I t ias* **where** *part1*: *preprocess-part-1 ?cs* = (*t*,*ias*,*I*) **by** (*cases preprocess-part-1 ?cs*, *auto*)
let *?as* = *snd* ‘ *set ias*
let *?s* = *start-fresh-variable ?cs*
have *fin*: *finite ?as* **by** *auto*
have *id*: *ias* = *Atoms* (*preprocess′ ?cs ?s*) *t* = *Tableau* (*preprocess′ ?cs ?s*)
I = *UnsatIndices* (*preprocess′ ?cs ?s*)
using *part1* **unfolding** *preprocess-part-1-def Let-def* **by** *auto*
have *norm*: $\triangle$ *t* **using** *normalized-tableau-preprocess′*[*of ?cs*] **unfolding** *id* **.**
{
fix *v*
assume $v \models_{as}$ *?as* $v \models_t$ *t*
from *preprocess′-sat*[*OF this*[*unfolded id*], *folded id*] *unsat*[*unfolded ns*]
have *set I* $\neq$ {} **by** *auto*
then obtain *i* **where** *i* $\in$ *set I* **using** *all-not-in-conv* **by** *blast*
from *preprocess′-unsat-index-farkas-coefficients-ns*[*OF this*[*unfolded id*]]
have $\exists$ *C*. *farkas-coefficients-ns* (*snd* ‘ *set ?cs*) *C* **by** *simp*
}
with *farkas-coefficients-atoms-tableau*[*OF norm fin*]
obtain *C* **where** *farkas-coefficients-atoms-tableau ?as t C*
$\vee$ ($\exists$ *C*. *farkas-coefficients-ns* (*snd* ‘ *set ?cs*) *C*) **by** *blast*
from *farkas-coefficients-preprocess′*[*of ?cs*, *OF refl*] *this*
have $\exists$ *C*. *farkas-coefficients-ns* (*snd* ‘ *set ?cs*) *C*
using *part1* **unfolding** *preprocess-part-1-def Let-def* **by** *auto*
also have *snd* ‘ *set ?cs* = *ns* **unfolding** *ns* **by** *force*
finally show $\exists$ *C*. *farkas-coefficients-ns ns C* **.**
qed

2.5 Farkas' Lemma on Layer 1

The main difference of layers 1 and 2 is the restriction to non-strict constraints via delta-rationals. Since we now work with another constraint type, *constraint*, we again need translations into linear inequalities of type *le-constraint*. Moreover, we also need to define scaling of constraints where flipping the comparison sign may be required.

fun *is-le* :: *constraint* ⇒ *bool* **where**
is-le (*LT* - -) = *True*
| *is-le* (*LEQ* - -) = *True*
| *is-le* - = *False*

fun *lec-of-constraint* **where**
lec-of-constraint (*LEQ p c*) = (*Le-Constraint Leq-Rel p c*)
| *lec-of-constraint* (*LT p c*) = (*Le-Constraint Lt-Rel p c*)

lemma *lec-of-constraint*:
assumes *is-le c*
shows ($v \models_{le}$ (*lec-of-constraint c*)) ⟷ ($v \models_c c$)
using *assms* **by** (*cases c*, *auto*)

instantiation *constraint* :: *scaleRat*
begin
fun *scaleRat-constraint* :: *rat* ⇒ *constraint* ⇒ *constraint* **where**
scaleRat-constraint r cc = (*if r = 0 then LEQ 0 0 else*
(*case cc of*
LEQ p c ⇒
(*if* (*r* < *0*) *then GEQ* (*r* ∗*R p*) (*r* ∗*R c*) *else LEQ* (*r* ∗*R p*) (*r* ∗*R c*))
| *LT p c* ⇒
(*if* (*r* < *0*) *then GT* (*r* ∗*R p*) (*r* ∗*R c*) *else LT* (*r* ∗*R p*) (*r* ∗*R c*))
| *GEQ p c* ⇒
(*if* (*r* > *0*) *then GEQ* (*r* ∗*R p*) (*r* ∗*R c*) *else LEQ* (*r* ∗*R p*) (*r* ∗*R c*))
| *GT p c* ⇒
(*if* (*r* > *0*) *then GT* (*r* ∗*R p*) (*r* ∗*R c*) *else LT* (*r* ∗*R p*) (*r* ∗*R c*))
| *EQ p c* ⇒ *LEQ* (*r* ∗*R p*) (*r* ∗*R c*) — We do not keep equality, since the aim is to convert the scaled constraints into inequalities, which will then be summed up.
))

instance ..
end

lemma *sat-scale-rat*: **assumes** (*v* :: *rat valuation*) $\models_c c$
shows $v \models_c$ (*r* ∗*R c*)
proof −
have *r* < *0* ∨ *r* = *0* ∨ *r* > *0* **by** *auto*
then show *?thesis* **using** *assms* **by** (*cases c*, *auto simp*: *right-diff-distrib valuate-minus valuate-scaleRat scaleRat-leq1 scaleRat-leq2 valuate-zero*)
qed

In the following definition of Farkas coefficients (for layer 1), the main difference to *farkas-coefficients-ns* is that the linear combination evaluates either to a strict inequality where the constant must be non-positive, or to a non-strict inequality where the constant must be negative.

definition *farkas-coefficients* **where**
  farkas-coefficients cs C = ($\exists$ *d rel*.
    ($\forall$ (*r*,*c*) $\in$ *set C*. *c* $\in$ *cs* $\wedge$ *is-le* (*r* ∗*R c*) $\wedge$ *r* $\neq$ *0*) $\wedge$
    ($\sum$ (*r*,*c*) $\leftarrow$ *C*. *lec-of-constraint* (*r* ∗*R c*)) = *Le-Constraint rel 0 d* $\wedge$
    (*rel* = *Leq-Rel* $\wedge$ *d* < *0* $\vee$ *rel* = *Lt-Rel* $\wedge$ *d* $\leq$ *0*))

Again, the existence Farkas coefficients immediately implies unsatisfiability.

lemma *farkas-coefficients-unsat*:
  assumes *farkas-coefficients cs C*
  shows $\nexists$ *v*. *v* $\models_{cs}$ *cs*
proof
  assume $\exists$ *v*. *v* $\models_{cs}$ *cs*
  then obtain *v* **where** ∗: *v* $\models_{cs}$ *cs* **by** *auto*
  obtain *d rel* **where**
    isleq: ($\forall$ (*r*,*c*) $\in$ *set C*. *c* $\in$ *cs* $\wedge$ *is-le* (*r* ∗*R c*) $\wedge$ *r* $\neq$ *0*) **and**
    leq: ($\sum$ (*r*,*c*) $\leftarrow$ *C*. *lec-of-constraint* (*r* ∗*R c*)) = *Le-Constraint rel 0 d* **and**
    choice: *rel* = *Lt-Rel* $\wedge$ *d* $\leq$ *0* $\vee$ *rel* = *Leq-Rel* $\wedge$ *d* < *0* **using** *assms farkas-coefficients-def*
by *blast*
  {
    fix *r c*
    assume *c*: (*r*,*c*) $\in$ *set C*
    from *c* ∗ *isleq* **have** *v* $\models_{c}$ *c* **by** *auto*
    hence *v*: *v* $\models_{c}$ (*r* ∗*R c*) **by** (*rule sat-scale-rat*)
    from *c isleq* **have** *is-le* (*r* ∗*R c*) **by** *auto*
    from *lec-of-constraint*[*OF this*] *v*
    have *v* $\models_{le}$ *lec-of-constraint* (*r* ∗*R c*) **by** *blast*
  } note *v* = *this*
  have *v* $\models_{le}$ *Le-Constraint rel 0 d* **unfolding** *leq*[*symmetric*]
    by (*rule satisfies-sumlist-le-constraints*, *insert v*, *auto*)
  then show *False* **using** *choice*
    by (*cases rel*, *auto simp*: *valuate-zero*)
qed

Now follows the difficult implication. The major part is proving that the translation *constraint-to-qdelta-constraint* preserves the existence of Farkas coefficients via pointwise compatibility of the sum. Here, compatibility links a strict or non-strict inequality from the input constraint to a translated non-strict inequality over delta-rationals.

fun *compatible-cs* **where**
  compatible-cs (*Le-Constraint Leq-Rel p c*) (*Le-Constraint Leq-Rel q d*) = (*q* = *p* $\wedge$ *d* = *QDelta c 0*)
| *compatible-cs* (*Le-Constraint Lt-Rel p c*) (*Le-Constraint Leq-Rel q d*) = (*q* = *p* $\wedge$ *qdfst d* = *c*)

| *compatible-cs - - = False*

lemma *compatible-cs-0-0*: *compatible-cs 0 0* **by** *code-simp*

lemma *compatible-cs-plus*: *compatible-cs c1 d1* ⟹ *compatible-cs c2 d2* ⟹ *compatible-cs* (*c1* + *c2*) (*d1* + *d2*)
by (*cases c1*; *cases d1*; *cases c2*; *cases d2*; *cases lec-rel c1*; *cases lec-rel d1*; *cases lec-rel c2*;
cases lec-rel d2; *auto simp*: *plus-QDelta-def*)

lemma *unsat-farkas-coefficients*: **assumes** ∄ *v*. *v* $\models_{cs}$ *cs*
and *fin*: *finite cs*
shows ∃ *C*. *farkas-coefficients cs C*
proof −
from *finite-list*[*OF fin*] **obtain** *csl* **where** *cs*: *cs* = *set csl* **by** *blast*
let *?csl* = *map* (*Pair* ()) *csl*
let *?ns* = (*snd* ‘ *set* (*to-ns ?csl*))
let *?nsl* = *concat* (*map constraint-to-qdelta-constraint csl*)
have *id*: *snd* ‘ *set ?csl* = *cs* **unfolding** *cs* **by** *force*
have *id2*: *?ns* = *set ?nsl* **unfolding** *to-ns-def set-concat* **by** *force*
from *SolveExec'Default.to-ns-sat*[*of ?csl*, *unfolded id*] *assms*
have *unsat*: ∄ *v*. ⟨*v*⟩ $\models_{nss}$ *?ns* **by** *metis*
have *fin*: *finite ?ns* **by** *auto*
have ∄ *v*. *v* $\models_{nss}$ *?ns*
proof
assume ∃ *v*. *v* $\models_{nss}$ *?ns*
then obtain *v* **where** *model*: *v* $\models_{nss}$ *?ns* **by** *blast*
let *?v* = *Mapping.Mapping* (λ *x*. *Some* (*v x*))
have *v* = ⟨*?v*⟩ **by** (*intro ext*, *auto simp*: *map2fun-def Mapping.lookup.abs-eq*)
from *model this unsat* **show** *False* **by** *metis*
qed
from *farkas-coefficients-ns*[*OF fin*] *this id2* **obtain** *N* **where**
farkas: *farkas-coefficients-ns* (*set ?nsl*) *N* **by** *metis*
from *this*[*unfolded farkas-coefficients-ns-def*]
obtain *d* **where**
is-leq: ⋀ *a n*. (*a*,*n*) ∈ *set N* ⟹ *n* ∈ *set ?nsl* ∧ *is-leq-ns* (*a* ∗*R n*) ∧ *a* ≠ *0* **and**

sum: (∑(*a*,*n*)←*N*. *lec-of-nsc* (*a* ∗*R n*)) = *Le-Constraint Leq-Rel 0 d* **and**
d0: *d* < *0* **by** *blast*
let *?prop* = λ *NN C*. (∀ (*a*,*c*) ∈ *set C*. *c* ∈ *cs* ∧ *is-le* (*a* ∗*R c*) ∧ *a* ≠ *0*)
∧ *compatible-cs* (∑ (*a*,*c*) ← *C*. *lec-of-constraint* (*a* ∗*R c*))
(∑(*a*,*n*)←*NN*. *lec-of-nsc* (*a* ∗*R n*))
have *set NN* ⊆ *set N* ⟹ ∃ *C*. *?prop NN C* **for** *NN*
proof (*induct NN*)
case *Nil*
have *?prop Nil Nil* **by** (*simp add*: *compatible-cs-0-0*)
thus *?case* **by** *blast*
next
case (*Cons an NN*)

obtain *a n* **where** *an*: *an* = (*a*,*n*) **by** *force*
from *Cons an* **obtain** *C* **where** *IH*: *?prop NN C* **and** *n*: (*a*,*n*) ∈ *set N* **by** *auto*
have *compat-CN*: *compatible-cs* (∑(*f*, *c*)←*C*. *lec-of-constraint* (*f* ∗*R* *c*))
(∑(*a*,*n*)←*NN*. *lec-of-nsc* (*a* ∗*R* *n*))
using *IH* **by** *blast*
from *n is-leq* **obtain** *c* **where** *c*: *c* ∈ *cs* **and** *nc*: *n* ∈ *set* (*constraint-to-qdelta-constraint* *c*)
unfolding *cs* **by** *force*
from *is-leq*[*OF n*] **have** *is-leq*: *is-leq-ns* (*a* ∗*R* *n*) ∧ *a* ≠ *0* **by** *blast*
have *is-less*: *is-le* (*a* ∗*R* *c*) **and**
a0: *a* ≠ *0* **and**
compat-cn: *compatible-cs* (*lec-of-constraint* (*a* ∗*R* *c*)) (*lec-of-nsc* (*a* ∗*R* *n*))
by (*atomize*(*full*), *cases c*, *insert is-leq nc*, *auto simp*: *QDelta-0-0 scaleRat-QDelta-def qdsnd-0 qdfst-0*)
let *?C* = *Cons* (*a*, *c*) *C*
let *?N* = *Cons* (*a*, *n*) *NN*
have *?prop ?N ?C* **unfolding** *an*
proof (*intro conjI*)
show ∀ (*a*,*c*) ∈ *set ?C*. *c* ∈ *cs* ∧ *is-le* (*a* ∗*R* *c*) ∧ *a* ≠ *0* **using** *IH is-less a0 c* **by** *auto*
show *compatible-cs* (∑(*a*, *c*)←*?C*. *lec-of-constraint* (*a* ∗*R* *c*)) (∑(*a*,*n*)←*?N*. *lec-of-nsc* (*a* ∗*R* *n*))
using *compatible-cs-plus*[*OF compat-cn compat-CN*] **by** *simp*
qed
thus *?case* **unfolding** *an* **by** *blast*
qed
from *this*[*OF subset-refl*, *unfolded sum*]
obtain *C* **where**
is-less: (∀(*a*, *c*)∈*set C*. *c* ∈ *cs* ∧ *is-le* (*a* ∗*R* *c*) ∧ *a* ≠ *0*) **and**
compat: *compatible-cs* (∑(*f*, *c*)←*C*. *lec-of-constraint* (*f* ∗*R* *c*)) (*Le-Constraint Leq-Rel 0 d*)
(**is** *compatible-cs ?sum* -)
by *blast*
obtain *rel p e* **where** *?sum* = *Le-Constraint rel p e* **by** (*cases ?sum*)
with *compat* **have** *sum*: *?sum* = *Le-Constraint rel 0 e* **by** (*cases rel*, *auto*)
have *e*: (*rel* = *Leq-Rel* ∧ *e* < *0* ∨ *rel* = *Lt-Rel* ∧ *e* ≤ *0*) **using** *compat*[*unfolded sum*] *d0*
by (*cases rel*, *auto simp*: *less-QDelta-def qdfst-0 qdsnd-0*)
show *?thesis* **unfolding** *farkas-coefficients-def*
by (*intro exI conjI*, *rule is-less*, *rule sum*, *insert e*, *auto*)
qed

Finally we can prove on layer 1 that a finite set of constraints is unsatisfiable if and only if there are Farkas coefficients.

lemma *farkas-coefficients*: **assumes** *finite cs*
shows (∃ *C*. *farkas-coefficients cs C*) ⟷ (∄ *v*. *v* $\models_{cs}$ *cs*)
using *farkas-coefficients-unsat unsat-farkas-coefficients*[*OF - assms*] **by** *blast*

3 Corollaries from the Literature

In this section, we convert the previous variations of Farkas' Lemma into more well-known forms of this result. Moreover, instead of referring to the various constraint types of the simplex formalization, we now speak solely about constraints of type *le-constraint*.

3.1 Farkas' Lemma on Delta-Rationals

We start with Lemma 2 of [1], a variant of Farkas' Lemma for delta-rationals. To be more precise, it states that a set of non-strict inequalities over delta-rationals is unsatisfiable if and only if there is a linear combination of the inequalities that results in a trivial unsatisfiable constraint $0 < const$ for some negative constant *const*. We can easily prove this statement via the lemma *farkas-coefficients-ns* and some conversions between the different constraint types.

lemma *Farkas′-Lemma-Delta-Rationals*: **fixes** *cs* :: *QDelta le-constraint set*
  assumes *only-non-strict*: *lec-rel ‘ cs* ⊆ {*Leq-Rel*}
    and *fin*: *finite cs*
  shows (∄ *v*. ∀ *c* ∈ *cs*. $v \models_{le}$ *c*) ⟷
      (∃ *C const*. (∀ (*r*, *c*) ∈ *set C*. *r* > *0* ∧ *c* ∈ *cs*)
       ∧ (∑ (*r*,*c*) ← *C*. *Leqc* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) = *Leqc 0 const*
       ∧ *const* < *0*)
    (**is** *?lhs* = *?rhs*)
proof −
  {
    fix *c*
    assume *c* ∈ *cs*
    with *only-non-strict* **have** *lec-rel c* = *Leq-Rel* **by** *auto*
    then have ∃ *p const*. *c* = *Leqc p const* **by** (*cases c*, *auto*)
  } note *leqc* = *this*
  let *?to-ns* = λ *c*. *LEQ-ns* (*lec-poly c*) (*lec-const c*)
  let *?ns* = *?to-ns ‘ cs*
  from *fin* **have** *fin*: *finite ?ns* **by** *auto*
  have *v* $\models_{nss}$ *?ns* ⟷ (∀ *c* ∈ *cs*. *v* $\models_{le}$ *c*) **for** *v* **using** *leqc* **by** *fastforce*
  hence *?lhs* = (∄ *v*. *v* $\models_{nss}$ *?ns*) **by** *simp*
 also have ... = (∃ *C*. *farkas-coefficients-ns ?ns C*) **unfolding** *farkas-coefficients-ns*[*OF fin*] **..**
  also have ... = *?rhs*
  proof
    assume ∃ *C*. *farkas-coefficients-ns ?ns C*
    then obtain *C const* **where** *is-leq*: ∀ (*s*, *n*) ∈ *set C*. *n* ∈ *?ns* ∧ *is-leq-ns* (*s* ∗*R n*) ∧ *s* ≠ *0*
      and *sum*: (∑(*s*, *n*)←*C*. *lec-of-nsc* (*s* ∗*R n*)) = *Leqc 0 const*
      and *c0*: *const* < *0* **unfolding** *farkas-coefficients-ns-def* **by** *blast*
    let *?C* = *map* (λ (*s*,*n*). (*s*,*lec-of-nsc n*)) *C*
    show *?rhs*

proof (*intro exI*[*of - ?C*] *exI*[*of - const*] *conjI c0*, *unfold sum*[*symmetric*] *map-map o-def set-map*,
intro ballI, *clarify*)
{
fix *s n*
assume *sn*: (*s*, *n*) ∈ *set C*
with *is-leq*
have *n-ns*: *n* ∈ *?ns* **and** *is-leq*: *is-leq-ns* (*s* ∗*R n*) *s* ≠ *0* **by** *force+*
from *n-ns* **obtain** *c* **where** *c*: *c* ∈ *cs* **and** *n*: *n* = *LEQ-ns* (*lec-poly c*) (*lec-const c*) **by** *auto*
with *leqc*[*OF c*] **obtain** *p d* **where** *cs*: *Leqc p d* ∈ *cs* **and** *n*: *n* = *LEQ-ns p d* **by** *auto*
from *is-leq*[*unfolded n*] **have** *s0*: *s* > *0* **by** (*auto split*: *if-splits*)
let *?n* = *lec-of-nsc n*
from *cs n* **have** *mem*: *?n* ∈ *cs* **by** *auto*
show *0* < *s* ∧ *?n* ∈ *cs* **using** *s0 mem* **by** *blast*
have *Leqc* (*s* ∗*R lec-poly ?n*) (*s* ∗*R lec-const ?n*) = *lec-of-nsc* (*s* ∗*R n*)
unfolding *n* **using** *s0* **by** *simp*
} **note** *id* = *this*
show (∑*x*←*C*. *case case x of* (*s*, *n*) ⇒ (*s*, *lec-of-nsc n*) *of*
(*r*, *c*) ⇒ *Leqc* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) =
(∑(*s*, *n*)←*C*. *lec-of-nsc* (*s* ∗*R n*)) (**is** *sum-list* (*map ?f C*) = *sum-list* (*map ?g C*))
proof (*rule arg-cong*[*of - - sum-list*], *rule map-cong*[*OF refl*])
fix *pair*
assume *mem*: *pair* ∈ *set C*
then obtain *s n* **where** *pair*: *pair* = (*s*,*n*) **by** *force*
show *?f pair* = *?g pair* **unfolding** *pair split* **using** *id*[*OF mem*[*unfolded pair*]] **.**
qed
qed
next
assume *?rhs*
then obtain *C const*
where *C*: ⋀ *r c*. (*r*,*c*) ∈ *set C* ⟹ *0* < *r* ∧ *c* ∈ *cs*
and *sum*: (∑(*r*, *c*)←*C*. *Leqc* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) = *Leqc 0 const*
and *const*: *const* < *0*
by *blast*
let *?C* = *map* (λ (*r*,*c*). (*r*, *?to-ns c*)) *C*
show ∃ *C*. *farkas-coefficients-ns ?ns C* **unfolding** *farkas-coefficients-ns-def*
proof (*intro exI*[*of - ?C*] *exI*[*of - const*] *conjI const*, *unfold sum*[*symmetric*])
show ∀(*s*, *n*)∈*set ?C*. *n* ∈ *?ns* ∧ *is-leq-ns* (*s* ∗*R n*) ∧ *s* ≠ *0* **using** *C* **by** *fastforce*
show (∑(*s*, *n*)←*?C*. *lec-of-nsc* (*s* ∗*R n*))
= (∑(*r*, *c*)←*C*. *Leqc* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*))
unfolding *map-map o-def*
by (*rule arg-cong*[*of - - sum-list*], *rule map-cong*[*OF refl*], *insert C*, *force*)
qed

qed
finally show *?thesis* **.**
qed

3.2 Motzkin's Transposition Theorem or the Kuhn-Fourier Theorem

Next, we prove a generalization of Farkas' Lemma that permits arbitrary combinations of strict and non-strict inequalities: Motzkin's Transposition Theorem which is also known as the Kuhn–Fourier Theorem.

The proof is mainly based on the lemma *farkas-coefficients*, again requiring conversions between constraint types.

```
theorem Motzkin's-transposition-theorem: fixes cs :: rat le-constraint set
  assumes fin: finite cs
  shows (∄ v. ∀ c ∈ cs. v ⊨_le c) ⟷
     (∃ C const rel. (∀ (r, c) ∈ set C. r > 0 ∧ c ∈ cs)
      ∧ (∑ (r,c) ← C. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const c))
           = Le-Constraint rel 0 const
       ∧ (rel = Leq-Rel ∧ const < 0 ∨ rel = Lt-Rel ∧ const ≤ 0))
    (is ?lhs = ?rhs)
proof −
  let ?to-cs = λ c. (case lec-rel c of Leq-Rel ⇒ LEQ | - ⇒ LT) (lec-poly c) (lec-const c)
  have to-cs: v ⊨_c ?to-cs c ⟷ v ⊨_le c for v c by (cases c, cases lec-rel c, auto)
  let ?cs = ?to-cs ‘ cs
  from fin have fin: finite ?cs by auto
  have v ⊨_cs ?cs ⟷ (∀ c ∈ cs. v ⊨_le c) for v using to-cs by auto
  hence ?lhs = (∄ v. v ⊨_cs ?cs) by simp
  also have ... = (∃ C. farkas-coefficients ?cs C) unfolding farkas-coefficients[OF fin] ..
  also have ... = ?rhs
  proof
    assume ∃ C. farkas-coefficients ?cs C
    then obtain C const rel where is-leq: ∀ (s, n) ∈ set C. n ∈ ?cs ∧ is-le (s *R n) ∧ s ≠ 0
      and sum: (∑ (s, n)←C. lec-of-constraint (s *R n)) = Le-Constraint rel 0 const
      and c0: (rel = Leq-Rel ∧ const < 0 ∨ rel = Lt-Rel ∧ const ≤ 0)
      unfolding farkas-coefficients-def by blast
    let ?C = map (λ (s,n). (s,lec-of-constraint n)) C
    show ?rhs
    proof (intro exI[of - ?C] exI[of - const] exI[of - rel] conjI c0, unfold map-map o-def set-map sum[symmetric],
        intro ballI, clarify)
      {
        fix s n
        assume sn: (s, n) ∈ set C
```

```
      with is-leq
      have n-ns: n ∈ ?cs and is-leq: is-le (s *R n) and s0: s ≠ 0 by force+
      from n-ns obtain c where c: c ∈ cs and n: n = ?to-cs c by auto
      from is-leq[unfolded n] have s ≥ 0 by (cases lec-rel c, auto split: if-splits)
      with s0 have s0: s > 0 by auto
      let ?c = lec-of-constraint n
      from c n have mem: ?c ∈ cs by (cases c, cases lec-rel c, auto)
      show 0 < s ∧ ?c ∈ cs using s0 mem by blast
      have lec-of-constraint (s *R n) = Le-Constraint (lec-rel ?c) (s *R lec-poly ?c) (s *R lec-const ?c)
        unfolding n using s0 by (cases c, cases lec-rel c, auto)
    } note id = this
    show (∑ x←C. case case x of (s, n) ⇒ (s, lec-of-constraint n) of
         (r, c) ⇒ Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const c)) =
         (∑ (s, n)←C. lec-of-constraint (s *R n))
      (is sum-list (map ?f C) = sum-list (map ?g C))
    proof (rule arg-cong[of - - sum-list], rule map-cong[OF refl])
      fix pair
      assume mem: pair ∈ set C
      obtain r c where pair: pair = (r,c) by force
      show ?f pair = ?g pair unfolding pair split id[OF mem[unfolded pair]] ..
    qed
  qed
next
  assume ?rhs
  then obtain C const rel
    where C: ⋀ r c. (r,c) ∈ set C ⟹ 0 < r ∧ c ∈ cs
      and sum: (∑ (r, c)←C. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const c))
          = Le-Constraint rel 0 const
      and const: rel = Leq-Rel ∧ const < 0 ∨ rel = Lt-Rel ∧ const ≤ 0
    by blast
  let ?C = map (λ (r,c). (r, ?to-cs c)) C
  show ∃ C. farkas-coefficients ?cs C unfolding farkas-coefficients-def
    proof (intro exI[of - ?C] exI[of - const] exI[of - rel] conjI const, unfold sum[symmetric])
     show ∀ (s, n)∈set ?C. n ∈ ?cs ∧ is-le (s *R n) ∧ s ≠ 0 using C by (fastforce split: le-rel.splits)
    show (∑ (s, n)←?C. lec-of-constraint (s *R n))
      = (∑ (r, c)←C. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const c))
      unfolding map-map o-def
      by (rule arg-cong[of - - sum-list], rule map-cong[OF refl], insert C, fastforce split: le-rel.splits)
  qed
qed
finally show ?thesis .
qed
```

3.3 Farkas' Lemma

Finally we derive the commonly used form of Farkas' Lemma, which easily follows from *Motzkin's-transposition-theorem*. It only permits non-strict inequalities and, as a result, the sum of inequalities will always be non-strict.

```
lemma Farkas'-Lemma: fixes cs :: rat le-constraint set
  assumes only-non-strict: lec-rel ` cs ⊆ {Leq-Rel}
    and fin: finite cs
  shows (∄ v. ∀ c ∈ cs. v ⊨le c) ⟷
     (∃ C const. (∀ (r, c) ∈ set C. r > 0 ∧ c ∈ cs)
      ∧ (∑ (r,c) ← C. Leqc (r *R lec-poly c) (r *R lec-const c)) = Leqc 0 const
      ∧ const < 0)
    (is - = ?rhs)
proof −
  {
    fix c
    assume c ∈ cs
    with only-non-strict have lec-rel c = Leq-Rel by auto
    then have ∃ p const. c = Leqc p const by (cases c, auto)
  } note leqc = this
  let ?lhs = ∃ C const rel.
      (∀(r, c)∈set C. 0 < r ∧ c ∈ cs) ∧
      (∑(r, c)←C. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const c))
         = Le-Constraint rel 0 const ∧
      (rel = Leq-Rel ∧ const < 0 ∨ rel = Lt-Rel ∧ const ≤ 0)
  show ?thesis unfolding Motzkin's-transposition-theorem[OF fin]
  proof
    assume ?rhs
    then obtain C const where C: ⋀ r c. (r, c)∈set C ⟹ 0 < r ∧ c ∈ cs and
      sum: (∑(r, c)←C. Leqc (r *R lec-poly c) (r *R lec-const c)) = Leqc 0 const
and
      const: const < 0 by blast
    show ?lhs
    proof (intro exI[of - C] exI[of - const] exI[of - Leq-Rel] conjI)
      show ∀ (r,c) ∈ set C. 0 < r ∧ c ∈ cs using C by force
     show (∑(r, c)← C. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const
c)) =
        Leqc 0 const unfolding sum[symmetric]
         by (rule arg-cong[of - - sum-list], rule map-cong[OF refl], insert C, force
dest!: leqc)
    qed (insert const, auto)
  next
    assume ?lhs
     then obtain C const rel where C: ⋀ r c. (r, c)∈set C ⟹ 0 < r ∧ c ∈ cs
and
      sum: (∑(r, c)←C. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const
c))
        = Le-Constraint rel 0 const  and
      const: rel = Leq-Rel ∧ const < 0 ∨ rel = Lt-Rel ∧ const ≤ 0 by blast
```

have *id*: ($\sum$(*r*, *c*)←*C*. *Le-Constraint* (*lec-rel c*) (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) =
($\sum$(*r*, *c*)←*C*. *Leqc* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) (**is** - = *?sum*)
by (*rule arg-cong*[*of* - - *sum-list*], *rule map-cong*, *auto dest*!: *C leqc*)
have *lec-rel ?sum* = *Leq-Rel* **unfolding** *sum-list-lec* **by** (*auto simp add*: *sum-list-Leq-Rel o-def*)
with *sum*[*unfolded id*] **have** *rel*: *rel* = *Leq-Rel* **by** *auto*
with *const* **have** *const*: *const* < *0* **by** *auto*
show *?rhs*
by (*intro exI*[*of* - *C*] *exI*[*of* - *const*] *conjI const*, *insert sum id C rel*, *force*+)
qed
qed

We also present slightly modified versions

lemma *sum-list-map-filter-sum*: **fixes** *f* :: *'a* ⇒ *'b* :: *comm-monoid-add*
shows *sum-list* (*map f* (*filter g xs*)) + *sum-list* (*map f* (*filter* (*Not o g*) *xs*)) = *sum-list* (*map f xs*)
by (*induct xs*, *auto simp*: *ac-simps*)

A version where every constraint obtains exactly one coefficient and where 0 coefficients are allowed.

lemma *Farkas'-Lemma-set-sum*: **fixes** *cs* :: *rat le-constraint set*
assumes *only-non-strict*: *lec-rel* ‘ *cs* ⊆ {*Leq-Rel*}
and *fin*: *finite cs*
shows ($\nexists$ *v*. ∀ *c* ∈ *cs*. *v* $\models_{le}$ *c*) ⟷
(∃ *C const*. (∀ *c* ∈ *cs*. *C c* ≥ *0*)
∧ ($\sum$ *c* ∈ *cs*. *Leqc* ((*C c*) ∗*R lec-poly c*) ((*C c*) ∗*R lec-const c*)) = *Leqc 0 const*
∧ *const* < *0*)
unfolding *Farkas'-Lemma*[*OF only-non-strict fin*]
proof ((*standard*; *elim exE conjE*), *goal-cases*)
case (*2 C const*)
from *finite-distinct-list*[*OF fin*] **obtain** *csl* **where** *csl*: *set csl* = *cs* **and** *dist*: *distinct csl*
by *auto*
let *?list* = *filter* (λ *c*. *C c* ≠ *0*) *csl*
let *?C* = *map* (λ *c*. (*C c*, *c*)) *?list*
show *?case*
proof (*intro exI*[*of* - *?C*] *exI*[*of* - *const*] *conjI*)
have ($\sum$(*r*, *c*)←*?C*. *Le-Constraint Leq-Rel* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*))
= ($\sum$(*r*, *c*)←*map* (λ*c*. (*C c*, *c*)) *csl*. *Le-Constraint Leq-Rel* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*))
unfolding *map-map*
by (*rule sum-list-map-filter*, *auto simp*: *zero-le-constraint-def*)
also have ... = *Le-Constraint Leq-Rel 0 const* **unfolding** *2*(*2*)[*symmetric*] *csl*[*symmetric*]
unfolding *sum.distinct-set-conv-list*[*OF dist*] *map-map o-def split* **..**
finally

show ($\sum$(*r*, *c*)←*?C*. *Le-Constraint Leq-Rel* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) = *Le-Constraint Leq-Rel 0 const*
by *auto*
show *const* < *0* **by** *fact*
show ∀(*r*, *c*)∈*set ?C*. *0* < *r* ∧ *c* ∈ *cs* **using** *2*(*1*) **unfolding** *set-map set-filter csl* **by** *auto*
qed
next
case (*1 C const*)
define *CC* **where** *CC* = (λ *c*. *sum-list* (*map fst* (*filter* (λ *rc*. *snd rc* = *c*) *C*)))
show (∃ *C const*. (∀ *c* ∈ *cs*. *C c* ≥ *0*)
∧ ($\sum$ *c* ∈ *cs*. *Leqc* ((*C c*) ∗*R lec-poly c*) ((*C c*) ∗*R lec-const c*)) = *Leqc 0 const*
∧ *const* < *0*)
proof (*intro exI*[*of - CC*] *exI*[*of - const*] *conjI*)
show ∀ *c*∈*cs*. *0* ≤ *CC c* **unfolding** *CC-def* **using** *1*(*1*)
by (*force intro*!: *sum-list-nonneg*)
show *const* < *0* **by** *fact*
from *1* **have** *snd*: *snd* ' *set C* ⊆ *cs* **by** *auto*
show ($\sum$ *c*∈*cs*. *Le-Constraint Leq-Rel* (*CC c* ∗*R lec-poly c*) (*CC c* ∗*R lec-const c*)) = *Le-Constraint Leq-Rel 0 const*
unfolding *1*(*2*)[*symmetric*] **using** *fin snd* **unfolding** *CC-def*
proof (*induct cs arbitrary*: *C rule*: *finite-induct*)
case *empty*
hence *C*: *C* = [] **by** *auto*
thus *?case* **by** *simp*
next
case ∗: (*insert c cs C*)
let *?D* = *filter* (*Not* ∘ (λ*rc*. *snd rc* = *c*)) *C*
from ∗ **have** *snd* ' *set ?D* ⊆ *cs* **by** *auto*
note *IH* = ∗(*3*)[*OF this*]
have *id*: ($\sum$ *a*← *?D*. *case a of* (*r*, *c*) ⇒ *Le-Constraint Leq-Rel* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) =
($\sum$(*r*, *c*)←*?D*. *Le-Constraint Leq-Rel* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*))
by (*induct C*, *force*+)
show *?case*
unfolding *sum.insert*[*OF* ∗(*1*,*2*)]
unfolding *sum-list-map-filter-sum*[*of - λ rc. snd rc = c C*, *symmetric*]
proof (*rule arg-cong2*[*of - - - -* (+)], *goal-cases*)
case *2*
show *?case* **unfolding** *IH*[*symmetric*]
by (*rule sum.cong*, *insert* ∗(*2*,*1*), *auto intro*!: *arg-cong*[*of - - λ xs. sum-list* (*map - xs*)], (*induct C*, *auto*)+)
next
case *1*
show *?case*
proof (*rule sym*, *induct C*)
case (*Cons rc C*)
thus *?case* **by** (*cases rc*, *cases snd rc* = *c*, *auto simp*: *field-simps*

scaleRat-left-distrib)
        qed (*auto simp*: *zero-le-constraint-def*)
      qed
    qed
  qed
qed

A version with indexed constraints, i.e., in particular where constraints may occur several times.

lemma *Farkas′-Lemma-indexed*: **fixes** *c* :: *nat* ⇒ *rat le-constraint*
  assumes *only-non-strict*: *lec-rel* ‘ *c* ‘ *Is* ⊆ {*Leq-Rel*}
  and *fin*: *finite Is*
  shows ($\nexists$ *v*. ∀ *i* ∈ *Is*. *v* $\models_{le}$ *c i*) ⟷
      (∃ *C const*. (∀ *i* ∈ *Is*. *C i* ≥ *0*)
        ∧ ($\sum$ *i* ∈ *Is*. *Leqc* ((*C i*) ∗*R lec-poly* (*c i*)) ((*C i*) ∗*R lec-const* (*c i*))) = *Leqc 0 const*
        ∧ *const* < *0*)
proof −
  let *?C* = *c* ‘ *Is*
  have *fin*: *finite ?C* **using** *fin* **by** *auto*
  have ($\nexists$ *v*. ∀ *i* ∈ *Is*. *v* $\models_{le}$ *c i*) = ($\nexists$ *v*. ∀ *cc* ∈ *?C*. *v* $\models_{le}$ *cc*) **by** *force*
  also have ... = (∃ *C const*. (∀ *i* ∈ *Is*. *C i* ≥ *0*)
        ∧ ($\sum$ *i* ∈ *Is*. *Leqc* ((*C i*) ∗*R lec-poly* (*c i*)) ((*C i*) ∗*R lec-const* (*c i*))) = *Leqc 0 const*
        ∧ *const* < *0*) (**is** *?l* = *?r*)
  proof
    assume *?r*
    then obtain *C const* **where** *r*: (∀ *i* ∈ *Is*. *C i* ≥ *0*)
        and *eq*: ($\sum$ *i* ∈ *Is*. *Leqc* ((*C i*) ∗*R lec-poly* (*c i*)) ((*C i*) ∗*R lec-const* (*c i*))) = *Leqc 0 const*
        and *const* < *0* **by** *auto*
    from *finite-distinct-list*[*OF* ‹*finite Is*›]
      obtain *Isl* **where** *isl*: *set Isl* = *Is* **and** *dist*: *distinct Isl* **by** *auto*
    let *?CC* = *filter* (λ *rc*. *fst rc* ≠ *0*) (*map* (λ *i*. (*C i*, *c i*)) *Isl*)
    show *?l* **unfolding** *Farkas′-Lemma*[*OF only-non-strict fin*]
    proof (*intro exI*[*of - ?CC*] *exI*[*of - const*] *conjI*)
      show *const* < *0* **by** *fact*
      show ∀ (*r*, *ca*) ∈ *set ?CC*. *0* < *r* ∧ *ca* ∈ *?C* **using** *r*(*1*) *isl* **by** *auto*
     show ($\sum$(*r*, *c*)←*?CC*. *Le-Constraint Leq-Rel* (*r* ∗*R lec-poly c*) (*r* ∗*R lec-const c*)) =
        Le-Constraint Leq-Rel 0 const **unfolding** *eq*[*symmetric*]
        by (*subst sum-list-map-filter*, *force simp*: *zero-le-constraint-def*,
          unfold map-map o-def, *subst sum-list-distinct-conv-sum-set*[*OF dist*], *rule sum.cong*, *auto simp*: *isl*)
    qed
  next
    assume *?l*
    from *this*[*unfolded Farkas′-Lemma-set-sum*[*OF only-non-strict fin*]]
    obtain *C const* **where** *nonneg*: (∀ *c*∈ *?C*. *0* ≤ *C c*)

and *sum*: ($\sum$ *c*∈ *?C*. *Le-Constraint Leq-Rel* (*C c* ∗*R lec-poly c*) (*C c* ∗*R lec-const c*)) =
Le-Constraint Leq-Rel 0 const
and *const*: *const* < *0*
by *blast*
define *I* **where** *I* = (λ *i*. (*C* (*c i*) / *rat-of-nat* (*card* (*Is* ∩ { *j*. *c i* = *c j*}))))
show *?r*
proof (*intro exI*[*of - I*] *exI*[*of - const*] *conjI const*)
show ∀ *i* ∈ *Is*. *0* ≤ *I i* **using** *nonneg* **unfolding** *I-def* **by** *auto*
show ($\sum$ *i* ∈ *Is*. *Le-Constraint Leq-Rel* (*I i* ∗*R lec-poly* (*c i*)) (*I i* ∗*R lec-const* (*c i*))) =
Le-Constraint Leq-Rel 0 const **unfolding** *sum*[*symmetric*]
unfolding *sum.image-gen*[*OF* ‹*finite Is*›, *of - c*]
proof (*rule sum.cong*[*OF refl*], *goal-cases*)
case (*1 cc*)
define *II* **where** *II* = (*Is* ∩ {*j*. *cc* = *c j*})
from *1* **have** *II* ≠ {} **unfolding** *II-def* **by** *auto*
moreover have *finII*: *finite II* **using** ‹*finite Is*› **unfolding** *II-def* **by** *auto*
ultimately have *card*: *card II* ≠ *0* **by** *auto*
let *?C* = λ *II*. *rat-of-nat* (*card II*)
define *ii* **where** *ii* = *C cc* / *rat-of-nat* (*card II*)
have ($\sum$ *i*∈{*x* ∈ *Is*. *c x* = *cc*}. *Le-Constraint Leq-Rel* (*I i* ∗*R lec-poly* (*c i*)) (*I i* ∗*R lec-const* (*c i*)))
= ($\sum$ *i*∈ *II*. *Le-Constraint Leq-Rel* (*ii* ∗*R lec-poly cc*) (*ii* ∗*R lec-const cc*))
unfolding *I-def ii-def II-def* **by** (*rule sum.cong*, *auto*)
also have . . . = *Le-Constraint Leq-Rel* ((*?C II* ∗ *ii*) ∗*R lec-poly cc*) ((*?C II* ∗ *ii*) ∗*R lec-const cc*)
using *finII* **by** (*induct II rule*: *finite-induct*, *auto simp*: *zero-le-constraint-def field-simps*
scaleRat-left-distrib)
also have *?C II* ∗ *ii* = *C cc* **unfolding** *ii-def* **using** *card* **by** *auto*
finally show *?case* **.**
qed
qed
qed
finally show *?thesis* **.**
qed

end

3.4 Farkas Lemma for Matrices

In this part we convert the simplex-structures like linear polynomials, etc., into equivalent formulations using matrices and vectors. As a result we present Farkas' Lemma via matrices and vectors.

theory *Matrix-Farkas*
imports *Farkas*
Jordan-Normal-Form.Matrix

begin

lift-definition *poly-of-vec* :: *rat vec* ⇒ *linear-poly* **is**
λ v x. if (*x* < *dim-vec v*) *then v* $ *x else 0*
by *auto*

definition *val-of-vec* :: *rat vec* ⇒ *rat valuation* **where**
val-of-vec v x = *v* $ *x*

lemma *valuate-poly-of-vec*: **assumes** *w* ∈ *carrier-vec n*
and *v* ∈ *carrier-vec n*
shows *valuate* (*poly-of-vec v*) (*val-of-vec w*) = *v* · *w*
using *assms* **by** (*transfer*, *auto simp*: *val-of-vec-def scalar-prod-def intro*: *sum.mono-neutral-left*)

definition *constraints-of-mat-vec* :: *rat mat* ⇒ *rat vec* ⇒ *rat le-constraint set*
where
constraints-of-mat-vec A b = (*λ i* . *Leqc* (*poly-of-vec* (*row A i*)) (*b* $ *i*)) ' {*0* ..<
dim-row A}

lemma *constraints-of-mat-vec-solution-main*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
and *x*: *x* ∈ *carrier-vec nc*
and *b*: *b* ∈ *carrier-vec nr*
and *sol*: $A *_v x \leq b$
and *c*: *c* ∈ *constraints-of-mat-vec A b*
shows *val-of-vec x* $\models_{le}$ *c*
proof −
from *c*[*unfolded constraints-of-mat-vec-def*] *A* **obtain** *i* **where**
i: *i* < *nr* **and** *c*: *c* = *Leqc* (*poly-of-vec* (*row A i*)) (*b* $ *i*) **by** *auto*
from *i A* **have** *ri*: *row A i* ∈ *carrier-vec nc* **by** *auto*
from *sol i A x b* **have** *sol*: ($A *_v x$) $ *i* ≤ *b* $ *i* **unfolding** *less-eq-vec-def* **by**
auto
thus *val-of-vec x* $\models_{le}$ *c* **unfolding** *c satisfiable-le-constraint.simps rel-of.simps*
valuate-poly-of-vec[*OF x ri*] **using** *A x i* **by** *auto*
qed

lemma *vars-poly-of-vec*: *vars* (*poly-of-vec v*) ⊆ { *0* ..< *dim-vec v*}
by (*transfer'*, *auto*)

lemma *finite-constraints-of-mat-vec*: *finite* (*constraints-of-mat-vec A b*)
unfolding *constraints-of-mat-vec-def* **by** *auto*

lemma *lec-rec-constraints-of-mat-vec*: *lec-rel* ' *constraints-of-mat-vec A b* ⊆ {*Leq-Rel*}

unfolding *constraints-of-mat-vec-def* **by** *auto*

lemma *constraints-of-mat-vec-solution-1*:
assumes *A*: *A* ∈ *carrier-mat nr nc*
and *b*: *b* ∈ *carrier-vec nr*
and *sol*: ∃ *x* ∈ *carrier-vec nc*. $A *_v x \leq b$

shows ∃ *v*. ∀ *c* ∈ *constraints-of-mat-vec A b*. $v \models_{le} c$
using *constraints-of-mat-vec-solution-main*[*OF A - b -*] *sol* **by** *blast*

lemma *constraints-of-mat-vec-solution-2*:
assumes *A*: *A* ∈ *carrier-mat nr nc*
and *b*: *b* ∈ *carrier-vec nr*
and *sol*: ∃ *v*. ∀ *c* ∈ *constraints-of-mat-vec A b*. $v \models_{le} c$
shows ∃ *x* ∈ *carrier-vec nc*. $A *_v x \leq b$
proof −
from *sol* **obtain** *v* **where** *sol*: $v \models_{le} c$ **if** *c* ∈ *constraints-of-mat-vec A b* **for** *c*
by *auto*
define *x* **where** *x* = *vec nc* (λ *i*. *v i*)
show *?thesis*
proof (*intro bexI*[*of - x*])
show *x*: *x* ∈ *carrier-vec nc* **unfolding** *x-def* **by** *auto*
have *row A i* · *x* ≤ *b* $ *i* **if** *i* < *nr* **for** *i*
proof −
from *that* **have** *Leqc* (*poly-of-vec* (*row A i*)) (*b* $ *i*) ∈ *constraints-of-mat-vec A b*
unfolding *constraints-of-mat-vec-def* **using** *A* **by** *auto*
from *sol*[*OF this*, *simplified*] **have** *valuate* (*poly-of-vec* (*row A i*)) *v* ≤ *b* $ *i*
by *simp*
also have *valuate* (*poly-of-vec* (*row A i*)) *v* = *valuate* (*poly-of-vec* (*row A i*)) (*val-of-vec x*)
by (*rule valuate-depend*, *insert A that*,
auto simp: *x-def val-of-vec-def dest*!: *set-mp*[*OF vars-poly-of-vec*])
also have . . . = *row A i* · *x*
by (*subst valuate-poly-of-vec*[*OF x*], *insert that A x*, *auto*)
finally show *?thesis* **.**
qed
thus $A *_v x \leq b$ **unfolding** *less-eq-vec-def* **using** *x A b* **by** *auto*
qed
qed

lemma *constraints-of-mat-vec-solution*:
assumes *A*: *A* ∈ *carrier-mat nr nc*
and *b*: *b* ∈ *carrier-vec nr*
shows (∃ *x* ∈ *carrier-vec nc*. $A *_v x \leq b$) =
(∃ *v*. ∀ *c* ∈ *constraints-of-mat-vec A b*. $v \models_{le} c$)
using *constraints-of-mat-vec-solution-1*[*OF assms*] *constraints-of-mat-vec-solution-2*[*OF assms*]
by *blast*

lemma *farkas-lemma-matrix*: **fixes** *A* :: *rat mat*
assumes *A*: *A* ∈ *carrier-mat nr nc*
and *b*: *b* ∈ *carrier-vec nr*
shows (∃ *x* ∈ *carrier-vec nc*. $A *_v x \leq b$) ⟷
(∀ *y*. $y \geq 0_v$ *nr* ⟶ *mat-of-row y* ∗ *A* = 0_m *1 nc* ⟶ *y* · *b* ≥ *0*)
proof −

define *cs* **where** *cs* = *constraints-of-mat-vec A b*
have *fin*: *finite* {*0* ..< *nr*} **by** *auto*
have *dim*: *dim-row A* = *nr* **using** *A* **by** *simp*
have *sum-id*: ($\sum$ *i* = *0*..<*nr*. *f i*) = *sum-list* (*map f* [*0*..<*nr*]) **for** *f*
by (*subst sum-list-distinct-conv-sum-set*, *auto*)
have (∃ *x* ∈ *carrier-vec nc*. *A* $*_v$ *x* ≤ *b*) =
(¬ (∄ *v*. ∀ *c* ∈ *cs*. *v* $\models_{le}$ *c*))
unfolding *constraints-of-mat-vec-solution*[*OF assms*] *cs-def* **by** *simp*
also have ... = (¬ (∄ *v*. ∀ *i*∈{*0*..<*nr*}. *v* $\models_{le}$ *Le-Constraint Leq-Rel* (*poly-of-vec* (*row A i*)) (*b* $ *i*)))
unfolding *cs-def constraints-of-mat-vec-def dim* **by** *auto*
also have ... = (∄ *C*.
(∀ *i*∈{*0*..<*nr*}. *0* ≤ *C i*) ∧
($\sum$ *i* = *0*..<*nr*. (*C i* ∗*R poly-of-vec* (*row A i*))) = *0* ∧
($\sum$ *i* = *0*..<*nr*. (*C i* ∗ *b* $ *i*)) < *0*)
unfolding *Farkas'-Lemma-indexed*[*OF*
lec-rec-constraints-of-mat-vec[*unfolded constraints-of-mat-vec-def*], *of A b*,
unfolded dim, *OF fin*] *sum-id sum-list-lec le-constraint.simps*
sum-list-Leq-Rel map-map o-def **unfolding** *sum-id*[*symmetric*] **by** *simp*
also have ... = (∀ *C*. (∀ *i*∈ {*0*..<*nr*}. *0* ≤ *C i*) ⟶
($\sum$ *i* = *0*..<*nr*. (*C i* ∗*R poly-of-vec* (*row A i*))) = *0* ⟶
($\sum$ *i* = *0*..<*nr*. (*C i* ∗ *b* $ *i*)) ≥ *0*)
using *not-less* **by** *blast*
also have ... = (∀ *y*. *y* ≥ 0_v *nr* ⟶ *mat-of-row y* ∗ *A* = 0_m *1 nc* ⟶ *y* • *b* ≥ *0*)
proof ((*standard*; *intro allI impI*), *goal-cases*)
case ∗: (*1 y*)
define *C* **where** *C* = (λ *i*. *y* $ *i*)
note *main* = ∗(*1*)[*rule-format*, *of C*]
from ∗(*2*) **have** *y*: *y* ∈ *carrier-vec nr* **and** *nonneg*: ⋀*i*. *i* ∈ {*0*..<*nr*} ⟹ *0* ≤ *C i*
unfolding *less-eq-vec-def C-def* **by** *auto*
have *sum-0*: ($\sum$ *i* = *0*..<*nr*. *C i* ∗*R poly-of-vec* (*row A i*)) = *0* **unfolding** *C-def*
unfolding *zero-coeff-zero coeff-sum*
proof
fix *v*
have ($\sum$ *i* = *0*..<*nr*. *coeff* (*y* $ *i* ∗*R poly-of-vec* (*row A i*)) *v*) =
($\sum$ *i* < *nr*. *y* $ *i* ∗ *coeff* (*poly-of-vec* (*row A i*)) *v*) **by** (*rule sum.cong*, *auto*)
also have ... = *0*
proof (*cases v* < *nc*)
case *False*
have ($\sum$ *i* < *nr*. *y* $ *i* ∗ *coeff* (*poly-of-vec* (*row A i*)) *v*) =
($\sum$ *i* < *nr*. *y* $ *i* ∗ *0*)
by (*rule sum.cong*[*OF refl*], *rule arg-cong*[*of - - λ x. - ∗ x*], *insert A False*, *transfer*, *auto*)
also have ... = *0* **by** *simp*
finally show *?thesis* **by** *simp*

```
      next
        case True
        have (∑ i<nr. y $ i ∗ coeff (poly-of-vec (row A i)) v) =
              (∑ i<nr. y $ i ∗ row A i $ v)
          by (rule sum.cong[OF refl], rule arg-cong[of - - λ x. - ∗ x], insert A True,
transfer, auto)
        also have ... = (mat-of-row y ∗ A) $$ (0,v)
          unfolding times-mat-def scalar-prod-def
          using A y True by (auto intro: sum.cong)
        also have ... = 0 unfolding ∗(3) using True by simp
        finally show ?thesis .
      qed
      finally show (∑ i = 0..<nr. coeff (y $ i ∗R poly-of-vec (row A i)) v) = 0 .
    qed
    from main[OF nonneg sum-0] have le: 0 ≤ (∑ i = 0..<nr. C i ∗ b $ i) .
    thus ?case using y b unfolding scalar-prod-def C-def by auto
  next
    case ∗: (2 C)
    define y where y = vec nr C
    have y: y ∈ carrier-vec nr unfolding y-def by auto
    note main = ∗(1)[rule-format, of y]
    from ∗(2) have y0: y ≥ 0_v nr unfolding less-eq-vec-def y-def by auto
    have prod0: mat-of-row y ∗ A = 0_m 1 nc
    proof −
      {
        fix j
        assume j: j < nc
        from arg-cong[OF ∗(3), of λ x. coeff x j, unfolded coeff-sum]
        have 0 = (∑ i = 0..<nr. C i ∗ coeff (poly-of-vec (row A i)) j) by simp
        also have ... = (∑ i = 0..<nr. C i ∗ row A i $ j)
          by (rule sum.cong[OF refl], rule arg-cong[of - - λ x. - ∗ x], insert A j,
transfer, auto)
        also have ... = y · col A j unfolding scalar-prod-def y-def using A j
          by (intro sum.cong, auto)
        finally have y · col A j = 0 by simp
      }
      thus ?thesis by (intro eq-matI, insert A y, auto)
    qed
    from main[OF y0 prod0] have 0 ≤ y · b .
    thus ?case unfolding scalar-prod-def y-def using b by auto
  qed
  finally show ?thesis .
qed

lemma farkas-lemma-matrix': fixes A :: rat mat
  assumes A: A ∈ carrier-mat nr nc
  and b: b ∈ carrier-vec nr
shows (∃ x ≥ 0_v nc. A ∗_v x = b) ⟷
  (∀ y ∈ carrier-vec nr. mat-of-row y ∗ A ≥ 0_m 1 nc ⟶ y · b ≥ 0)
```

proof −
define B **where** $B = (-\ 1_m\ nc)\ @_r\ (A\ @_r\ -A)$
define b' **where** $b' = 0_v\ nc\ @_v\ (b\ @_v\ -b)$
define n **where** $n = nc + (nr + nr)$
have *id0*: $0_v\ (nc + (nr + nr)) = 0_v\ nc\ @_v\ (0_v\ nr\ @_v\ 0_v\ nr)$ **by** (*intro eq-vecI*, *auto*)
have B: $B \in$ *carrier-mat n nc* **unfolding** *B-def n-def* **using** A **by** *auto*
have b': $b' \in$ *carrier-vec n* **unfolding** *b′-def n-def* **using** b **by** *auto*
have $(\exists\ x \geq 0_v\ nc.\ A *_v x = b) = (\exists\ x.\ x \in \textit{carrier-vec}\ nc \wedge x \geq 0_v\ nc \wedge A *_v x = b)$
by (*rule arg-cong*[*of - - Ex*], *intro ext*, *insert A b*, *auto simp*: *less-eq-vec-def*)
also have $\ldots = (\exists\ x \in \textit{carrier-vec}\ nc.\ x \geq 0_v\ nc \wedge A *_v x = b)$ **by** *blast*
also have $\ldots = (\exists\ x \in \textit{carrier-vec}\ nc.\ 1_m\ nc *_v x \geq 0_v\ nc \wedge A *_v x \leq b \wedge A *_v x \geq b)$
by (*rule bex-cong*[*OF refl*], *insert A b*, *auto*)
also have $\ldots = (\exists\ x \in \textit{carrier-vec}\ nc.\ (-\ 1_m\ nc) *_v x \leq 0_v\ nc \wedge A *_v x \leq b \wedge (-\ A) *_v x \leq -b)$
by (*rule bex-cong*[*OF refl*], *insert A b*, *auto simp*: *less-eq-vec-def*)
also have $\ldots = (\exists\ x \in \textit{carrier-vec}\ nc.\ B *_v x \leq b')$
by (*rule bex-cong*[*OF refl*], *insert A b*, *unfold B-def b′-def*,
subst append-rows-le[*of -*], (*auto*)[*4*], *intro conj-cong*[*OF refl*], *subst append-rows-le*, *auto*)
also have $\ldots = (\forall\, y \geq 0_v\ n.\ \textit{mat-of-row}\ y * B = 0_m\ 1\ nc \longrightarrow y \cdot b' \geq 0)$
by (*rule farkas-lemma-matrix*[*OF B b′*])
also have $\ldots = (\forall\ y.\ y \in \textit{carrier-vec}\ n \longrightarrow y \geq 0_v\ n \longrightarrow \textit{mat-of-row}\ y * B = 0_m\ 1\ nc \longrightarrow y \cdot b' \geq 0)$
by (*intro arg-cong*[*of - - All*], *intro ext*, *auto simp*: *less-eq-vec-def*)
also have $\ldots = (\forall\ y \in \textit{carrier-vec}\ n.\ y \geq 0_v\ n \longrightarrow \textit{mat-of-row}\ y * B = 0_m\ 1\ nc \longrightarrow y \cdot b' \geq 0)$
by *blast*
also have $\ldots = (\forall\, y1 \in \textit{carrier-vec}\ nc.\ \forall\, y2 \in \textit{carrier-vec}\ nr.\ \forall\, y3 \in \textit{carrier-vec}\ nr.$
$0_v\ nc\ @_v\ (0_v\ nr\ @_v\ 0_v\ nr) \leq y1\ @_v\ y2\ @_v\ y3 \longrightarrow$
$\textit{mat-of-row}\ (y1\ @_v\ y2\ @_v\ y3) * ((-\ 1_m\ nc)\ @_r\ (A\ @_r\ -A)) = 0_m\ 1\ nc$
$\longrightarrow 0 \leq (y1\ @_v\ y2\ @_v\ y3) \cdot (0_v\ nc\ @_v\ (b\ @_v\ -b)))$
unfolding *n-def all-vec-append id0 b′-def B-def* **by** *auto*
also have $\ldots = (\forall\, y1 \in \textit{carrier-vec}\ nc.\ \forall\, y2 \in \textit{carrier-vec}\ nr.\ \forall\, y3 \in \textit{carrier-vec}\ nr.$
$0_v\ nc \leq y1 \longrightarrow 0_v\ nr \leq y2 \longrightarrow 0_v\ nr \leq y3 \longrightarrow$
$(-\ \textit{mat-of-row}\ y1) +$
$(\textit{mat-of-row}\ y2 * A - (\textit{mat-of-row}\ y3 * A)) = 0_m\ 1\ nc$
$\longrightarrow y2 \cdot b - y3 \cdot b \geq 0)$
by (*intro ball-cong*[*OF refl*], *subst append-vec-le*, (*auto*)[*2*], *subst append-vec-le*, (*auto*)[*2*], *insert A b*,
subst scalar-prod-append, (*auto*)[*4*], *subst scalar-prod-append*, (*auto*)[*4*],
subst mat-of-row-mult-append-rows, (*auto*)[*4*],
subst mat-of-row-mult-append-rows, (*auto*)[*4*],
subst add-uminus-minus-mat[*symmetric*], *auto*)

also have ... = (∀ *y1* ∈*carrier-vec nc*. ∀ *y2* ∈*carrier-vec nr*. ∀ *y3* ∈*carrier-vec nr*.
0_v *nc* ≤ *y1* ⟶ 0_v *nr* ≤ *y2* ⟶ 0_v *nr* ≤ *y3* ⟶
mat-of-row y1 = *mat-of-row y2* ∗ *A* − *mat-of-row y3* ∗ *A*
⟶ *y2* · *b* − *y3* · *b* ≥ *0*)
proof ((*intro ball-cong*[*OF refl*] *arg-cong2*[*of* - - - - (⟶)] *refl*, *standard*), *goal-cases*)
case (*1 y1 y2 y3*)
from *arg-cong*[*OF 1*(*4*), *of* λ *x*. *mat-of-row y1* + *x*] **show** *?case* **using** *1*(*1*−*3*) *A*
by (*subst* (*asm*) *assoc-add-mat*[*symmetric*], (*auto*)[*3*],
subst (*asm*) *add-uminus-minus-mat*, (*auto*)[*1*],
subst (*asm*) *minus-r-inv-mat*, *force*,
subst (*asm*) *right-add-zero-mat*, *force*,
subst (*asm*) *left-add-zero-mat*, *force*, *auto*)
next
case (*2 y1 y2 y3*)
show *?case* **unfolding** *2*(*4*) **using** *2*(*1*−*3*) *A*
by (*intro eq-matI*, *auto*)
qed
also have ... = (∀ *y1* ∈*carrier-vec nc*. ∀ *y2* ∈*carrier-vec nr*. ∀ *y3* ∈*carrier-vec nr*.
0_v *nc* ≤ *y1* ⟶ 0_v *nr* ≤ *y2* ⟶ 0_v *nr* ≤ *y3* ⟶
mat-of-row y1 = *mat-of-row* (*y2* − *y3*) ∗ *A*
⟶ (*y2* − *y3*) · *b* ≥ *0*)
by (*intro ball-cong*[*OF refl*] *imp-cong refl*
arg-cong2[*of* - - - - (≤)] *arg-cong2*[*of* - - - - (=)],
subst minus-mult-distrib-mat[*symmetric*], *insert A b*, *auto*
simp: *minus-scalar-prod-distrib mat-of-rows-def*
intro!: *arg-cong*[*of* - - λ *x*. *x* ∗ -])
also have ... = (∀ *y1* ∈*carrier-vec nc*. ∀ *y2* ∈*carrier-vec nr*. ∀ *y3* ∈*carrier-vec nr*.
0_v *nc* ≤ *y1* ⟶ 0_v *nr* ≤ *y2* ⟶ 0_v *nr* ≤ *y3* ⟶
y1 = *row* (*mat-of-row* (*y2* − *y3*) ∗ *A*) *0*
⟶ (*y2* − *y3*) · *b* ≥ *0*)
proof (*intro ball-cong*[*OF refl*] *arg-cong2*[*of* - - - - (⟶)] *refl*, *standard*, *goal-cases*)
case (*1 y1 y2 y3*)
from *arg-cong*[*OF 1*(*4*), *of* λ *x*. *row x 0*] *1*(*1*−*3*) *A*
show *?case* **by** *auto*
qed (*insert A*, *auto*)
also have ... = (∀ *y2* ∈*carrier-vec nr*. ∀ *y3* ∈*carrier-vec nr*.
0_v *nc* ≤ *row* (*mat-of-row* (*y2* − *y3*) ∗ *A*) *0* ⟶ 0_v *nr* ≤ *y2* ⟶ 0_v *nr* ≤ *y3* ⟶
row (*mat-of-row* (*y2* − *y3*) ∗ *A*) *0* ∈ *carrier-vec nc*
⟶ (*y2* − *y3*) · *b* ≥ *0*) **by** *blast*
also have ... = (∀ *y2* ∈*carrier-vec nr*. ∀ *y3* ∈*carrier-vec nr*.
0_v *nc* ≤ *row* (*mat-of-row* (*y2* − *y3*) ∗ *A*) *0* ⟶ 0_v *nr* ≤ *y2* ⟶ 0_v *nr* ≤ *y3*
⟶ (*y2* − *y3*) · *b* ≥ *0*)
by (*intro ball-cong*[*OF refl*] *arg-cong2*[*of* - - - - (⟶)] *refl*, *insert A*,

auto simp: *row-def*)
also have ... = (∀ *y* ∈ *carrier-vec nr*. *row* (*mat-of-row y* ∗ *A*) *0* ≥ 0_v *nc* ⟶ *y* · *b* ≥ *0*)
proof ((*standard*; *intro ballI impI*), *goal-cases*)
case (*1 y*)
define *y2* **where** *y2* = *vec nr* (λ *i*. *if y* $ *i* ≥ *0 then y* $ *i else 0*)
define *y3* **where** *y3* = *vec nr* (λ *i*. *if y* $ *i* ≥ *0 then 0 else* − *y* $ *i*)
have *y*: *y* = *y2* − *y3* **unfolding** *y2-def y3-def* **using** *1*(*2*)
by (*intro eq-vecI*, *auto*)
show *?case* **by** (*rule 1*(*1*)[*rule-format*, *of y2 y3*, *folded y*, *OF - - 1*(*3*)],
auto simp: *y2-def y3-def less-eq-vec-def*)
qed *auto*
also have ... = (∀ *y* ∈ *carrier-vec nr*. *mat-of-row y* ∗ *A* ≥ 0_m *1 nc* ⟶ *y* · *b* ≥ *0*)
by (*intro ball-cong arg-cong2*[*of - - - -* (⟶)] *refl*,
insert A, *auto simp*: *less-eq-vec-def less-eq-mat-def*)
finally show *?thesis* **.**
qed

end

4 Unsatisfiability over the Reals

By using Farkas' Lemma we prove that a finite set of linear rational inequalities is satisfiable over the rational numbers if and only if it is satisfiable over the real numbers. Hence, the simplex algorithm either gives a rational solution or shows unsatisfiability over the real numbers.

theory *Simplex-for-Reals*
imports
Farkas
Simplex.Simplex-Incremental
begin

instantiation *real* :: *lrv*
begin
definition *scaleRat-real* :: *rat* ⇒ *real* ⇒ *real* **where**
[*simp*]: *x* ∗*R y* = *real-of-rat x* ∗ *y*
instance by *standard* (*auto simp add*: *field-simps of-rat-mult of-rat-add*)
end

abbreviation *real-satisfies-constraints* :: *real valuation* ⇒ *constraint set* ⇒ *bool*
(**infixl** ‹$\models_{rcs}$› *100*) **where**
v $\models_{rcs}$ *cs* ≡ ∀ *c* ∈ *cs*. *v* $\models_c$ *c*

definition *of-rat-val* :: *rat valuation* ⇒ *real valuation* **where**
of-rat-val v x = *of-rat* (*v x*)

lemma *of-rat-val-eval*: $p \{\!| \textit{of-rat-val } v |\!\} = \textit{of-rat } (p \{\!| v |\!\})$
  unfolding *of-rat-val-def linear-poly-sum of-rat-sum*
  by (*rule sum.cong*, *auto simp*: *of-rat-mult*)

lemma *of-rat-val-constraint*: $\textit{of-rat-val } v \models_c c \longleftrightarrow v \models_c c$
  by (*cases c*, *auto simp*: *of-rat-val-eval of-rat-less of-rat-less-eq*)

lemma *of-rat-val-constraints*: $\textit{of-rat-val } v \models_{rcs} cs \longleftrightarrow v \models_{cs} cs$
  using *of-rat-val-constraint* **by** *auto*

lemma *sat-scale-rat-real*: **assumes** $(v :: \textit{real valuation}) \models_c c$
  shows $v \models_c (r *R\ c)$
proof −
  have $r < 0 \lor r = 0 \lor r > 0$ **by** *auto*
  then show *?thesis* **using** *assms* **by** (*cases c*, *simp-all add*: *right-diff-distrib*
    valuate-minus valuate-scaleRat scaleRat-leq1 scaleRat-leq2 valuate-zero
    of-rat-less of-rat-mult)
qed

fun *of-rat-lec* :: *rat le-constraint* ⇒ *real le-constraint* **where**
  of-rat-lec (*Le-Constraint r p c*) = *Le-Constraint r p* (*of-rat c*)

lemma *lec-of-constraint-real*:
  assumes *is-le c*
  shows $(v \models_{le} \textit{of-rat-lec } (\textit{lec-of-constraint } c)) \longleftrightarrow (v \models_c c)$
  using *assms* **by** (*cases c*, *auto*)

lemma *of-rat-lec-add*: *of-rat-lec* (*c* + *d*) = *of-rat-lec c* + *of-rat-lec d*
  by (*cases c*; *cases d*, *auto simp*: *of-rat-add*)

lemma *of-rat-lec-zero*: *of-rat-lec 0* = *0*
  unfolding *zero-le-constraint-def* **by** *simp*

lemma *of-rat-lec-sum*: *of-rat-lec* (*sum-list c*) = *sum-list* (*map of-rat-lec c*)
  by (*induct c*, *auto simp*: *of-rat-lec-zero of-rat-lec-add*)

This is the main lemma: a finite set of linear constraints is satisfiable over Q if and only if it is satisfiable over R.

lemma *rat-real-conversion*: **assumes** *finite cs*
  shows $(\exists\ v :: \textit{rat valuation}.\ v \models_{cs} cs) \longleftrightarrow (\exists\ v :: \textit{real valuation}.\ v \models_{rcs} cs)$
proof
  show $\exists v.\ v \models_{cs} cs \Longrightarrow \exists v.\ v \models_{rcs} cs$ **using** *of-rat-val-constraint* **by** *auto*
  assume $\exists v.\ v \models_{rcs} cs$
  then obtain *v* **where** ∗: $v \models_{rcs} cs$ **by** *auto*
  show $\exists v.\ v \models_{cs} cs$
  proof (*rule ccontr*)
    assume $\nexists v.\ v \models_{cs} cs$
    from *farkas-coefficients*[*OF assms*] *this*
    obtain *C* **where** *farkas-coefficients cs C* **by** *auto*

from *this*[*unfolded farkas-coefficients-def*]
obtain *d rel* **where**
isleq: $(\forall (r,c) \in$ *set C*. $c \in cs \wedge$ *is-le* $(r *R\ c) \wedge r \neq 0)$ **and**
leq: $(\sum (r,c) \leftarrow C.$ *lec-of-constraint* $(r *R\ c))$ = *Le-Constraint rel 0 d* **and**
choice: *rel* = *Lt-Rel* $\wedge\ d \leq 0 \vee$ *rel* = *Leq-Rel* $\wedge\ d < 0$ **by** *blast*
{
fix *r c*
assume *c*: $(r,c) \in$ *set C*
from $c *$ *isleq* **have** $v \models_c c$ **by** *auto*
hence *v*: $v \models_c (r *R\ c)$ **by** (*rule sat-scale-rat-real*)
from *c isleq* **have** *is-le* $(r *R\ c)$ **by** *auto*
from *lec-of-constraint-real*[*OF this*] *v*
have $v \models_{le}$ *of-rat-lec* (*lec-of-constraint* $(r *R\ c)$) **by** *blast*
} **note** *v* = *this*
have *Le-Constraint rel 0* (*of-rat d*) = *of-rat-lec* $(\sum (r,c) \leftarrow C.$ *lec-of-constraint* $(r *R\ c))$
unfolding *leq* **by** *simp*
also have ... = $(\sum (r,c) \leftarrow C.$ *of-rat-lec* (*lec-of-constraint* $(r *R\ c)$)) (**is** - = *?sum*)
unfolding *of-rat-lec-sum map-map o-def* **by** (*rule arg-cong*[*of - - sum-list*], *auto*)
finally have *leq*: *Le-Constraint rel 0* (*of-rat d*) = *?sum* **by** *simp*
have $v \models_{le}$ *Le-Constraint rel 0* (*of-rat d*) **unfolding** *leq*
by (*rule satisfies-sumlist-le-constraints*, *insert v*, *auto*)
with *choice* **show** *False* **by** (*auto simp*: *linear-poly-sum*)
qed
qed

The main result of simplex, now using unsatisfiability over the reals.

fun *i-satisfies-cs-real* (**infixl** $\langle\models_{rics}\rangle$ *100*) **where**
$(I,v) \models_{rics} cs \longleftrightarrow v \models_{rcs}$ *Simplex.restrict-to I cs*

lemma *simplex-index-real*:
simplex-index cs = *Unsat I* $\Longrightarrow$ *set I* $\subseteq$ *fst* ‘ *set cs* $\wedge \neg (\exists\ v.$ (*set I*, *v*) $\models_{rics}$ *set cs*) $\wedge$
(*distinct-indices cs* $\longrightarrow (\forall\ J \subset$ *set I*. $(\exists\ v.\ (J,\ v) \models_{ics}$ *set cs*))) — minimal unsat core over the reals
simplex-index cs = *Sat v* $\Longrightarrow \langle v\rangle \models_{cs}$ (*snd* ‘ *set cs*) — satisfying assingment
using *simplex-index*(*1*)[*of cs I*] *simplex-index*(*2*)[*of cs v*]
rat-real-conversion[*of Simplex.restrict-to* (*set I*) (*set cs*)]
by *auto*

lemma *simplex-real*:
simplex cs = *Unsat I* $\Longrightarrow \neg (\exists\ v.\ v \models_{rcs}$ *set cs*) — unsat of original constraints over the reals
simplex cs = *Unsat I* $\Longrightarrow$ *set I* $\subseteq \{0..<$*length cs*$\} \wedge \neg (\exists\ v.\ v \models_{rcs} \{cs\ !\ i \mid i.\ i \in$ *set I*$\})$
$\wedge\ (\forall\ J \subset$ *set I*. $\exists\ v.\ v \models_{cs} \{cs\ !\ i \mid i.\ i \in J\})$ — minimal unsat core over reals

simplex cs = *Sat v* $\Longrightarrow$ $\langle v \rangle \models_{cs}$ *set cs* — satisfying assignment over the rationals
proof (*intro simplex*(*1*)[*unfolded rat-real-conversion*[*OF finite-set*]])
assume *unsat*: *simplex cs* = *Inl I*
have *finite* $\{cs\ !\ i\ |i.\ i \in set\ I\}$ **by** *auto*
from *simplex*(*2*)[*OF unsat*, *unfolded rat-real-conversion*[*OF this*]]
show $set\ I \subseteq \{0..<length\ cs\} \wedge \neg\ (\exists\ v.\ v \models_{rcs} \{cs\ !\ i\ |\ i.\ i \in set\ I\})$
$\wedge\ (\forall\ J \subset set\ I.\ \exists\ v.\ v \models_{cs} \{cs\ !\ i\ |i.\ i \in J\})$ **by** *auto*
qed (*insert simplex*(*3*), *auto*)

Define notion of minimal unsat core over the reals: the subset has to be unsat over the reals, and every proper subset has to be satisfiable over the rational numbers.

definition *minimal-unsat-core-real* :: *'i set* ⇒ *'i i-constraint list* ⇒ *bool* **where**
minimal-unsat-core-real I ics $= ((I \subseteq fst\ `\ set\ ics) \wedge (\neg\ (\exists\ v.\ (I,v) \models_{rics}\ set\ ics))$
$\wedge\ (distinct\text{-}indices\ ics \longrightarrow (\forall\ J.\ J \subset I \longrightarrow (\exists\ v.\ (J,v) \models_{ics}\ set\ ics))))$

Because of equi-satisfiability the two notions of minimal unsat cores coincide.

lemma *minimal-unsat-core-real-conv*: *minimal-unsat-core-real I ics* = *minimal-unsat-core I ics*
proof
show *minimal-unsat-core-real I ics* $\Longrightarrow$ *minimal-unsat-core I ics*
unfolding *minimal-unsat-core-real-def minimal-unsat-core-def*
using *of-rat-val-constraint* **by** *simp metis*
next
assume *minimal-unsat-core I ics*
thus *minimal-unsat-core-real I ics*
unfolding *minimal-unsat-core-real-def minimal-unsat-core-def*
using *rat-real-conversion*[*of Simplex.restrict-to I* (*set ics*)]
by *auto*
qed

Easy consequence: The incremental simplex algorithm is also sound wrt. minimal-unsat-cores over the reals.

lemmas *incremental-simplex-real* =
init-simplex
assert-simplex-ok
assert-simplex-unsat[*folded minimal-unsat-core-real-conv*]
assert-all-simplex-ok
assert-all-simplex-unsat[*folded minimal-unsat-core-real-conv*]
check-simplex-ok
check-simplex-unsat[*folded minimal-unsat-core-real-conv*]
solution-simplex
backtrack-simplex
checked-invariant-simplex

end

References

[1] M. Bromberger and C. Weidenbach. New techniques for linear arithmetic: cubes and equalities. *Formal Methods in System Design*, 51(3):433–461, Dec 2017.

[2] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *CAV'06*, volume 4144 of *LNCS*, pages 81–94, 2006.

[3] F. Marić, M. Spasić, and R. Thiemann. An incremental simplex algorithm with unsatisfiable core generation. *Archive of Formal Proofs*, Aug. 2018. http://isa-afp.org/entries/Simplex.html, Formal proof development.

[4] A. Schrijver. *Theory of linear and integer programming.* Wiley, 1999.

[5] M. Spasić and F. Marić. Formalization of incremental simplex algorithm by stepwise refinement. In D. Giannakopoulou and D. Méry, editors, *FM'12*, volume 7436 of *LNCS*, pages 434–449, 2012.

[6] J. Stoer and C. Witzgall. *Convexity and Optimization in Finite Dimensions I.* Die Grundlehren der mathematischen Wissenschaften 163. Springer-Verlag Berlin Heidelberg, 1 edition, 1970.

[7] R. Thiemann. Extending a verified simplex algorithm. In G. Barthe, K. Korovin, S. Schulz, M. Suda, G. Sutcliffe, and M. Veanes, editors, *LPAR-22 Workshop and Short Paper Proceedings*, volume 9 of *Kalpa Publications in Computing*, pages 37–48. EasyChair, 2018.