An Incremental Simplex Algorithm with Unsatisfiable Core Generation*

Filip Marić   Mirko Spasić   René Thiemann

June 19, 2024

Abstract

We present an Isabelle/HOL formalization and total correctness proof for the incremental version of the Simplex algorithm which is used in most state-of-the-art SMT solvers. It supports extraction of satisfying assignments, extraction of minimal unsatisfiable cores, incremental assertion of constraints and backtracking. The formalization relies on stepwise program refinement, starting from a simple specification, going through a number of refinement steps, and ending up in a fully executable functional implementation. Symmetries present in the algorithm are handled with special care.

Contents

*Supported by the Serbian Ministry of Education and Science grant 174021, by the SNF grant SCOPES IZ73Z0127979/1, and by FWF (Austrian Science Fund) project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

1 Introduction

This formalization closely follows the simplex algorithm as it is described by Dutertre and de Moura [1].

The original formalization has been developed and is extensively described by Spasić and Marić [3]. It features a front-end that for a given set of constraints either returns a satisfying assignment or the information that it is unsatisfiable.

The original formalization was extended by Thiemann in three different ways.

- The extended simplex method returns a minimal unsatisfiable core instead of just a bit "unsatisfiable".
- The extension also contains an incremental interface to the simplex method where one can dynamically assert and retract linear constraints. In contrast, the original simplex formalization only offered an interface which demands all constraints as input and which restarts the computation from scratch on every input.
- The optimization of eliminating unused variables in the preprocessing phase [1, Section 3] has been integrated in the formalization.

The first two of these extensions required the introduction of *indexed* constraints in combination with generalised lemmas. In these generalisations, global constraints had to be replaced by arbitrary (indexed) subsets of constraints.

2 Auxiliary Results

theory *Simplex-Auxiliary*
imports
HOL−Library.Mapping
begin

lemma *map-reindex*:
assumes $\forall\ i < length\ l.\ g\ (l\ !\ i) = f\ i$
shows $map\ f\ [0..<length\ l] = map\ g\ l$
⟨*proof*⟩

lemma *map-parametrize-idx*:
$map\ f\ l = map\ (\lambda i.\ f\ (l\ !\ i))\ [0..<length\ l]$
⟨*proof*⟩

lemma *last-tl*:
assumes $length\ l > 1$
shows $last\ (tl\ l) = last\ l$
⟨*proof*⟩

lemma *hd-tl*:
assumes $length\ l > 1$
shows $hd\ (tl\ l) = l\ !\ 1$
⟨*proof*⟩

lemma *butlast-empty-conv-length*:
shows $(butlast\ l = []) = (length\ l \leq 1)$
⟨*proof*⟩

lemma *butlast-nth*:
assumes $n + 1 < length\ l$
shows $butlast\ l\ !\ n = l\ !\ n$
⟨*proof*⟩

lemma *last-take-conv-nth*:
assumes $0 < n\ n \leq length\ l$
shows $last\ (take\ n\ l) = l\ !\ (n - 1)$
⟨*proof*⟩

lemma *tl-nth*:
assumes $l \neq []$
shows $tl\ l\ !\ n = l\ !\ (n + 1)$
⟨*proof*⟩

lemma *interval-3split*:

assumes $i < n$
shows $[0..<n] = [0..<i]$ @ $[i]$ @ $[i+1..<n]$
$\langle proof \rangle$

abbreviation *list-min l* ≡ *foldl min* (*hd l*) (*tl l*)
lemma *list-min-Min*[*simp*]: $l \neq [] \Longrightarrow$ *list-min l* = *Min* (*set l*)
$\langle proof \rangle$

definition *min-satisfying* :: ((*'a*::*linorder*) ⇒ *bool*) ⇒ *'a list* ⇒ *'a option* **where**
min-satisfying P l ≡
let xs = *filter P l in*
if xs = [] *then None else Some* (*list-min xs*)

lemma *min-satisfying-None*:
min-satisfying P l = *None* ⟶
(∀ $x \in$ *set l*. ¬ $P\ x$)
$\langle proof \rangle$

lemma *min-satisfying-Some*:
min-satisfying P l = *Some x* ⟶
$x \in$ *set l* ∧ $P\ x$ ∧ (∀ $x' \in$ *set l*. $x' < x$ ⟶ ¬ $P\ x'$)
$\langle proof \rangle$

lemma *min-element*:
fixes *k* :: *nat*
assumes ∃ (*m*::*nat*). *P m*
shows ∃ *mm*. *P mm* ∧ (∀ *m'*. *m'* < *mm* ⟶ ¬ *P m'*)
$\langle proof \rangle$

lemma *finite-fun-args*:
assumes *finite A* ∀ $a \in A$. *finite* (*B a*)
shows *finite* {*f*. (∀ *a*. *if* $a \in A$ *then f a* ∈ *B a else f a* = *f0 a*)} (**is** *finite* (*?F A*))
$\langle proof \rangle$

lemma *foldl-mapping-update*:

assumes $X \in set\ l$ $distinct\ (map\ f\ l)$
shows $Mapping.lookup\ (foldl\ (\lambda m\ a.\ Mapping.update\ (f\ a)\ (g\ a)\ m)\ i\ l)\ (f\ X) = Some\ (g\ X)$
⟨*proof*⟩

end

theory *Rel-Chain*
imports
Simplex-Auxiliary
begin

definition
rel-chain :: $'a\ list \Rightarrow ('a \times 'a)\ set \Rightarrow bool$
where
rel-chain $l\ r = (\forall\ k < length\ l - 1.\ (l\ !\ k,\ l\ !\ (k + 1)) \in r)$

lemma
rel-chain-Nil: *rel-chain* [] r **and**
rel-chain-Cons: *rel-chain* $(x\ \#\ xs)\ r = (if\ xs = []\ then\ True\ else\ ((x,\ hd\ xs) \in r) \wedge rel\text{-}chain\ xs\ r)$
⟨*proof*⟩

lemma *rel-chain-drop*:
rel-chain $l\ R$ ==> *rel-chain* $(drop\ n\ l)\ R$
⟨*proof*⟩

lemma *rel-chain-take*:
rel-chain $l\ R$ ==> *rel-chain* $(take\ n\ l)\ R$
⟨*proof*⟩

lemma *rel-chain-butlast*:
rel-chain $l\ R$ ==> *rel-chain* $(butlast\ l)\ R$
⟨*proof*⟩

lemma *rel-chain-tl*:
rel-chain $l\ R$ ==> *rel-chain* $(tl\ l)\ R$
⟨*proof*⟩

lemma *rel-chain-append*:
assumes *rel-chain* $l\ R$ *rel-chain* $l'\ R$ $(last\ l,\ hd\ l') \in R$
shows *rel-chain* $(l\ @\ l')\ R$
⟨*proof*⟩

lemma *rel-chain-appendD*:
assumes *rel-chain* $(l\ @\ l')\ R$
shows *rel-chain* $l\ R$ *rel-chain* $l'\ R$ $l \neq [] \wedge l' \neq [] \longrightarrow (last\ l,\ hd\ l') \in R$
⟨*proof*⟩

lemma *rtrancl-rel-chain*:
$(x, y) \in R^* \longleftrightarrow (\exists\ l.\ l \neq [] \wedge hd\ l = x \wedge last\ l = y \wedge rel\text{-}chain\ l\ R)$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *trancl-rel-chain*:
$(x, y) \in R^+ \longleftrightarrow (\exists\ l.\ l \neq [] \wedge length\ l > 1 \wedge hd\ l = x \wedge last\ l = y \wedge rel\text{-}chain\ l\ R)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *rel-chain-elems-rtrancl*:
assumes $rel\text{-}chain\ l\ R\ i \leq j\ j < length\ l$
shows $(l\ !\ i, l\ !\ j) \in R^*$
⟨*proof*⟩

lemma *reorder-cyclic-list*:
assumes $hd\ l = s\ last\ l = s\ length\ l > 2\ sl + 1 < length\ l$
rel-chain l r
obtains $l' :: 'a\ list$
where $hd\ l' = l\ !\ (sl + 1)\ last\ l' = l\ !\ sl\ rel\text{-}chain\ l'\ r\ length\ l' = length\ l - 1$
$\forall\ i.\ i + 1 < length\ l' \longrightarrow$
$(\exists\ j.\ j + 1 < length\ l \wedge l'\ !\ i = l\ !\ j \wedge l'\ !\ (i + 1) = l\ !\ (j + 1))$
⟨*proof*⟩

end

3 Linearly Ordered Rational Vectors

theory *Simplex-Algebra*
imports
HOL.Rat
HOL.Real-Vector-Spaces
begin

class *scaleRat* =
fixes $scaleRat :: rat \Rightarrow 'a \Rightarrow 'a$ (**infixr** $*R\ 75$)
begin

abbreviation
$divideRat :: 'a \Rightarrow rat \Rightarrow 'a$ (**infixl** $'/R\ 70$)
where
$x\ /R\ r == scaleRat\ (inverse\ r)\ x$
end

class *rational-vector* = *scaleRat* + *ab-group-add* +
assumes *scaleRat-right-distrib*: $scaleRat\ a\ (x + y) = scaleRat\ a\ x + scaleRat\ a\ y$
and *scaleRat-left-distrib*: $scaleRat\ (a + b)\ x = scaleRat\ a\ x + scaleRat\ b\ x$
and *scaleRat-scaleRat*: $scaleRat\ a\ (scaleRat\ b\ x) = scaleRat\ (a * b)\ x$

and *scaleRat-one*: $scaleRat\ 1\ x = x$

interpretation *rational-vector*:
$vector\text{-}space\ scaleRat :: rat \Rightarrow 'a \Rightarrow 'a::rational\text{-}vector$
⟨*proof*⟩

class $ordered\text{-}rational\text{-}vector = rational\text{-}vector + order$

class $linordered\text{-}rational\text{-}vector = ordered\text{-}rational\text{-}vector + linorder +$
assumes *plus-less*: $(a::'a) < b \Longrightarrow a + c < b + c$ **and**
scaleRat-less1: $[\![(a::'a) < b;\ k > 0]\!] \Longrightarrow (k *R\ a) < (k *R\ b)$ **and**
scaleRat-less2: $[\![(a::'a) < b;\ k < 0]\!] \Longrightarrow (k *R\ a) > (k *R\ b)$
begin

lemma *scaleRat-leq1*: $[\![\ a \leq b;\ k > 0]\!] \Longrightarrow k *R\ a \leq k *R\ b$
⟨*proof*⟩

lemma *scaleRat-leq2*: $[\![\ a \leq b;\ k < 0]\!] \Longrightarrow k *R\ a \geq k *R\ b$
⟨*proof*⟩

lemma *zero-scaleRat*
[*simp*]: $0 *R\ v = zero$
⟨*proof*⟩

lemma *scaleRat-zero*
[*simp*]: $a *R\ (0::'a) = 0$
⟨*proof*⟩

lemma *scaleRat-uminus* [*simp*]:
$-1 *R\ x = -\ (x :: 'a)$
⟨*proof*⟩

lemma *minus-lt*: $(a::'a) < b \longleftrightarrow a - b < 0$
⟨*proof*⟩

lemma *minus-gt*: $(a::'a) < b \longleftrightarrow 0 < b - a$
⟨*proof*⟩

lemma *minus-leq*:
$(a::'a) \leq b \longleftrightarrow a - b \leq 0$
⟨*proof*⟩

lemma *minus-geq*: $(a::'a) \leq b \longleftrightarrow 0 \leq b - a$
⟨*proof*⟩

lemma *divide-lt*:
$[\![c *R\ (a::'a) < b;\ (c::rat) > 0\ ]\!] \Longrightarrow a < (1/c) *R\ b$
⟨*proof*⟩

lemma *divide-gt*:
  $[\![c *R (a::'a) > b;(c::rat) > 0]\!] \Longrightarrow a > (1/c) *R b$
  ⟨*proof*⟩

lemma *divide-leq*:
  $[\![c *R (a::'a) \leq b; (c::rat) > 0]\!] \Longrightarrow a \leq (1/c) *R b$
⟨*proof*⟩

lemma *divide-geq*:
  $[\![c *R (a::'a) \geq b; (c::rat) > 0]\!] \Longrightarrow a \geq (1/c) *R b$
⟨*proof*⟩

lemma *divide-lt1*:
  $[\![c *R (a::'a) < b; (c::rat) < 0]\!] \Longrightarrow a > (1/c) *R b$
  ⟨*proof*⟩

lemma *divide-gt1*:
  $[\![c *R (a::'a) > b; (c::rat) < 0]\!] \Longrightarrow a < (1/c) *R b$
  ⟨*proof*⟩

lemma *divide-leq1*:
  $[\![c *R (a::'a) \leq b; (c::rat) < 0]\!] \Longrightarrow a \geq (1/c) *R b$
⟨*proof*⟩

lemma *divide-geq1*:
  $[\![c *R (a::'a) \geq b; (c::rat) < 0]\!] \Longrightarrow a \leq (1/c) *R b$
⟨*proof*⟩

end

class *lrv* = *linordered-rational-vector* + *one* +
  assumes *zero-neq-one*: $0 \neq 1$

subclass (**in** *linordered-rational-vector*) *ordered-ab-semigroup-add*
⟨*proof*⟩

instantiation *rat* :: *rational-vector*
begin
definition *scaleRat-rat* :: *rat* ⇒ *rat* ⇒ *rat* **where**
  [*simp*]: $x *R y = x * y$
instance ⟨*proof*⟩
end

instantiation *rat* :: *ordered-rational-vector*
begin
instance ⟨*proof*⟩
end

instantiation *rat* :: *linordered-rational-vector*

begin
instance ⟨*proof*⟩
end

instantiation *rat* :: *lrv*
begin
instance ⟨*proof*⟩
end

lemma *uminus-less-lrv*[*simp*]: **fixes** *a b* :: *'a* :: *lrv*
shows − *a* < − *b* ⟷ *b* < *a*
⟨*proof*⟩

end

4 Linear Polynomials and Constraints

theory *Abstract-Linear-Poly*
imports
Simplex-Algebra
begin

type-synonym *var* = *nat*

(Infinite) linear polynomials as functions from vars to coeffs

definition *fun-zero* :: *var* ⇒ *'a*::*zero* **where**
[*simp*]: *fun-zero* == λ *v*. *0*
definition *fun-plus* :: (*var* ⇒ *'a*) ⇒ (*var* ⇒ *'a*) ⇒ *var* ⇒ *'a*::*plus* **where**
[*simp*]: *fun-plus f1 f2* == λ *v*. *f1 v* + *f2 v*
definition *fun-scale* :: *'a* ⇒ (*var* ⇒ *'a*) ⇒ (*var* ⇒ *'a*::*ring*) **where**
[*simp*]: *fun-scale c f* == λ *v*. *c*∗(*f v*)
definition *fun-coeff* :: (*var* ⇒ *'a*) ⇒ *var* ⇒ *'a* **where**
[*simp*]: *fun-coeff f var* = *f var*
definition *fun-vars* :: (*var* ⇒ *'a*::*zero*) ⇒ *var set* **where**
[*simp*]: *fun-vars f* = {*v*. *f v* ≠ *0*}
definition *fun-vars-list* :: (*var* ⇒ *'a*::*zero*) ⇒ *var list* **where**
[*simp*]: *fun-vars-list f* = *sorted-list-of-set* {*v*. *f v* ≠ *0*}
definition *fun-var* :: *var* ⇒ (*var* ⇒ *'a*::{*zero*,*one*}) **where**
[*simp*]: *fun-var x* = (λ *x'*. *if x'* = *x then 1 else 0*)
type-synonym *'a valuation* = *var* ⇒ *'a*
definition *fun-valuate* :: (*var* ⇒ *rat*) ⇒ *'a valuation* ⇒ (*'a*::*rational-vector*) **where**
[*simp*]: *fun-valuate lp val* = (∑ *x*∈{*v*. *lp v* ≠ *0*}. *lp x* ∗*R val x*)

Invariant – only finitely many variables

definition *inv* **where**
[*simp*]: *inv c* == *finite* {*v*. *c v* ≠ *0*}

lemma *inv-fun-zero* [*simp*]:
inv fun-zero ⟨*proof*⟩

lemma *inv-fun-plus* [*simp*]:
  $[\![inv\ (f1\ ::\ nat \Rightarrow\ 'a::monoid\text{-}add);\ inv\ f2]\!] \Longrightarrow inv\ (fun\text{-}plus\ f1\ f2)$
⟨*proof*⟩

lemma *inv-fun-scale* [*simp*]:
  $inv\ (f\ ::\ nat \Rightarrow\ 'a::ring) \Longrightarrow inv\ (fun\text{-}scale\ r\ f)$
⟨*proof*⟩

linear-poly type – rat coeffs

typedef *linear-poly* = $\{c\ ::\ var \Rightarrow rat.\ inv\ c\}$
  ⟨*proof*⟩

Linear polynomials are of the form $a_1 \cdot x_1 + ... + a_n \cdot x_n$. Their formalization follows the data-refinement approach of Isabelle/HOL [2]. Abstract representation of polynomials are functions mapping variables to their coefficients, where only finitely many variables have non-zero coefficients. Operations on polynomials are defined as operations on functions. For example, the sum of p_1 and p_2 is defined by $\lambda v.\ p_1\ v + p_2\ v$ and the value of a polynomial p for a valuation v (denoted by $p\{\!|v|\!\}$), is defined by $\sum x \mid p\ x \neq (0::'b).\ p\ x * v\ x$. Executable representation of polynomials uses RBT mappings instead of functions.

setup-lifting *type-definition-linear-poly*

Vector space operations on polynomials

instantiation *linear-poly* :: *rational-vector*
begin

lift-definition *zero-linear-poly* :: *linear-poly* **is** *fun-zero* ⟨*proof*⟩

lift-definition *plus-linear-poly* :: *linear-poly* ⇒ *linear-poly* ⇒ *linear-poly* **is** *fun-plus*
  ⟨*proof*⟩

lift-definition *scaleRat-linear-poly* :: *rat* ⇒ *linear-poly* ⇒ *linear-poly* **is** *fun-scale*
  ⟨*proof*⟩

definition *uminus-linear-poly* :: *linear-poly* ⇒ *linear-poly* **where**
  uminus-linear-poly lp = −*1* ∗*R lp*

definition *minus-linear-poly* :: *linear-poly* ⇒ *linear-poly* ⇒ *linear-poly* **where**
  minus-linear-poly lp1 lp2 = *lp1* + (− *lp2*)

instance
⟨*proof*⟩

end

Coefficient

lift-definition *coeff* :: *linear-poly* ⇒ *var* ⇒ *rat* **is** *fun-coeff* ⟨*proof*⟩

lemma *coeff-plus* [*simp*] : *coeff* (*lp1* + *lp2*) *var* = *coeff lp1 var* + *coeff lp2 var*
⟨*proof*⟩

lemma *coeff-scaleRat* [*simp*]: *coeff* (*k* ∗*R lp1*) *var* = *k* ∗ *coeff lp1 var*
⟨*proof*⟩

lemma *coeff-uminus* [*simp*]: *coeff* (−*lp*) *var* = − *coeff lp var*
⟨*proof*⟩

lemma *coeff-minus* [*simp*]: *coeff* (*lp1* − *lp2*) *var* = *coeff lp1 var* − *coeff lp2 var*
⟨*proof*⟩

Set of variables

lift-definition *vars* :: *linear-poly* ⇒ *var set* **is** *fun-vars* ⟨*proof*⟩

lemma *coeff-zero*: *coeff p x* ≠ *0* ⟷ *x* ∈ *vars p*
⟨*proof*⟩

lemma *finite-vars*: *finite* (*vars p*)
⟨*proof*⟩

List of variables

lift-definition *vars-list* :: *linear-poly* ⇒ *var list* **is** *fun-vars-list* ⟨*proof*⟩

lemma *set-vars-list*: *set* (*vars-list lp*) = *vars lp*
⟨*proof*⟩

Construct single variable polynomial

lift-definition *Var* :: *var* ⇒ *linear-poly* **is** *fun-var* ⟨*proof*⟩

Value of a polynomial in a given valuation

lift-definition *valuate* :: *linear-poly* ⇒ *′a valuation* ⇒ (*′a*::*rational-vector*) **is** *fun-valuate*
⟨*proof*⟩

syntax
-valuate :: *linear-poly* ⇒ *′a valuation* ⇒ *′a* (- ⦃ - ⦄)
translations
p⦃*v*⦄ == *CONST valuate p v*

lemma *valuate-zero*: (*0* ⦃*v*⦄) = *0*
⟨*proof*⟩

lemma
valuate-diff: (*p* ⦃*v1*⦄) − (*p* ⦃*v2*⦄) = (*p* ⦃ λ *x*. *v1 x* − *v2 x* ⦄)
⟨*proof*⟩

lemma *valuate-opposite-val*:
shows p ⦃ λ x. $-$ v x ⦄ $= -$ (p ⦃ v ⦄)
⟨*proof*⟩

lemma *valuate-nonneg*:
fixes v :: *'a::linordered-rational-vector valuation*
assumes $\forall$ $x \in$ *vars p*. (*coeff p x* $> 0 \longrightarrow v\ x \geq 0$) $\wedge$ (*coeff p x* $< 0 \longrightarrow v\ x \leq 0$)
shows p ⦃ v ⦄ ≥ 0
⟨*proof*⟩

lemma *valuate-nonpos*:
fixes v :: *'a::linordered-rational-vector valuation*
assumes $\forall$ $x \in$ *vars p*. (*coeff p x* $> 0 \longrightarrow v\ x \leq 0$) $\wedge$ (*coeff p x* $< 0 \longrightarrow v\ x \geq 0$)
shows p ⦃ v ⦄ ≤ 0
⟨*proof*⟩

lemma *valuate-uminus*: $(-p)$ ⦃v⦄ $= -$ (p ⦃v⦄)
⟨*proof*⟩

lemma *valuate-add-lemma*:
fixes v :: *'a* $\Rightarrow$ *'b::rational-vector*
assumes *finite* $\{v.\ f1\ v \neq 0\}$ *finite* $\{v.\ f2\ v \neq 0\}$
shows
$(\sum x{\in}\{v.\ f1\ v + f2\ v \neq 0\}.\ (f1\ x + f2\ x) *R\ v\ x) =$
$(\sum x{\in}\{v.\ f1\ v \neq 0\}.\ f1\ x *R\ v\ x) + (\sum x{\in}\{v.\ f2\ v \neq 0\}.\ f2\ x *R\ v\ x)$
⟨*proof*⟩

lemma *valuate-add*: $(p1 + p2)$ ⦃v⦄ $=$ ($p1$ ⦃v⦄) $+$ ($p2$ ⦃v⦄)
⟨*proof*⟩

lemma *valuate-minus*: $(p1 - p2)$ ⦃v⦄ $=$ ($p1$ ⦃v⦄) $-$ ($p2$ ⦃v⦄)
⟨*proof*⟩

lemma *valuate-scaleRat*:
(c $*R$ lp) ⦃ v ⦄ $= c$ $*R$ (lp⦃v⦄)
⟨*proof*⟩

lemma *valuate-Var*: (*Var* x) ⦃v⦄ $= v\ x$
⟨*proof*⟩

lemma *valuate-sum*: (($\sum x{\in}A.\ f\ x$) ⦃ v ⦄) $=$ ($\sum x{\in}A.$ ((f x) ⦃ v ⦄))
⟨*proof*⟩

lemma *distinct-vars-list*:
distinct (*vars-list p*)

⟨*proof*⟩

lemma *zero-coeff-zero*: $p = 0 \longleftrightarrow (\forall\ v.\ coeff\ p\ v = 0)$
⟨*proof*⟩

lemma *all-val*:
assumes $\forall\ (v::var \Rightarrow 'a::lrv).\ \exists\ v'.\ (\forall\ x \in vars\ p.\ v'\ x = v\ x) \wedge (p\ \{\!|v'|\!\} = 0)$
shows $p = 0$
⟨*proof*⟩

lift-definition *lp-monom* :: $rat \Rightarrow var \Rightarrow linear\text{-}poly$ **is**
$\lambda\ c\ x\ y.\ if\ x = y\ then\ c\ else\ 0$ ⟨*proof*⟩

lemma *valuate-lp-monom*: $((lp\text{-}monom\ c\ x)\ \{\!|v|\!\}) = c * (v\ x)$
⟨*proof*⟩

lemma *valuate-lp-monom-1*[*simp*]: $((lp\text{-}monom\ 1\ x)\ \{\!|v|\!\}) = v\ x$
⟨*proof*⟩

lemma *coeff-lp-monom* [*simp*]:
shows $coeff\ (lp\text{-}monom\ c\ v)\ v' = (if\ v = v'\ then\ c\ else\ 0)$
⟨*proof*⟩

lemma *vars-uminus* [*simp*]: $vars\ (-p) = vars\ p$
⟨*proof*⟩

lemma *vars-plus* [*simp*]: $vars\ (p1 + p2) \subseteq vars\ p1 \cup vars\ p2$
⟨*proof*⟩

lemma *vars-minus* [*simp*]: $vars\ (p1 - p2) \subseteq vars\ p1 \cup vars\ p2$
⟨*proof*⟩

lemma *vars-lp-monom*: $vars\ (lp\text{-}monom\ r\ x) = (if\ r = 0\ then\ \{\}\ else\ \{x\})$
⟨*proof*⟩

lemma *vars-scaleRat1*: $vars\ (c *R\ p) \subseteq vars\ p$
⟨*proof*⟩

lemma *vars-scaleRat*: $c \neq 0 \Longrightarrow vars(c *R\ p) = vars\ p$
⟨*proof*⟩

lemma *vars-Var* [*simp*]: $vars\ (Var\ x) = \{x\}$
⟨*proof*⟩

lemma *coeff-Var1* [*simp*]: $coeff\ (Var\ x)\ x = 1$
⟨*proof*⟩

lemma *coeff-Var2*: $x \neq y \Longrightarrow coeff\ (Var\ x)\ y = 0$

⟨*proof*⟩

lemma *valuate-depend*:
assumes $\forall\ x \in vars\ p.\ v\ x = v'\ x$
shows $(p\ \{\!|v|\!\}) = (p\ \{\!|v'|\!\})$
⟨*proof*⟩

lemma *valuate-update-x-lemma*:
fixes $v1\ v2$ :: *'a::rational-vector valuation*
assumes
$\forall y.\ f\ y \neq 0 \longrightarrow y \neq x \longrightarrow v1\ y = v2\ y$
finite $\{v.\ f\ v \neq 0\}$
shows
$(\sum x{\in}\{v.\ f\ v \neq 0\}.\ f\ x *R\ v1\ x) + f\ x *R\ (v2\ x - v1\ x) = (\sum x{\in}\{v.\ f\ v \neq 0\}.\ f\ x *R\ v2\ x)$
⟨*proof*⟩

lemma *valuate-update-x*:
fixes $v1\ v2$ :: *'a::rational-vector valuation*
assumes $\forall y \in vars\ lp.\ y{\neq}x \longrightarrow v1\ y = v2\ y$
shows $lp\ \{\!|v1|\!\} + coeff\ lp\ x *R\ (v2\ x - v1\ x) = (lp\ \{\!|v2|\!\})$
⟨*proof*⟩

lemma *vars-zero*: $vars\ 0 = \{\}$
⟨*proof*⟩

lemma *vars-empty-zero*: $vars\ lp = \{\} \longleftrightarrow lp = 0$
⟨*proof*⟩

definition *max-var*:: *linear-poly* ⇒ *var* **where**
$max\text{-}var\ lp \equiv if\ lp = 0\ then\ 0\ else\ Max\ (vars\ lp)$

lemma *max-var-max*:
assumes $a \in vars\ lp$
shows $max\text{-}var\ lp \geq a$
⟨*proof*⟩

lemma *max-var-code*[*code*]:
$max\text{-}var\ lp = (let\ vl = vars\text{-}list\ lp$
$in\ if\ vl = []\ then\ 0\ else\ foldl\ max\ (hd\ vl)\ (tl\ vl))$
⟨*proof*⟩

definition *monom-var*:: *linear-poly* ⇒ *var* **where**
$monom\text{-}var\ l = max\text{-}var\ l$

definition *monom-coeff*:: *linear-poly* ⇒ *rat* **where**
$monom\text{-}coeff\ l = coeff\ l\ (monom\text{-}var\ l)$

definition *is-monom* :: *linear-poly* ⇒ *bool* **where**

is-monom l $\longleftrightarrow$ *length* (*vars-list l*) = *1*

lemma *is-monom-vars-not-empty*:
is-monom l $\Longrightarrow$ *vars l* $\neq$ {}
⟨*proof*⟩

lemma *monom-var-in-vars*:
is-monom l $\Longrightarrow$ *monom-var l* $\in$ *vars l*
⟨*proof*⟩

lemma *zero-is-no-monom*[*simp*]: ¬ *is-monom 0*
⟨*proof*⟩

lemma *is-monom-monom-coeff-not-zero*:
is-monom l $\Longrightarrow$ *monom-coeff l* $\neq$ *0*
⟨*proof*⟩

lemma *list-two-elements*:
⟦*y* $\in$ *set l*; *x* $\in$ *set l*; *length l* = *Suc 0*; *y* $\neq$ *x*⟧ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *is-monom-vars-monom-var*:
assumes *is-monom l*
shows *vars l* = {*monom-var l*}
⟨*proof*⟩

lemma *monom-valuate*:
assumes *is-monom m*
shows *m*⦃*v*⦄ = (*monom-coeff m*) *∗R* *v* (*monom-var m*)
⟨*proof*⟩

lemma *coeff-zero-simp* [*simp*]:
coeff 0 v = *0*
⟨*proof*⟩

lemma *poly-eq-iff*: *p* = *q* $\longleftrightarrow$ (∀ *v*. *coeff p v* = *coeff q v*)
⟨*proof*⟩

lemma *poly-eqI*:
assumes $\bigwedge$*v*. *coeff p v* = *coeff q v*
shows *p* = *q*
⟨*proof*⟩

lemma *coeff-sum-list*:
assumes *distinct xs*
shows *coeff* ($\sum$ *x*←*xs*. *f x* *∗R* *lp-monom 1 x*) *v* = (*if v* $\in$ *set xs then f v else 0*)
⟨*proof*⟩

lemma *linear-poly-sum*:

p ⦃ *v* ⦄ = (∑*x*∈*vars p*. *coeff p x* ∗*R v x*)
⟨*proof*⟩

lemma *all-valuate-zero*: **assumes** ⋀(*v*::′*a*::*lrv valuation*). *p* ⦃*v*⦄ = *0*
shows *p* = *0*
⟨*proof*⟩

lemma *linear-poly-eqI*: **assumes** ⋀(*v*::′*a*::*lrv valuation*). (*p* ⦃*v*⦄) = (*q* ⦃*v*⦄)
shows *p* = *q*
⟨*proof*⟩

lemma *monom-poly-assemble*:
assumes *is-monom p*
shows *monom-coeff p* ∗*R lp-monom 1* (*monom-var p*) = *p*
⟨*proof*⟩

lemma *coeff-sum*: *coeff* (*sum* (*f* :: - ⇒ *linear-poly*) *is*) *x* = *sum* (λ *i*. *coeff* (*f i*) *x*) *is*
⟨*proof*⟩

end

theory *Linear-Poly-Maps*
imports *Abstract-Linear-Poly*
HOL−Library.Finite-Map
HOL−Library.Monad-Syntax
begin

definition *get-var-coeff* :: (*var*, *rat*) *fmap* ⇒ *var* ⇒ *rat* **where**
get-var-coeff lp v == *case fmlookup lp v of None* ⇒ *0* | *Some c* ⇒ *c*

definition *set-var-coeff* :: *var* ⇒ *rat* ⇒ (*var*, *rat*) *fmap* ⇒ (*var*, *rat*) *fmap* **where**
set-var-coeff v c lp ==
if c = *0 then fmdrop v lp else fmupd v c lp*

lift-definition *LinearPoly* :: (*var*, *rat*) *fmap* ⇒ *linear-poly* **is** *get-var-coeff*
⟨*proof*⟩

definition *ordered-keys* :: (′*a* :: *linorder*, ′*b*)*fmap* ⇒ ′*a list* **where**
ordered-keys m = *sorted-list-of-set* (*fset* (*fmdom m*))

context includes *fmap.lifting lifting-syntax*
begin

lemma [*transfer-rule*]: (((=) ===> (=)) ===> *pcr-linear-poly* ===> (=)) (=) *pcr-linear-poly*

⟨*proof*⟩

lemma [*transfer-rule*]: (*pcr-fmap* (=) (=) ===> *pcr-linear-poly*) (λ *f x*. *case f x of None* ⇒ *0* | *Some x* ⇒ *x*) *LinearPoly*
⟨*proof*⟩

lift-definition *linear-poly-map* :: *linear-poly* ⇒ (*var*, *rat*) *fmap* **is**
λ *lp x*. *if lp x* = *0 then None else Some* (*lp x*) ⟨*proof*⟩

lemma *certificate*[*code abstype*]:
LinearPoly (*linear-poly-map lp*) = *lp*
⟨*proof*⟩

Zero

definition *zero* :: (*var*, *rat*)*fmap* **where** *zero* = *fmempty*

lemma [*code abstract*]:
linear-poly-map 0 = *zero* ⟨*proof*⟩

Addition

definition *add-monom* :: *rat* ⇒ *var* ⇒ (*var*, *rat*) *fmap* ⇒ (*var*, *rat*) *fmap* **where**
add-monom c v lp == *set-var-coeff v* (*c* + *get-var-coeff lp v*) *lp*

definition *add* :: (*var*, *rat*) *fmap* ⇒ (*var*, *rat*) *fmap* ⇒ (*var*, *rat*) *fmap* **where**
add lp1 lp2 = *foldl* (λ *lp v*. *add-monom* (*get-var-coeff lp1 v*) *v lp*) *lp2* (*ordered-keys lp1*)

lemma *lookup-add-monom*:
get-var-coeff lp v + *c* ≠ *0* ⟹
fmlookup (*add-monom c v lp*) *v* = *Some* (*get-var-coeff lp v* + *c*)
get-var-coeff lp v + *c* = *0* ⟹
fmlookup (*add-monom c v lp*) *v* = *None*
x ≠ *v* ⟹ *fmlookup* (*add-monom c v lp*) *x* = *fmlookup lp x*
⟨*proof*⟩

lemma *fmlookup-fold-not-mem*: *x* ∉ *set k1* ⟹
fmlookup (*foldl* (λ*lp v*. *add-monom* (*get-var-coeff P1 v*) *v lp*) *P2 k1*) *x*
= *fmlookup P2 x*
⟨*proof*⟩

lemma [*code abstract*]:
linear-poly-map (*p1* + *p2*) = *add* (*linear-poly-map p1*) (*linear-poly-map p2*)
⟨*proof*⟩

Scaling

definition *scale* :: *rat* ⇒ (*var*, *rat*) *fmap* ⇒ (*var*, *rat*) *fmap* **where**
scale r lp = (*if r* = *0 then fmempty else* (*fmmap* ((∗) *r*) *lp*))

lemma [*code abstract*]:

linear-poly-map (*r* ∗*R* *p*) = *scale r* (*linear-poly-map p*)
⟨*proof*⟩

lemma *coeff-code* [*code*]:
coeff lp = *get-var-coeff* (*linear-poly-map lp*)
⟨*proof*⟩

lemma *Var-code*[*code abstract*]:
linear-poly-map (*Var x*) = *set-var-coeff x 1 fmempty*
⟨*proof*⟩

lemma *vars-code*[*code*]: *vars lp* = *fset* (*fmdom* (*linear-poly-map lp*))
⟨*proof*⟩

lemma *vars-list-code*[*code*]: *vars-list lp* = *ordered-keys* (*linear-poly-map lp*)
⟨*proof*⟩

lemma *valuate-code*[*code*]: *valuate lp val* = (
let lpm = *linear-poly-map lp*
in sum-list (*map* (λ *x*. (*the* (*fmlookup lpm x*)) ∗*R* (*val x*)) (*vars-list lp*)))
⟨*proof*⟩

end

lemma *lp-monom-code*[*code*]: *linear-poly-map* (*lp-monom c x*) = (*if c* = *0 then fmempty else fmupd x c fmempty*)
⟨*proof*⟩
include *fmap.lifting*
⟨*proof*⟩

instantiation *linear-poly* :: *equal*
begin

definition *equal-linear-poly x y* = (*linear-poly-map x* = *linear-poly-map y*)

instance
⟨*proof*⟩
end

end

5 Rational Numbers Extended with Infinitesimal Element

theory *QDelta*
  imports
    Abstract-Linear-Poly
    Simplex-Algebra
begin

datatype *QDelta* = *QDelta rat rat*

primrec *qdfst* :: *QDelta* ⇒ *rat* **where**
  qdfst (*QDelta a b*) = *a*

primrec *qdsnd* :: *QDelta* ⇒ *rat* **where**
  qdsnd (*QDelta a b*) = *b*

lemma [*simp*]: *QDelta* (*qdfst qd*) (*qdsnd qd*) = *qd*
  ⟨*proof*⟩

lemma [*simp*]: ⟦*QDelta.qdsnd x* = *QDelta.qdsnd y*; *QDelta.qdfst y* = *QDelta.qdfst x*⟧ ⟹ *x* = *y*
  ⟨*proof*⟩

instantiation *QDelta* :: *rational-vector*
begin

definition *zero-QDelta* :: *QDelta*
  where
    0 = *QDelta 0 0*

definition *plus-QDelta* :: *QDelta* ⇒ *QDelta* ⇒ *QDelta*
  where
    qd1 + *qd2* = *QDelta* (*qdfst qd1* + *qdfst qd2*) (*qdsnd qd1* + *qdsnd qd2*)

definition *minus-QDelta* :: *QDelta* ⇒ *QDelta* ⇒ *QDelta*
  where
    qd1 − *qd2* = *QDelta* (*qdfst qd1* − *qdfst qd2*) (*qdsnd qd1* − *qdsnd qd2*)

definition *uminus-QDelta* :: *QDelta* ⇒ *QDelta*
  where
    − *qd* = *QDelta* (− (*qdfst qd*)) (− (*qdsnd qd*))

definition *scaleRat-QDelta* :: *rat* ⇒ *QDelta* ⇒ *QDelta*
  where
    r **R qd* = *QDelta* (*r**(*qdfst qd*)) (*r**(*qdsnd qd*))

instance
⟨*proof*⟩

end

instantiation *QDelta* :: *linorder*
begin
definition *less-eq-QDelta* :: *QDelta* ⇒ *QDelta* ⇒ *bool*
  where
    $qd1 \leq qd2 \longleftrightarrow (qdfst\ qd1 < qdfst\ qd2) \lor (qdfst\ qd1 = qdfst\ qd2 \land qdsnd\ qd1 \leq qdsnd\ qd2)$

definition *less-QDelta* :: *QDelta* ⇒ *QDelta* ⇒ *bool*
  where
    $qd1 < qd2 \longleftrightarrow (qdfst\ qd1 < qdfst\ qd2) \lor (qdfst\ qd1 = qdfst\ qd2 \land qdsnd\ qd1 < qdsnd\ qd2)$

instance ⟨*proof*⟩
end

instantiation *QDelta*:: *linordered-rational-vector*
begin
instance ⟨*proof*⟩
end

instantiation *QDelta* :: *lrv*
begin
definition *one-QDelta* **where**
  one-QDelta = *QDelta 1 0*
instance ⟨*proof*⟩
end

definition *δ0* :: *QDelta* ⇒ *QDelta* ⇒ *rat*
  where
    δ0 qd1 qd2 ==
    let c1 = *qdfst qd1*; *c2* = *qdfst qd2*; *k1* = *qdsnd qd1*; *k2* = *qdsnd qd2 in*
      (*if* ($c1 < c2 \land k1 > k2$) *then*
            $(c2 - c1) / (k1 - k2)$
       else
            1
)

definition *val* :: *QDelta* ⇒ *rat* ⇒ *rat*
  where $val\ qd\ \delta = (qdfst\ qd) + \delta * (qdsnd\ qd)$

lemma *val-plus*:
  $val\ (qd1 + qd2)\ \delta = val\ qd1\ \delta + val\ qd2\ \delta$
  ⟨*proof*⟩

lemma *val-scaleRat*:
  $val\ (c *R\ qd)\ \delta = c * val\ qd\ \delta$

⟨*proof*⟩

lemma *qdfst-setsum*:
  $finite\ A \Longrightarrow qdfst\ (\sum x\in A.\ f\ x) = (\sum x\in A.\ qdfst\ (f\ x))$
  ⟨*proof*⟩

lemma *qdsnd-setsum*:
  $finite\ A \Longrightarrow qdsnd\ (\sum x\in A.\ f\ x) = (\sum x\in A.\ qdsnd\ (f\ x))$
  ⟨*proof*⟩

lemma *valuate-valuate-rat*:
  $lp\ \{\!|(\lambda v.\ (QDelta\ (vl\ v)\ 0))|\!\} = QDelta\ (lp\{\!|vl|\!\})\ 0$
  ⟨*proof*⟩

lemma *valuate-rat-valuate*:
  $lp\{\!|(\lambda v.\ val\ (vl\ v)\ \delta)|\!\} = val\ (lp\{\!|vl|\!\})\ \delta$
  ⟨*proof*⟩

lemma *delta0*:
  assumes $qd1 \le qd2$
  shows $\forall\ \varepsilon.\ \varepsilon > 0 \wedge \varepsilon \le (\delta 0\ qd1\ qd2) \longrightarrow val\ qd1\ \varepsilon \le val\ qd2\ \varepsilon$
⟨*proof*⟩

primrec
  δ-*min* ::(*QDelta* × *QDelta*) *list* ⇒ *rat* **where**
  $\delta\text{-}min\ [] = 1\ |$
  $\delta\text{-}min\ (h\ \#\ t) = min\ (\delta\text{-}min\ t)\ (\delta 0\ (fst\ h)\ (snd\ h))$

lemma *delta-gt-zero*:
  $\delta\text{-}min\ l > 0$
  ⟨*proof*⟩

lemma *delta-le-one*:
  $\delta\text{-}min\ l \le 1$
  ⟨*proof*⟩

lemma *delta-min-append*:
  $\delta\text{-}min\ (as\ @\ bs) = min\ (\delta\text{-}min\ as)\ (\delta\text{-}min\ bs)$
  ⟨*proof*⟩

lemma *delta-min-mono*: $set\ as \subseteq set\ bs \Longrightarrow \delta\text{-}min\ bs \le \delta\text{-}min\ as$
⟨*proof*⟩

lemma *delta-min*:
  assumes $\forall\ qd1\ qd2.\ (qd1,\ qd2) \in set\ qd \longrightarrow qd1 \le qd2$
  shows $\forall\ \varepsilon.\ \varepsilon > 0 \wedge \varepsilon \le \delta\text{-}min\ qd \longrightarrow (\forall\ qd1\ qd2.\ (qd1,\ qd2) \in set\ qd \longrightarrow val\ qd1\ \varepsilon \le val\ qd2\ \varepsilon)$
  ⟨*proof*⟩

lemma *QDelta-0-0*: *QDelta 0 0 = 0* ⟨*proof*⟩
lemma *qdsnd-0*: *qdsnd 0 = 0* ⟨*proof*⟩
lemma *qdfst-0*: *qdfst 0 = 0* ⟨*proof*⟩

end

6 The Simplex Algorithm

theory *Simplex*
  imports
    Linear-Poly-Maps
    QDelta
    Rel-Chain
    Simplex-Algebra
    HOL−Library.Multiset
    HOL−Library.RBT-Mapping
    HOL−Library.Code-Target-Numeral
begin

Linear constraints are of the form $p \bowtie c$, where p is a homogenenous linear polynomial, c is a rational constant and $\bowtie \in \{<, >, \leq, \geq, =\}$. Their abstract syntax is given by the *constraint* type, and semantics is given by the relation $\models_c$, defined straightforwardly by primitive recursion over the *constraint* type. A set of constraints is satisfied, denoted by $\models_{cs}$, if all constraints are. There is also an indexed version $\models_{ics}$ which takes an explicit set of indices and then only demands that these constraints are satisfied.

datatype *constraint = LT linear-poly rat*
  | *GT linear-poly rat*
  | *LEQ linear-poly rat*
  | *GEQ linear-poly rat*
  | *EQ linear-poly rat*

Indexed constraints are just pairs of indices and constraints. Indices will be used to identify constraints, e.g., to easily specify an unsatisfiable core by a list of indices.

type-synonym *′i i-constraint = ′i × constraint*

abbreviation (*input*) *restrict-to* :: *′i set ⇒* (*′i × ′a*) *set ⇒ ′a set* **where**
  restrict-to I xs ≡ snd ‘ (*xs ∩* (*I × UNIV*))

The operation *restrict-to* is used to select constraints for a given index set.

abbreviation (*input*) *flat* :: (*′i × ′a*) *set ⇒ ′a set* **where**
  flat xs ≡ snd ‘ xs

The operation *flat* is used to drop indices from a set of indexed constraints.

abbreviation (*input*) *flat-list* :: (*'i* × *'a*) *list* ⇒ *'a list* **where**
flat-list xs ≡ *map snd xs*

primrec
satisfies-constraint :: *'a* :: *lrv valuation* ⇒ *constraint* ⇒ *bool* (**infixl** $\models_c$ *100*)
where
$v \models_c (LT\ l\ r) \longleftrightarrow (l\{v\}) < r *R\ 1$
$|\ v \models_c GT\ l\ r \longleftrightarrow (l\{v\}) > r *R\ 1$
$|\ v \models_c LEQ\ l\ r \longleftrightarrow (l\{v\}) \leq r *R\ 1$
$|\ v \models_c GEQ\ l\ r \longleftrightarrow (l\{v\}) \geq r *R\ 1$
$|\ v \models_c EQ\ l\ r \longleftrightarrow (l\{v\}) = r *R\ 1$

abbreviation *satisfies-constraints* :: *rat valuation* ⇒ *constraint set* ⇒ *bool* (**infixl** $\models_{cs}$ *100*) **where**
$v \models_{cs} cs \equiv \forall\ c \in cs.\ v \models_c c$

lemma *unsat-mono*: **assumes** $\neg (\exists\ v.\ v \models_{cs} cs)$
and $cs \subseteq ds$
shows $\neg (\exists\ v.\ v \models_{cs} ds)$
⟨*proof*⟩

fun *i-satisfies-cs* (**infixl** $\models_{ics}$ *100*) **where**
$(I,v) \models_{ics} cs \longleftrightarrow v \models_{cs}$ *restrict-to I cs*

definition *distinct-indices* :: (*'i* × *'c*) *list* ⇒ *bool* **where**
distinct-indices as = (*distinct* (*map fst as*))

lemma *distinct-indicesD*: *distinct-indices as* ⟹ (*i*,*x*) ∈ *set as* ⟹ (*i*,*y*) ∈ *set as* ⟹ *x* = *y*
⟨*proof*⟩

For the unsat-core predicate we only demand minimality in case that the indices are distinct. Otherwise, minimality does in general not hold. For instance, consider the input constraints $c_1 : x < 0$, $c_2 : x > 2$ and $c_2 : x < 1$ where the index c_2 occurs twice. If the simplex-method first encounters constraint c_1, then it will detect that there is a conflict between c_1 and the first c_2-constraint. Consequently, the index-set $\{c_1, c_2\}$ will be returned, but this set is not minimal since $\{c_2\}$ is already unsatisfiable.

definition *minimal-unsat-core* :: *'i set* ⇒ *'i i-constraint list* ⇒ *bool* **where**
minimal-unsat-core I ics = ((*I* ⊆ *fst* ‘ *set ics*) ∧ (¬ (∃ *v*. (*I*,*v*) $\models_{ics}$ *set ics*))
∧ (*distinct-indices ics* ⟶ (∀ *J*. *J* ⊂ *I* ⟶ (∃ *v*. (*J*,*v*) $\models_{ics}$ *set ics*))))

6.1 Procedure Specification

abbreviation (*input*) *Unsat* **where** *Unsat* ≡ *Inl*
abbreviation (*input*) *Sat* **where** *Sat* ≡ *Inr*

The specification for the satisfiability check procedure is given by:

locale *Solve* =
— Decide if the given list of constraints is satisfiable. Return either an unsat core, or a satisfying valuation.
fixes *solve* :: *'i i-constraint list* ⇒ *'i list* + *rat valuation*
— If the status *Sat* is returned, then returned valuation satisfies all constraints.
assumes *simplex-sat*: *solve cs* = *Sat v* ⟹ $v \models_{cs}$ *flat* (*set cs*)
— If the status *Unsat* is returned, then constraints are unsatisfiable, i.e., an unsatisfiable core is returned.
assumes *simplex-unsat*: *solve cs* = *Unsat I* ⟹ *minimal-unsat-core* (*set I*) *cs*

abbreviation (*input*) *look* **where** *look* ≡ *Mapping.lookup*
abbreviation (*input*) *upd* **where** *upd* ≡ *Mapping.update*

lemma *look-upd*: *look* (*upd k v m*) = (*look m*)(*k* ↦ *v*)
⟨*proof*⟩

lemmas *look-upd-simps*[*simp*] = *look-upd Mapping.lookup-empty*

definition *map2fun*:: (*var*, *'a* :: *zero*) *mapping* ⇒ *var* ⇒ *'a* **where**
map2fun v ≡ λ*x*. *case look v x of None* ⇒ *0* | *Some y* ⇒ *y*
syntax
-map2fun :: (*var*, *'a*) *mapping* ⇒ *var* ⇒ *'a* (⟨-⟩)
translations
⟨*v*⟩ == *CONST map2fun v*

lemma *map2fun-def'*:
⟨*v*⟩ *x* ≡ *case Mapping.lookup v x of None* ⇒ *0* | *Some y* ⇒ *y*
⟨*proof*⟩

Note that the above specification requires returning a valuation (defined as a HOL function), which is not efficiently executable. In order to enable more efficient data structures for representing valuations, a refinement of this specification is needed and the function *solve* is replaced by the function *solve-exec* returning optional (*var*, *rat*) *mapping* instead of *var* ⇒ *rat* function. This way, efficient data structures for representing mappings can be easily plugged-in during code generation [2]. A conversion from the *mapping* datatype to HOL function is denoted by ⟨-⟩ and given by: ⟨*v*⟩ *x* ≡ *case Mapping.lookup v x of None* ⇒ *0*::*'a* | *Some y* ⇒ *y*.

locale *SolveExec* =
fixes *solve-exec* :: *'i i-constraint list* ⇒ *'i list* + (*var*, *rat*) *mapping*
assumes *simplex-sat0*: *solve-exec cs* = *Sat v* ⟹ ⟨*v*⟩ $\models_{cs}$ *flat* (*set cs*)
assumes *simplex-unsat0*: *solve-exec cs* = *Unsat I* ⟹ *minimal-unsat-core* (*set I*) *cs*
begin
definition *solve* **where**
solve cs ≡ *case solve-exec cs of Sat v* ⇒ *Sat* ⟨*v*⟩ | *Unsat c* ⇒ *Unsat c*
end

sublocale *SolveExec* < *Solve solve*
⟨*proof*⟩

6.2 Handling Strict Inequalities

The first step of the procedure is removing all equalities and strict inequalities. Equalities can be easily rewritten to non-strict inequalities. Removing strict inequalities can be done by replacing the list of constraints by a new one, formulated over an extension $\mathbb{Q}'$ of the space of rationals $\mathbb{Q}$. $\mathbb{Q}'$ must have a structure of a linearly ordered vector space over $\mathbb{Q}$ (represented by the type class *lrv*) and must guarantee that if some non-strict constraints are satisfied in $\mathbb{Q}'$, then there is a satisfying valuation for the original constraints in $\mathbb{Q}$. Our final implementation uses the $\mathbb{Q}_\delta$ space, defined in [1] (basic idea is to replace $p < c$ by $p \leq c - \delta$ and $p > c$ by $p \geq c + \delta$ for a symbolic parameter δ). So, all constraints are reduced to the form $p \bowtie b$, where p is a linear polynomial (still over $\mathbb{Q}$), b is constant from $\mathbb{Q}'$ and $\bowtie \in \{\leq, \geq\}$. The non-strict constraints are represented by the type *'a ns-constraint*, and their semantics is denoted by $\models_{ns}$ and $\models_{nss}$. The indexed variant is $\models_{inss}$.

datatype *'a ns-constraint* = *LEQ-ns linear-poly 'a* | *GEQ-ns linear-poly 'a*

type-synonym (*'i*,*'a*) *i-ns-constraint* = *'i* × *'a ns-constraint*

primrec *satisfiable-ns-constraint* :: *'a*::*lrv valuation* ⇒ *'a ns-constraint* ⇒ *bool* (**infixl** $\models_{ns}$ *100*) **where**
  $v \models_{ns}$ *LEQ-ns l r* ⟷ $l\{\!|v|\!\} \leq r$
| $v \models_{ns}$ *GEQ-ns l r* ⟷ $l\{\!|v|\!\} \geq r$

abbreviation *satisfies-ns-constraints* :: *'a*::*lrv valuation* ⇒ *'a ns-constraint set* ⇒ *bool* (**infixl** $\models_{nss}$ *100*) **where**
  $v \models_{nss} cs \equiv \forall\ c \in cs.\ v \models_{ns} c$

fun *i-satisfies-ns-constraints* :: *'i set* × *'a*::*lrv valuation* ⇒ (*'i*,*'a*) *i-ns-constraint set* ⇒ *bool* (**infixl** $\models_{inss}$ *100*) **where**
  $(I,v) \models_{inss} cs \longleftrightarrow v \models_{nss}$ *restrict-to I cs*

lemma *i-satisfies-ns-constraints-mono*:
  $(I,v) \models_{inss} cs \Longrightarrow J \subseteq I \Longrightarrow (J,v) \models_{inss} cs$
  ⟨*proof*⟩

primrec *poly* :: *'a ns-constraint* ⇒ *linear-poly* **where**
  poly (*LEQ-ns p a*) = *p*
| *poly* (*GEQ-ns p a*) = *p*

primrec *ns-constraint-const* :: *'a ns-constraint* ⇒ *'a* **where**
  ns-constraint-const (*LEQ-ns p a*) = *a*
| *ns-constraint-const* (*GEQ-ns p a*) = *a*

definition *distinct-indices-ns* :: $('i,'a :: lrv)$ *i-ns-constraint set* ⇒ *bool* **where**
distinct-indices-ns ns = ((∀ *n1 n2 i*. (*i*,*n1*) ∈ *ns* ⟶ (*i*,*n2*) ∈ *ns* ⟶
poly n1 = *poly n2* ∧ *ns-constraint-const n1* = *ns-constraint-const n2*))

definition *minimal-unsat-core-ns* :: *'i set* ⇒ $('i,'a :: lrv)$ *i-ns-constraint set* ⇒ *bool*
where
minimal-unsat-core-ns I cs = ((*I* ⊆ *fst* ' *cs*) ∧ (¬ (∃ *v*. (*I*,*v*) $\models_{inss}$ *cs*))
∧ (*distinct-indices-ns cs* ⟶ (∀ *J* ⊂ *I*. ∃ *v*. (*J*,*v*) $\models_{inss}$ *cs*)))

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

locale *To-ns* =
— Convert a constraint to an equisatisfiable non-strict constraint list. The conversion must work for arbitrary subsets of constraints – selected by some index set I – in order to carry over unsat-cores and in order to support incremental simplex solving.
fixes *to-ns* :: *'i i-constraint list* ⇒ *('i,'a::lrv) i-ns-constraint list*
— Convert the valuation that satisfies all non-strict constraints to the valuation that satisfies all initial constraints.
fixes *from-ns* :: (*var*, *'a*) *mapping* ⇒ *'a ns-constraint list* ⇒ (*var*, *rat*) *mapping*
assumes *to-ns-unsat*: *minimal-unsat-core-ns I* (*set* (*to-ns cs*)) ⟹ *minimal-unsat-core I cs*
assumes *i-to-ns-sat*: (*I*,⟨*v'*⟩) $\models_{inss}$ *set* (*to-ns cs*) ⟹ (*I*,⟨*from-ns v'* (*flat-list* (*to-ns cs*))⟩) $\models_{ics}$ *set cs*
assumes *to-ns-indices*: *fst* ' *set* (*to-ns cs*) = *fst* ' *set cs*
assumes *distinct-cond*: *distinct-indices cs* ⟹ *distinct-indices-ns* (*set* (*to-ns cs*))

begin
lemma *to-ns-sat*: ⟨*v'*⟩ $\models_{nss}$ *flat* (*set* (*to-ns cs*)) ⟹ ⟨*from-ns v'* (*flat-list* (*to-ns cs*))⟩ $\models_{cs}$ *flat* (*set cs*)
⟨*proof*⟩
end

locale *Solve-exec-ns* =
fixes *solve-exec-ns* :: *('i,'a::lrv) i-ns-constraint list* ⇒ *'i list* + (*var*, *'a*) *mapping*
assumes *simplex-ns-sat*: *solve-exec-ns cs* = *Sat v* ⟹ ⟨*v*⟩ $\models_{nss}$ *flat* (*set cs*)
assumes *simplex-ns-unsat*: *solve-exec-ns cs* = *Unsat I* ⟹ *minimal-unsat-core-ns* (*set I*) (*set cs*)

After the transformation, the procedure is reduced to solving only the non-strict constraints, implemented in the *solve-exec-ns* function having an analogous specification to the *solve* function. If *to-ns*, *from-ns* and *solve-exec-ns* are available, the *solve-exec* function can be easily defined and it can be easily shown that this definition satisfies its specification (also analogous to *solve*).

locale *SolveExec'* = *To-ns to-ns from-ns* + *Solve-exec-ns solve-exec-ns* **for**
to-ns:: *'i i-constraint list* ⇒ *('i,'a::lrv) i-ns-constraint list* **and**
from-ns :: (*var*, *'a*) *mapping* ⇒ *'a ns-constraint list* ⇒ (*var*, *rat*) *mapping* **and**
solve-exec-ns :: *('i,'a) i-ns-constraint list* ⇒ *'i list* + (*var*, *'a*) *mapping*

begin

definition *solve-exec* **where**
  solve-exec cs ≡ *let cs′ = to-ns cs in case solve-exec-ns cs′*
    of Sat v ⇒ *Sat (from-ns v (flat-list cs′))*
    | *Unsat is* ⇒ *Unsat is*

end

sublocale *SolveExec′* < *SolveExec solve-exec*
  ⟨*proof*⟩

6.3 Preprocessing

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

type-synonym *eq* = *var* × *linear-poly*
primrec *lhs* :: *eq* ⇒ *var* **where** *lhs* (*l*, *r*) = *l*
primrec *rhs* :: *eq* ⇒ *linear-poly* **where** *rhs* (*l*, *r*) = *r*
abbreviation *rvars-eq* :: *eq* ⇒ *var set* **where**
  rvars-eq eq ≡ *vars* (*rhs eq*)

definition *satisfies-eq* :: *′a::rational-vector valuation* ⇒ *eq* ⇒ *bool* (**infixl** $\models_e$ *100*)
where
  $v \models_e eq \equiv v\ (lhs\ eq) = (rhs\ eq)\{\!|v|\!\}$

lemma *satisfies-eq-iff*: $v \models_e (x,\ p) \equiv v\ x = p\{\!|v|\!\}$
  ⟨*proof*⟩

type-synonym *tableau* =*eq list*

definition *satisfies-tableau* ::*′a::rational-vector valuation* ⇒ *tableau* ⇒ *bool* (**infixl** $\models_t$ *100*) **where**
  $v \models_t t \equiv \forall\ e \in set\ t.\ v \models_e e$

definition *lvars* :: *tableau* ⇒ *var set* **where**

lvars t = *set* (*map lhs t*)
definition *rvars* :: *tableau* ⇒ *var set* **where**
rvars t = $\bigcup$ (*set* (*map rvars-eq t*))
abbreviation *tvars* **where** *tvars t* ≡ *lvars t* ∪ *rvars t*

The condition that the rhss are non-zero is required to obtain minimal unsatisfiable cores. To observe the problem with 0 as rhs, consider the tableau $x = 0$ in combination with atom $(A : x \leq 0)$ where then $(B : x \geq 1)$ is asserted. In this case, the unsat core would be computed as $\{A, B\}$, although already $\{B\}$ is unsatisfiable.

definition *normalized-tableau* :: *tableau* ⇒ *bool* (△) **where**
normalized-tableau t ≡ *distinct* (*map lhs t*) ∧ *lvars t* ∩ *rvars t* = {} ∧ *0* ∉ *rhs* ' *set t*

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

lemma *normalized-tableau-unique-eq-for-lvar*:
assumes △ *t*
shows ∀ *x* ∈ *lvars t*. ∃! *p*. (*x*, *p*) ∈ *set t*
⟨*proof*⟩

lemma *recalc-tableau-lvars*:
assumes △ *t*
shows ∀ *v*. ∃ *v'*. (∀ *x* ∈ *rvars t*. *v x* = *v' x*) ∧ $v' \models_t t$
⟨*proof*⟩

lemma *tableau-perm*:
assumes *lvars t1* = *lvars t2* *rvars t1* = *rvars t2*
△ *t1* △ *t2* $\bigwedge$ *v*::*'a*::*lrv valuation*. $v \models_t t1 \longleftrightarrow v \models_t t2$
shows *mset t1* = *mset t2*
⟨*proof*⟩

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

datatype *'a atom* = *Leq var 'a* | *Geq var 'a*

primrec *atom-var*::*'a atom* ⇒ *var* **where**
atom-var (*Leq var a*) = *var*
| *atom-var* (*Geq var a*) = *var*

primrec *atom-const*::′*a atom* ⇒ ′*a* **where**
atom-const (*Leq var a*) = *a*
| *atom-const* (*Geq var a*) = *a*

primrec *satisfies-atom* :: ′*a*::*linorder valuation* ⇒ ′*a atom* ⇒ *bool* (**infixl** $\models_a$ *100*) **where**
$v \models_a$ *Leq x c* ⟷ *v x* ≤ *c* | $v \models_a$ *Geq x c* ⟷ *v x* ≥ *c*

definition *satisfies-atom-set* :: ′*a*::*linorder valuation* ⇒ ′*a atom set* ⇒ *bool* (**infixl** $\models_{as}$ *100*) **where**
$v \models_{as}$ *as* ≡ ∀ *a* ∈ *as*. $v \models_a$ *a*

definition *satisfies-atom′* :: ′*a*::*linorder valuation* ⇒ ′*a atom* ⇒ *bool* (**infixl** $\models_{ae}$ *100*) **where**
$v \models_{ae}$ *a* ⟷ *v* (*atom-var a*) = *atom-const a*

lemma *satisfies-atom′-stronger*: $v \models_{ae}$ *a* ⟹ $v \models_a$ *a* ⟨*proof*⟩

abbreviation *satisfies-atom-set′* :: ′*a*::*linorder valuation* ⇒ ′*a atom set* ⇒ *bool* (**infixl** $\models_{aes}$ *100*) **where**
$v \models_{aes}$ *as* ≡ ∀ *a* ∈ *as*. $v \models_{ae}$ *a*

lemma *satisfies-atom-set′-stronger*: $v \models_{aes}$ *as* ⟹ $v \models_{as}$ *as*
⟨*proof*⟩

There is also the indexed variant of an atom

type-synonym (′*i*,′*a*) *i-atom* = ′*i* × ′*a atom*

fun *i-satisfies-atom-set* :: ′*i set* × ′*a*::*linorder valuation* ⇒ (′*i*,′*a*) *i-atom set* ⇒ *bool* (**infixl** $\models_{ias}$ *100*) **where**
(*I*,*v*) $\models_{ias}$ *as* ⟷ $v \models_{as}$ *restrict-to I as*

fun *i-satisfies-atom-set′* :: ′*i set* × ′*a*::*linorder valuation* ⇒ (′*i*,′*a*) *i-atom set* ⇒ *bool* (**infixl** $\models_{iaes}$ *100*) **where**
(*I*,*v*) $\models_{iaes}$ *as* ⟷ $v \models_{aes}$ *restrict-to I as*

lemma *i-satisfies-atom-set′-stronger*: *Iv* $\models_{iaes}$ *as* ⟹ *Iv* $\models_{ias}$ *as*
⟨*proof*⟩

lemma *satisifies-atom-restrict-to-Cons*: $v \models_{as}$ *restrict-to I* (*set as*) ⟹ (*i* ∈ *I* ⟹ $v \models_a$ *a*)
⟹ $v \models_{as}$ *restrict-to I* (*set* ((*i*,*a*) # *as*))
⟨*proof*⟩

lemma *satisfies-tableau-Cons*: $v \models_t$ *t* ⟹ $v \models_e$ *e* ⟹ $v \models_t$ (*e* # *t*)
⟨*proof*⟩

definition *distinct-indices-atoms* :: (′*i*,′*a*) *i-atom set* ⇒ *bool* **where**

distinct-indices-atoms as = (∀ *i a b*. (*i*,*a*) ∈ *as* ⟶ (*i*,*b*) ∈ *as* ⟶ *atom-var a* = *atom-var b* ∧ *atom-const a* = *atom-const b*)

The specification of the preprocessing function is given by:

locale *Preprocess* = **fixes** *preprocess*::(*'i*,*'a*::*lrv*) *i-ns-constraint list* ⇒ *tableau*× (*'i*,*'a*) *i-atom list*
× ((*var*,*'a*) *mapping* ⇒ (*var*,*'a*) *mapping*) × *'i list*
assumes
— The returned tableau is always normalized.
preprocess-tableau-normalized: *preprocess cs* = (*t*,*as*,*trans-v*,*U*) ⟹ △ *t* **and**

— Tableau and atoms are equisatisfiable with starting non-strict constraints.
i-preprocess-sat: ⋀ *v*. *preprocess cs* = (*t*,*as*,*trans-v*,*U*) ⟹ *I* ∩ *set U* = {} ⟹ (*I*,⟨*v*⟩) $\models_{ias}$ *set as* ⟹ ⟨*v*⟩ $\models_t$ *t* ⟹ (*I*,⟨*trans-v v*⟩) $\models_{inss}$ *set cs* **and**

preprocess-unsat: *preprocess cs* = (*t*, *as*,*trans-v*,*U*) ⟹ (*I*,*v*) $\models_{inss}$ *set cs* ⟹ ∃ *v'*. (*I*,*v'*) $\models_{ias}$ *set as* ∧ *v'* $\models_t$ *t* **and**

— distinct indices on ns-constraints ensures distinct indices in atoms
preprocess-distinct: *preprocess cs* = (*t*, *as*,*trans-v*, *U*) ⟹ *distinct-indices-ns* (*set cs*) ⟹ *distinct-indices-atoms* (*set as*) **and**

— unsat indices
preprocess-unsat-indices: *preprocess cs* = (*t*, *as*,*trans-v*, *U*) ⟹ *i* ∈ *set U* ⟹ ¬ (∃ *v*. ({*i*},*v*) $\models_{inss}$ *set cs*) **and**

— preprocessing cannot introduce new indices
preprocess-index: *preprocess cs* = (*t*,*as*,*trans-v*, *U*) ⟹ *fst* ‘ *set as* ∪ *set U* ⊆ *fst* ‘ *set cs*
begin
lemma *preprocess-sat*: *preprocess cs* = (*t*,*as*,*trans-v*,*U*) ⟹ *U* = [] ⟹ ⟨*v*⟩ $\models_{as}$ *flat* (*set as*) ⟹ ⟨*v*⟩ $\models_t$ *t* ⟹ ⟨*trans-v v*⟩ $\models_{nss}$ *flat* (*set cs*)
⟨*proof*⟩

end

definition *minimal-unsat-core-tabl-atoms* :: *'i set* ⇒ *tableau* ⇒ (*'i*,*'a*::*lrv*) *i-atom set* ⇒ *bool* **where**
minimal-unsat-core-tabl-atoms I t as = (*I* ⊆ *fst* ‘ *as* ∧ (¬ (∃ *v*. *v* $\models_t$ *t* ∧ (*I*,*v*) $\models_{ias}$ *as*)) ∧
(*distinct-indices-atoms as* ⟶ (∀ *J* ⊂ *I*. ∃ *v*. *v* $\models_t$ *t* ∧ (*J*,*v*) $\models_{iaes}$ *as*)))

lemma *minimal-unsat-core-tabl-atomsD*: **assumes** *minimal-unsat-core-tabl-atoms I t as*
shows *I* ⊆ *fst* ‘ *as*
¬ (∃ *v*. *v* $\models_t$ *t* ∧ (*I*,*v*) $\models_{ias}$ *as*)
distinct-indices-atoms as ⟹ *J* ⊂ *I* ⟹ ∃ *v*. *v* $\models_t$ *t* ∧ (*J*,*v*) $\models_{iaes}$ *as*
⟨*proof*⟩

locale *AssertAll* =
fixes *assert-all* :: *tableau* ⇒ ('*i*,'*a*::*lrv*) *i-atom list* ⇒ '*i list* + (*var*, '*a*)*mapping*
assumes *assert-all-sat*: $\triangle\ t \Longrightarrow$ *assert-all t as* = *Sat v* $\Longrightarrow \langle v\rangle \models_t t \wedge \langle v\rangle \models_{as}$ *flat* (*set as*)
assumes *assert-all-unsat*: $\triangle\ t \Longrightarrow$ *assert-all t as* = *Unsat I* $\Longrightarrow$ *minimal-unsat-core-tabl-atoms* (*set I*) *t* (*set as*)

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

locale *Solve-exec-ns'* = *Preprocess preprocess* + *AssertAll assert-all* **for**
preprocess:: ('*i*,'*a*::*lrv*) *i-ns-constraint list* ⇒ *tableau* × ('*i*,'*a*) *i-atom list* × ((*var*,'*a*)*mapping* ⇒ (*var*,'*a*)*mapping*) × '*i list* **and**
assert-all :: *tableau* ⇒ ('*i*,'*a*::*lrv*) *i-atom list* ⇒ '*i list* + (*var*, '*a*) *mapping*
begin
definition *solve-exec-ns* **where**

solve-exec-ns s ≡
case preprocess s of (*t*,*as*,*trans-v*,*ui*) ⇒
(*case ui of i* # - ⇒ *Inl* [*i*] | - ⇒
(*case assert-all t as of Inl I* ⇒ *Inl I* | *Inr v* ⇒ *Inr* (*trans-v v*)))
end

context *Preprocess*
begin

lemma *preprocess-unsat-index*: **assumes** *prep*: *preprocess cs* = (*t*,*as*,*trans-v*,*ui*)
and *i*: *i* ∈ *set ui*
shows *minimal-unsat-core-ns* {*i*} (*set cs*)
⟨*proof*⟩

lemma *preprocess-minimal-unsat-core*: **assumes** *prep*: *preprocess cs* = (*t*,*as*,*trans-v*,*ui*)
and *unsat*: *minimal-unsat-core-tabl-atoms I t* (*set as*)
and *inter*: *I* ∩ *set ui* = {}
shows *minimal-unsat-core-ns I* (*set cs*)
⟨*proof*⟩
end

sublocale *Solve-exec-ns'* < *Solve-exec-ns solve-exec-ns*
⟨*proof*⟩

6.4 Incrementally Asserting Atoms

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

type-synonym $'a$ *bounds* $= var \rightharpoonup 'a$

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

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

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

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

abbreviation (*input*) *le* **where**
$le\ lt\ x\ y \equiv lt\ x\ y \vee x = y$
definition *geub* ($\trianglerighteq_{ub}$) **where**
$\trianglerighteq_{ub}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow False \mid Some\ b' \Rightarrow le\ lt\ b'\ c$
definition *gtub* ($\triangleright_{ub}$) **where**
$\triangleright_{ub}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow False \mid Some\ b' \Rightarrow lt\ b'\ c$
definition *leub* ($\trianglelefteq_{ub}$) **where**
$\trianglelefteq_{ub}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow True \mid Some\ b' \Rightarrow le\ lt\ c\ b'$
definition *ltub* ($\triangleleft_{ub}$) **where**
$\triangleleft_{ub}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow True \mid Some\ b' \Rightarrow lt\ c\ b'$
definition *lelb* ($\trianglelefteq_{lb}$) **where**
$\trianglelefteq_{lb}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow False \mid Some\ b' \Rightarrow le\ lt\ c\ b'$
definition *ltlb* ($\triangleleft_{lb}$) **where**
$\triangleleft_{lb}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow False \mid Some\ b' \Rightarrow lt\ c\ b'$
definition *gelb* ($\trianglerighteq_{lb}$) **where**
$\trianglerighteq_{lb}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow True \mid Some\ b' \Rightarrow le\ lt\ b'\ c$
definition *gtlb* ($\triangleright_{lb}$) **where**
$\triangleright_{lb}\ lt\ c\ b \equiv case\ b\ of\ None \Rightarrow True \mid Some\ b' \Rightarrow lt\ b'\ c$

definition *ge-ubound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $\geq_{ub}$ *100*) **where**
$c \geq_{ub} b = \trianglerighteq_{ub}\ (<)\ c\ b$
definition *gt-ubound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $>_{ub}$ *100*) **where**
$c >_{ub} b = \triangleright_{ub}\ (<)\ c\ b$
definition *le-ubound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $\leq_{ub}$ *100*) **where**
$c \leq_{ub} b = \trianglelefteq_{ub}\ (<)\ c\ b$
definition *lt-ubound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $<_{ub}$ *100*) **where**
$c <_{ub} b = \triangleleft_{ub}\ (<)\ c\ b$
definition *le-lbound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $\leq_{lb}$ *100*) **where**
$c \leq_{lb} b = \trianglelefteq_{lb}\ (<)\ c\ b$
definition *lt-lbound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $<_{lb}$ *100*) **where**
$c <_{lb} b = \triangleleft_{lb}\ (<)\ c\ b$
definition *ge-lbound* :: $'a::linorder \Rightarrow 'a\ option \Rightarrow bool$ (**infixl** $\geq_{lb}$ *100*) **where**
$c \geq_{lb} b = \trianglerighteq_{lb}\ (<)\ c\ b$

definition *gt-lbound* :: $'a$::*linorder* ⇒ $'a$ *option* ⇒ *bool* (**infixl** $>_{lb}$ *100*) **where**
$c >_{lb} b = \rhd_{lb} (<)\ c\ b$

lemmas *bound-compare′-defs* =
geub-def gtub-def leub-def ltub-def
gelb-def gtlb-def lelb-def ltlb-def

lemmas *bound-compare″-defs* =
ge-ubound-def gt-ubound-def le-ubound-def lt-ubound-def
le-lbound-def lt-lbound-def ge-lbound-def gt-lbound-def

lemmas *bound-compare-defs* = *bound-compare′-defs bound-compare″-defs*

lemma *opposite-dir* [*simp*]:
$\unlhd_{lb} (>)\ a\ b = \unrhd_{ub} (<)\ a\ b$
$\unlhd_{ub} (>)\ a\ b = \unrhd_{lb} (<)\ a\ b$
$\unrhd_{lb} (>)\ a\ b = \unlhd_{ub} (<)\ a\ b$
$\unrhd_{ub} (>)\ a\ b = \unlhd_{lb} (<)\ a\ b$
$\lhd_{lb} (>)\ a\ b = \rhd_{ub} (<)\ a\ b$
$\lhd_{ub} (>)\ a\ b = \rhd_{lb} (<)\ a\ b$
$\rhd_{lb} (>)\ a\ b = \lhd_{ub} (<)\ a\ b$
$\rhd_{ub} (>)\ a\ b = \lhd_{lb} (<)\ a\ b$
⟨*proof*⟩

lemma [*simp*]: $\neg\ c \geq_{ub} None\ \ \neg\ c \leq_{lb} None$
⟨*proof*⟩

lemma *neg-bounds-compare*:
$(\neg\ (c \geq_{ub} b)) \Longrightarrow c <_{ub} b\ (\neg\ (c \leq_{ub} b)) \Longrightarrow c >_{ub} b$
$(\neg\ (c >_{ub} b)) \Longrightarrow c \leq_{ub} b\ (\neg\ (c <_{ub} b)) \Longrightarrow c \geq_{ub} b$
$(\neg\ (c \leq_{lb} b)) \Longrightarrow c >_{lb} b\ (\neg\ (c \geq_{lb} b)) \Longrightarrow c <_{lb} b$
$(\neg\ (c <_{lb} b)) \Longrightarrow c \geq_{lb} b\ (\neg\ (c >_{lb} b)) \Longrightarrow c \leq_{lb} b$
⟨*proof*⟩

lemma *bounds-compare-contradictory* [*simp*]:
$[\![c \geq_{ub} b;\ c <_{ub} b]\!] \Longrightarrow False\ [\![c \leq_{ub} b;\ c >_{ub} b]\!] \Longrightarrow False$
$[\![c >_{ub} b;\ c \leq_{ub} b]\!] \Longrightarrow False\ [\![c <_{ub} b;\ c \geq_{ub} b]\!] \Longrightarrow False$
$[\![c \leq_{lb} b;\ c >_{lb} b]\!] \Longrightarrow False\ [\![c \geq_{lb} b;\ c <_{lb} b]\!] \Longrightarrow False$
$[\![c <_{lb} b;\ c \geq_{lb} b]\!] \Longrightarrow False\ [\![c >_{lb} b;\ c \leq_{lb} b]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *compare-strict-nonstrict*:
$x <_{ub} b \Longrightarrow x \leq_{ub} b$
$x >_{ub} b \Longrightarrow x \geq_{ub} b$

$x <_{lb} b \Longrightarrow x \leq_{lb} b$
$x >_{lb} b \Longrightarrow x \geq_{lb} b$
⟨*proof*⟩

lemma [*simp*]:
$[\![x \leq c;\ c <_{ub} b]\!] \Longrightarrow x <_{ub} b$
$[\![x < c;\ c \leq_{ub} b]\!] \Longrightarrow x <_{ub} b$
$[\![x \leq c;\ c \leq_{ub} b]\!] \Longrightarrow x \leq_{ub} b$
$[\![x \geq c;\ c >_{lb} b]\!] \Longrightarrow x >_{lb} b$
$[\![x > c;\ c \geq_{lb} b]\!] \Longrightarrow x >_{lb} b$
$[\![x \geq c;\ c \geq_{lb} b]\!] \Longrightarrow x \geq_{lb} b$
⟨*proof*⟩

lemma *bounds-lg* [*simp*]:
$[\![c >_{ub} b;\ x \leq_{ub} b]\!] \Longrightarrow x < c$
$[\![c \geq_{ub} b;\ x <_{ub} b]\!] \Longrightarrow x < c$
$[\![c \geq_{ub} b;\ x \leq_{ub} b]\!] \Longrightarrow x \leq c$
$[\![c <_{lb} b;\ x \geq_{lb} b]\!] \Longrightarrow x > c$
$[\![c \leq_{lb} b;\ x >_{lb} b]\!] \Longrightarrow x > c$
$[\![c \leq_{lb} b;\ x \geq_{lb} b]\!] \Longrightarrow x \geq c$
⟨*proof*⟩

lemma *bounds-compare-Some* [*simp*]:
$x \leq_{ub}$ *Some* $c \longleftrightarrow x \leq c\ x \geq_{ub}$ *Some* $c \longleftrightarrow x \geq c$
$x <_{ub}$ *Some* $c \longleftrightarrow x < c\ x >_{ub}$ *Some* $c \longleftrightarrow x > c$
$x \geq_{lb}$ *Some* $c \longleftrightarrow x \geq c\ x \leq_{lb}$ *Some* $c \longleftrightarrow x \leq c$
$x >_{lb}$ *Some* $c \longleftrightarrow x > c\ x <_{lb}$ *Some* $c \longleftrightarrow x < c$
⟨*proof*⟩

fun *in-bounds* **where**
in-bounds $x\ v$ (*lb*, *ub*) = ($v\ x \geq_{lb}$ *lb* $x \wedge v\ x \leq_{ub}$ *ub* x)

fun *satisfies-bounds* :: *'a::linorder valuation* ⇒ *'a bounds* × *'a bounds* ⇒ *bool*
(**infixl** $\models_b$ *100*) **where**
$v \models_b b \longleftrightarrow (\forall\ x.$ *in-bounds* $x\ v\ b)$
declare *satisfies-bounds.simps* [*simp del*]

lemma *satisfies-bounds-iff*:
$v \models_b$ (*lb*, *ub*) $\longleftrightarrow (\forall\ x.\ v\ x \geq_{lb}$ *lb* $x \wedge v\ x \leq_{ub}$ *ub* x)
⟨*proof*⟩

lemma *not-in-bounds*:
¬ (*in-bounds* $x\ v$ (*lb*, *ub*)) = ($v\ x <_{lb}$ *lb* $x \vee v\ x >_{ub}$ *ub* x)
⟨*proof*⟩

fun *atoms-equiv-bounds* :: *'a::linorder atom set* ⇒ *'a bounds* × *'a bounds* ⇒ *bool*
(**infixl** $\doteq$ *100*) **where**
as $\doteq$ (*lb*, *ub*) $\longleftrightarrow (\forall\ v.\ v \models_{as}$ *as* $\longleftrightarrow v \models_b$ (*lb*, *ub*))

declare *atoms-equiv-bounds.simps* [*simp del*]

lemma *atoms-equiv-bounds-simps*:
$as \doteq (lb,\ ub) \equiv \forall\ v.\ v \models_{as} as \longleftrightarrow v \models_b (lb,\ ub)$
⟨*proof*⟩

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

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

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

Since we also need to now about the indices of atoms, actually, the bounds are also indexed, and in addition to the flag for unsatisfiability, we also store an optional unsat core.

type-synonym $'i$ *bound-index* $= var \Rightarrow\ 'i$

type-synonym $('i,'a)$ *bounds-index* $= (var,\ ('i \times\ 'a))mapping$

datatype $('i,'a)$ *state* $=$ *State*
($\mathcal{T}$: *tableau*)
($\mathcal{B}_{il}$: $('i,'a)$ *bounds-index*)
($\mathcal{B}_{iu}$: $('i,'a)$ *bounds-index*)
($\mathcal{V}$: $(var,\ 'a)$ *mapping*)
($\mathcal{U}$: *bool*)
($\mathcal{U}_c$: $'i$ *list option*)

definition *indexl* :: $('i,'a)$ *state* $\Rightarrow\ 'i$ *bound-index* ($\mathcal{I}_l$) **where**
$\mathcal{I}_l\ s = (fst\ o\ the)\ o\ look\ (\mathcal{B}_{il}\ s)$

definition *boundsl* :: $('i,'a)$ *state* $\Rightarrow\ 'a$ *bounds* ($\mathcal{B}_l$) **where**
$\mathcal{B}_l\ s = map\text{-}option\ snd\ o\ look\ (\mathcal{B}_{il}\ s)$

definition *indexu* :: $('i,'a)$ *state* $\Rightarrow\ 'i$ *bound-index* ($\mathcal{I}_u$) **where**
$\mathcal{I}_u\ s = (fst\ o\ the)\ o\ look\ (\mathcal{B}_{iu}\ s)$

definition *boundsu* :: $('i,'a)$ *state* $\Rightarrow\ 'a$ *bounds* ($\mathcal{B}_u$) **where**
$\mathcal{B}_u\ s = map\text{-}option\ snd\ o\ look\ (\mathcal{B}_{iu}\ s)$

abbreviation *BoundsIndicesMap* ($\mathcal{B}_i$) **where** $\mathcal{B}_i\ s \equiv (\mathcal{B}_{il}\ s,\ \mathcal{B}_{iu}\ s)$
abbreviation *Bounds* :: $('i,'a)$ *state* $\Rightarrow$ $'a$ *bounds* $\times$ $'a$ *bounds* ($\mathcal{B}$) **where** $\mathcal{B}\ s \equiv (\mathcal{B}_l\ s,\ \mathcal{B}_u\ s)$
abbreviation *Indices* :: $('i,'a)$ *state* $\Rightarrow$ $'i$ *bound-index* $\times$ $'i$ *bound-index* ($\mathcal{I}$) **where** $\mathcal{I}\ s \equiv (\mathcal{I}_l\ s,\ \mathcal{I}_u\ s)$
abbreviation *BoundsIndices* :: $('i,'a)$ *state* $\Rightarrow$ $('a$ *bounds* $\times$ $'a$ *bounds*$)$ $\times$ $('i$ *bound-index* $\times$ $'i$ *bound-index*$)$ ($\mathcal{BI}$)
where $\mathcal{BI}\ s \equiv (\mathcal{B}\ s,\ \mathcal{I}\ s)$

fun *satisfies-bounds-index* :: $'i$ *set* $\times$ $'a$::*lrv valuation* $\Rightarrow$ $('a$ *bounds* $\times$ $'a$ *bounds*$)$ $\times$
$('i$ *bound-index* $\times$ $'i$ *bound-index*$)$ $\Rightarrow$ *bool* (**infixl** $\models_{ib}$ *100*) **where**
$(I,v) \models_{ib} ((BL,BU),(IL,IU)) \longleftrightarrow ($
$(\forall\ x\ c.\ BL\ x = Some\ c \longrightarrow IL\ x \in I \longrightarrow v\ x \geq c)$
$\wedge\ (\forall\ x\ c.\ BU\ x = Some\ c \longrightarrow IU\ x \in I \longrightarrow v\ x \leq c))$
declare *satisfies-bounds-index.simps*[*simp del*]

fun *satisfies-bounds-index'* :: $'i$ *set* $\times$ $'a$::*lrv valuation* $\Rightarrow$ $('a$ *bounds* $\times$ $'a$ *bounds*$)$ $\times$
$('i$ *bound-index* $\times$ $'i$ *bound-index*$)$ $\Rightarrow$ *bool* (**infixl** $\models_{ibe}$ *100*) **where**
$(I,v) \models_{ibe} ((BL,BU),(IL,IU)) \longleftrightarrow ($
$(\forall\ x\ c.\ BL\ x = Some\ c \longrightarrow IL\ x \in I \longrightarrow v\ x = c)$
$\wedge\ (\forall\ x\ c.\ BU\ x = Some\ c \longrightarrow IU\ x \in I \longrightarrow v\ x = c))$
declare *satisfies-bounds-index'.simps*[*simp del*]

fun *atoms-imply-bounds-index* :: $('i,'a$::*lrv*$)$ *i-atom set* $\Rightarrow$ $('a$ *bounds* $\times$ $'a$ *bounds*$)$ $\times$ $('i$ *bound-index* $\times$ $'i$ *bound-index*$)$
$\Rightarrow$ *bool* (**infixl** $\models_i$ *100*) **where**
$as \models_i bi \longleftrightarrow (\forall\ I\ v.\ (I,v) \models_{ias} as \longrightarrow (I,v) \models_{ib} bi)$
declare *atoms-imply-bounds-index.simps*[*simp del*]

lemma *i-satisfies-atom-set-mono*: $as \subseteq as' \Longrightarrow v \models_{ias} as' \Longrightarrow v \models_{ias} as$
⟨*proof*⟩

lemma *atoms-imply-bounds-index-mono*: $as \subseteq as' \Longrightarrow as \models_i bi \Longrightarrow as' \models_i bi$
⟨*proof*⟩

definition *satisfies-state* :: $'a$::*lrv valuation* $\Rightarrow$ $('i,'a)$ *state* $\Rightarrow$ *bool* (**infixl** $\models_s$ *100*)
where
$v \models_s s \equiv v \models_b \mathcal{B}\ s \wedge v \models_t \mathcal{T}\ s$

definition *curr-val-satisfies-state* :: $('i,'a$::*lrv*$)$ *state* $\Rightarrow$ *bool* ($\models$) **where**
$\models s \equiv \langle \mathcal{V}\ s \rangle \models_s s$

fun *satisfies-state-index* :: $'i$ *set* $\times$ $'a$::*lrv valuation* $\Rightarrow$ $('i,'a)$ *state* $\Rightarrow$ *bool* (**infixl** $\models_{is}$ *100*) **where**
$(I,v) \models_{is} s \longleftrightarrow (v \models_t \mathcal{T}\ s \wedge (I,v) \models_{ib} \mathcal{BI}\ s)$
declare *satisfies-state-index.simps*[*simp del*]

fun *satisfies-state-index'* :: *'i set* × *'a::lrv valuation* ⇒ (*'i,'a*) *state* ⇒ *bool* (**infixl** $\models_{ise}$ *100*) **where**
(*I*,*v*) $\models_{ise}$ *s* ⟷ (*v* $\models_t$ $\mathcal{T}$ *s* ∧ (*I*,*v*) $\models_{ibe}$ $\mathcal{BI}$ *s*)
declare *satisfies-state-index'.simps*[*simp del*]

definition *indices-state* :: (*'i,'a*)*state* ⇒ *'i set* **where**
indices-state s = { *i*. ∃ *x b*. *look* ($\mathcal{B}_{il}$ *s*) *x* = *Some* (*i*,*b*) ∨ *look* ($\mathcal{B}_{iu}$ *s*) *x* = *Some* (*i*,*b*)}

distinctness requires that for each index i, there is at most one variable x and bound b such that $x \leq b$ or $x \geq b$ or both are enforced.

definition *distinct-indices-state* :: (*'i,'a*)*state* ⇒ *bool* **where**
distinct-indices-state s = (∀ *i x b x' b'*.
((*look* ($\mathcal{B}_{il}$ *s*) *x* = *Some* (*i*,*b*) ∨ *look* ($\mathcal{B}_{iu}$ *s*) *x* = *Some* (*i*,*b*)) ⟶
(*look* ($\mathcal{B}_{il}$ *s*) *x'* = *Some* (*i*,*b'*) ∨ *look* ($\mathcal{B}_{iu}$ *s*) *x'* = *Some* (*i*,*b'*)) ⟶
(*x* = *x'* ∧ *b* = *b'*)))

lemma *distinct-indices-stateD*: **assumes** *distinct-indices-state s*
shows *look* ($\mathcal{B}_{il}$ *s*) *x* = *Some* (*i*,*b*) ∨ *look* ($\mathcal{B}_{iu}$ *s*) *x* = *Some* (*i*,*b*) ⟹ *look* ($\mathcal{B}_{il}$ *s*) *x'* = *Some* (*i*,*b'*) ∨ *look* ($\mathcal{B}_{iu}$ *s*) *x'* = *Some* (*i*,*b'*)
⟹ *x* = *x'* ∧ *b* = *b'*
⟨*proof*⟩

definition *unsat-state-core* :: (*'i,'a::lrv*) *state* ⇒ *bool* **where**
unsat-state-core s = (*set* (*the* ($\mathcal{U}_c$ *s*)) ⊆ *indices-state s* ∧ (¬ (∃ *v*. (*set* (*the* ($\mathcal{U}_c$ *s*)),*v*) $\models_{is}$ *s*)))

definition *subsets-sat-core* :: (*'i,'a::lrv*) *state* ⇒ *bool* **where**
subsets-sat-core s = ((∀ *I*. *I* ⊂ *set* (*the* ($\mathcal{U}_c$ *s*)) ⟶ (∃ *v*. (*I*,*v*) $\models_{ise}$ *s*)))

definition *minimal-unsat-state-core* :: (*'i,'a::lrv*) *state* ⇒ *bool* **where**
minimal-unsat-state-core s = (*unsat-state-core s* ∧ (*distinct-indices-state s* ⟶ *subsets-sat-core s*))

lemma *minimal-unsat-core-tabl-atoms-mono*: **assumes** *sub*: *as* ⊆ *bs*
and *unsat*: *minimal-unsat-core-tabl-atoms I t as*
shows *minimal-unsat-core-tabl-atoms I t bs*
⟨*proof*⟩

lemma *state-satisfies-index*: **assumes** *v* $\models_s$ *s*
shows (*I*,*v*) $\models_{is}$ *s*
⟨*proof*⟩

lemma *unsat-state-core-unsat*: *unsat-state-core s* ⟹ ¬ (∃ *v*. *v* $\models_s$ *s*)
⟨*proof*⟩

definition *tableau-valuated* (∇) **where**
∇ *s* ≡ ∀ *x* ∈ *tvars* ($\mathcal{T}$ *s*). *Mapping.lookup* ($\mathcal{V}$ *s*) *x* ≠ *None*

definition *index-valid* **where**
index-valid as $(s :: ('i,'a)\ state) = (\forall\ x\ b\ i.$
$(look\ (\mathcal{B}_{il}\ s)\ x = Some\ (i,b) \longrightarrow ((i,\ Geq\ x\ b) \in as))$
$\wedge\ (look\ (\mathcal{B}_{iu}\ s)\ x = Some\ (i,b) \longrightarrow ((i,\ Leq\ x\ b) \in as)))$

lemma *index-valid-indices-state*: $index\text{-}valid\ as\ s \Longrightarrow indices\text{-}state\ s \subseteq fst\ `\ as$
⟨*proof*⟩

lemma *index-valid-mono*: $as \subseteq bs \Longrightarrow index\text{-}valid\ as\ s \Longrightarrow index\text{-}valid\ bs\ s$
⟨*proof*⟩

lemma *index-valid-distinct-indices*: **assumes** *index-valid as s*
and *distinct-indices-atoms as*
shows *distinct-indices-state s*
⟨*proof*⟩

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

definition *updateBI* **where**
[*simp*]: *updateBI field-update i x c s = field-update (upd x (i,c)) s*

fun $\mathcal{B}_{iu}$*-update* **where**
$\mathcal{B}_{iu}$*-update up (State T BIL BIU V U UC) = State T BIL (up BIU) V U UC*

fun $\mathcal{B}_{il}$*-update* **where**
$\mathcal{B}_{il}$*-update up (State T BIL BIU V U UC) = State T (up BIL) BIU V U UC*

fun $\mathcal{V}$*-update* **where**
$\mathcal{V}$*-update V (State T BIL BIU V-old U UC) = State T BIL BIU V U UC*

fun $\mathcal{T}$*-update* **where**
$\mathcal{T}$*-update T (State T-old BIL BIU V U UC) = State T BIL BIU V U UC*

lemma *update-simps*[*simp*]:
$\mathcal{B}_{iu}\ (\mathcal{B}_{iu}\text{-}update\ up\ s) = up\ (\mathcal{B}_{iu}\ s)$
$\mathcal{B}_{il}\ (\mathcal{B}_{iu}\text{-}update\ up\ s) = \mathcal{B}_{il}\ s$
$\mathcal{T}\ (\mathcal{B}_{iu}\text{-}update\ up\ s) = \mathcal{T}\ s$
$\mathcal{V}\ (\mathcal{B}_{iu}\text{-}update\ up\ s) = \mathcal{V}\ s$
$\mathcal{U}\ (\mathcal{B}_{iu}\text{-}update\ up\ s) = \mathcal{U}\ s$

$\mathcal{U}_c$ ($\mathcal{B}_{iu}$-*update up s*) = $\mathcal{U}_c$ *s*
$\mathcal{B}_{il}$ ($\mathcal{B}_{il}$-*update up s*) = *up* ($\mathcal{B}_{il}$ *s*)
$\mathcal{B}_{iu}$ ($\mathcal{B}_{il}$-*update up s*) = $\mathcal{B}_{iu}$ *s*
$\mathcal{T}$ ($\mathcal{B}_{il}$-*update up s*) = $\mathcal{T}$ *s*
$\mathcal{V}$ ($\mathcal{B}_{il}$-*update up s*) = $\mathcal{V}$ *s*
$\mathcal{U}$ ($\mathcal{B}_{il}$-*update up s*) = $\mathcal{U}$ *s*
$\mathcal{U}_c$ ($\mathcal{B}_{il}$-*update up s*) = $\mathcal{U}_c$ *s*
$\mathcal{V}$ ($\mathcal{V}$-*update V s*) = *V*
$\mathcal{B}_{il}$ ($\mathcal{V}$-*update V s*) = $\mathcal{B}_{il}$ *s*
$\mathcal{B}_{iu}$ ($\mathcal{V}$-*update V s*) = $\mathcal{B}_{iu}$ *s*
$\mathcal{T}$ ($\mathcal{V}$-*update V s*) = $\mathcal{T}$ *s*
$\mathcal{U}$ ($\mathcal{V}$-*update V s*) = $\mathcal{U}$ *s*
$\mathcal{U}_c$ ($\mathcal{V}$-*update V s*) = $\mathcal{U}_c$ *s*
$\mathcal{T}$ ($\mathcal{T}$-*update T s*) = *T*
$\mathcal{B}_{il}$ ($\mathcal{T}$-*update T s*) = $\mathcal{B}_{il}$ *s*
$\mathcal{B}_{iu}$ ($\mathcal{T}$-*update T s*) = $\mathcal{B}_{iu}$ *s*
$\mathcal{V}$ ($\mathcal{T}$-*update T s*) = $\mathcal{V}$ *s*
$\mathcal{U}$ ($\mathcal{T}$-*update T s*) = $\mathcal{U}$ *s*
$\mathcal{U}_c$ ($\mathcal{T}$-*update T s*) = $\mathcal{U}_c$ *s*
⟨*proof*⟩

declare
$\mathcal{B}_{iu}$-*update.simps*[*simp del*]
$\mathcal{B}_{il}$-*update.simps*[*simp del*]

fun *set-unsat* :: *'i list* ⇒ (*'i*,*'a*) *state* ⇒ (*'i*,*'a*) *state* **where**
set-unsat I (*State T BIL BIU V U UC*) = *State T BIL BIU V True* (*Some* (*remdups I*))

lemma *set-unsat-simps*[*simp*]: $\mathcal{B}_{il}$ (*set-unsat I s*) = $\mathcal{B}_{il}$ *s*
$\mathcal{B}_{iu}$ (*set-unsat I s*) = $\mathcal{B}_{iu}$ *s*
$\mathcal{T}$ (*set-unsat I s*) = $\mathcal{T}$ *s*
$\mathcal{V}$ (*set-unsat I s*) = $\mathcal{V}$ *s*
$\mathcal{U}$ (*set-unsat I s*) = *True*
$\mathcal{U}_c$ (*set-unsat I s*) = *Some* (*remdups I*)
⟨*proof*⟩

datatype (*'i*,*'a*) *Direction* = *Direction*
(*lt*: *'a*::*linorder* ⇒ *'a* ⇒ *bool*)
(*LBI*: (*'i*,*'a*) *state* ⇒ (*'i*,*'a*) *bounds-index*)
(*UBI*: (*'i*,*'a*) *state* ⇒ (*'i*,*'a*) *bounds-index*)
(*LB*: (*'i*,*'a*) *state* ⇒ *'a bounds*)
(*UB*: (*'i*,*'a*) *state* ⇒ *'a bounds*)
(*LI*: (*'i*,*'a*) *state* ⇒ *'i bound-index*)
(*UI*: (*'i*,*'a*) *state* ⇒ *'i bound-index*)
(*UBI-upd*: ((*'i*,*'a*) *bounds-index* ⇒ (*'i*,*'a*) *bounds-index*) ⇒ (*'i*,*'a*) *state* ⇒ (*'i*,*'a*) *state*)
(*LE*: *var* ⇒ *'a* ⇒ *'a atom*)
(*GE*: *var* ⇒ *'a* ⇒ *'a atom*)

($\mathit{le\text{-}rat}$: $\mathit{rat} \Rightarrow \mathit{rat} \Rightarrow \mathit{bool}$)

definition *Positive* **where**
[*simp*]: $\mathit{Positive} \equiv \mathit{Direction}\ (<)\ \mathcal{B}_{il}\ \mathcal{B}_{iu}\ \mathcal{B}_l\ \mathcal{B}_u\ \mathcal{I}_l\ \mathcal{I}_u\ \mathcal{B}_{iu}\text{-}\mathit{update}\ \mathit{Leq}\ \mathit{Geq}\ (\leq)$

definition *Negative* **where**
[*simp*]: $\mathit{Negative} \equiv \mathit{Direction}\ (>)\ \mathcal{B}_{iu}\ \mathcal{B}_{il}\ \mathcal{B}_u\ \mathcal{B}_l\ \mathcal{I}_u\ \mathcal{I}_l\ \mathcal{B}_{il}\text{-}\mathit{update}\ \mathit{Geq}\ \mathit{Leq}\ (\geq)$

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

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

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

locale *AssertAllState* = **fixes** $\mathit{assert\text{-}all\text{-}state}::\mathit{tableau} \Rightarrow ('i,'a::\mathit{lrv})\ \mathit{i\text{-}atom}\ \mathit{list} \Rightarrow ('i,'a)\ \mathit{state}$
assumes
— The final and the initial tableau are equivalent.
$\mathit{assert\text{-}all\text{-}state\text{-}tableau\text{-}equiv}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s' \Longrightarrow (v::'a\ \mathit{valuation}) \models_t t \longleftrightarrow v \models_t \mathcal{T}\ s'$ **and**

— If $\mathcal{U}$ is not raised, then the valuation in the final state satisfies its tableau and its bounds (that are, in this case, equivalent to the set of all asserted bounds).
$\mathit{assert\text{-}all\text{-}state\text{-}sat}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s' \Longrightarrow \neg\ \mathcal{U}\ s' \Longrightarrow\ \models s'$ **and**

$\mathit{assert\text{-}all\text{-}state\text{-}sat\text{-}atoms\text{-}equiv\text{-}bounds}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s' \Longrightarrow \neg\ \mathcal{U}\ s' \Longrightarrow \mathit{flat}\ (\mathit{set}\ \mathit{as}) \doteq \mathcal{B}\ s'$ **and**

— If $\mathcal{U}$ is raised, then there is no valuation satisfying the tableau and the bounds in the final state (that are, in this case, equivalent to a subset of asserted atoms).
$\mathit{assert\text{-}all\text{-}state\text{-}unsat}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s' \Longrightarrow \mathcal{U}\ s' \Longrightarrow \mathit{minimal\text{-}unsat\text{-}state\text{-}core}\ s'$ **and**

$\mathit{assert\text{-}all\text{-}state\text{-}unsat\text{-}atoms\text{-}equiv\text{-}bounds}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s' \Longrightarrow \mathcal{U}\ s' \Longrightarrow \mathit{set}\ \mathit{as} \models_i \mathcal{BI}\ s'$ **and**

— The set of indices is taken from the constraints
$\mathit{assert\text{-}all\text{-}state\text{-}indices}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s \Longrightarrow \mathit{indices\text{-}state}\ s \subseteq \mathit{fst}\ `\ \mathit{set}\ \mathit{as}$ **and**

$\mathit{assert\text{-}all\text{-}index\text{-}valid}$: $\triangle\ t \Longrightarrow \mathit{assert\text{-}all\text{-}state}\ t\ \mathit{as} = s \Longrightarrow \mathit{index\text{-}valid}\ (\mathit{set}\ \mathit{as})\ s$
begin

definition *assert-all* **where**
assert-all t as $\equiv$ *let s = assert-all-state t as in*
if ($\mathcal{U}$ *s*) *then Unsat* (*the* ($\mathcal{U}_c$ *s*)) *else Sat* ($\mathcal{V}$ *s*)
end

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

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

locale *Init'* =
fixes *init* :: *tableau* $\Rightarrow$ (*'i,'a::lrv*) *state*
assumes *init'-tableau-normalized*: $\triangle$ *t* $\Longrightarrow$ $\triangle$ ($\mathcal{T}$ (*init t*))
assumes *init'-tableau-equiv*: $\triangle$ *t* $\Longrightarrow$ (*v*::*'a valuation*) $\models_t$ *t* = *v* $\models_t$ $\mathcal{T}$ (*init t*)
assumes *init'-sat*: $\triangle$ *t* $\Longrightarrow$ $\neg$ $\mathcal{U}$ (*init t*) $\longrightarrow$ $\models$ (*init t*)
assumes *init'-unsat*: $\triangle$ *t* $\Longrightarrow$ $\mathcal{U}$ (*init t*) $\longrightarrow$ *minimal-unsat-state-core* (*init t*)
assumes *init'-atoms-equiv-bounds*: $\triangle$ *t* $\Longrightarrow$ {} $\doteq$ $\mathcal{B}$ (*init t*)
assumes *init'-atoms-imply-bounds-index*: $\triangle$ *t* $\Longrightarrow$ {} $\models_i$ $\mathcal{BI}$ (*init t*)

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

locale *Init* = **fixes** *init*::*tableau* $\Rightarrow$ (*'i,'a::lrv*) *state*
assumes
— Tableau in the initial state for *t* is *t*: *init-tableau-id*: $\mathcal{T}$ (*init t*) = *t* **and**

— Since unsatisfiability is not detected, $\mathcal{U}$ flag must not be set: *init-unsat-flag*: $\neg$ $\mathcal{U}$ (*init t*) **and**

— The current valuation must satisfy the tableau: *init-satisfies-tableau*: $\langle\mathcal{V}$ (*init t*)$\rangle$ $\models_t$ *t* **and**

— In an inital state no atoms are yet asserted so the bounds must be empty: *init-bounds*: $\mathcal{B}_{il}$ (*init t*) = *Mapping.empty* $\mathcal{B}_{iu}$ (*init t*) = *Mapping.empty* **and**

— All tableau vars are valuated: *init-tableau-valuated*: ∇ (*init t*)

begin

lemma *init-satisfies-bounds*:
$\langle \mathcal{V}\ (init\ t) \rangle \models_b \mathcal{B}\ (init\ t)$
$\langle proof \rangle$

lemma *init-satisfies*:
$\models (init\ t)$
$\langle proof \rangle$

lemma *init-atoms-equiv-bounds*:
$\{\} \doteq \mathcal{B}\ (init\ t)$
$\langle proof \rangle$

lemma *init-atoms-imply-bounds-index*:
$\{\} \models_i \mathcal{BI}\ (init\ t)$
$\langle proof \rangle$

lemma *init-tableau-normalized*:
$\triangle\ t \Longrightarrow \triangle\ (\mathcal{T}\ (init\ t))$
$\langle proof \rangle$

lemma *init-index-valid*: *index-valid as* (*init t*)
$\langle proof \rangle$

lemma *init-indices*: *indices-state* (*init t*) $= \{\}$
$\langle proof \rangle$
end

sublocale *Init* $<$ *Init′ init*
$\langle proof \rangle$

abbreviation *vars-list* **where**
vars-list t $\equiv$ *remdups* (*map lhs t* @ (*concat* (*map* (*Abstract-Linear-Poly.vars-list* $\circ$ *rhs*) *t*)))

lemma *tvars t* = *set* (*vars-list t*)
$\langle proof \rangle$

The *assert* function asserts a single atom. Since the *init* function does not raise the $\mathcal{U}$ flag, from the definition of *assert-loop*, it is clear that the flag is not raised when the *assert* function is called. Moreover, the assumptions about the *assert-all-state* imply that the loop invariant must be that if the $\mathcal{U}$ flag is not raised, then the current valuation must satisfy the state (i.e., $\models s$). The *assert* function will be more easily implemented if it is always applied to a state with a normalized and valuated tableau, so we make this another loop invariant. Therefore, the precondition for the *assert a s* function call is that $\neg\ \mathcal{U}\ s$, $\models\ s$, $\triangle\ (\mathcal{T}\ s)$ and $\nabla\ s$ hold. The specification for *assert* directly follows from the specification of *assert-all-state* (except that it is additionally required that bounds reflect asserted atoms also when unsatisfiability is detected, and that it is required that *assert* keeps the tableau normalized and valuated).

locale *Assert* = **fixes** *assert*::$('i,'a::lrv)$ *i-atom* $\Rightarrow$ $('i,'a)$ *state* $\Rightarrow$ $('i,'a)$ *state*
assumes
— Tableau remains equivalent to the previous one and normalized and valuated.
assert-tableau: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow$ *let* $s' =$ *assert a s in*
$((v::'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ s') \wedge \triangle\ (\mathcal{T}\ s') \wedge \nabla\ s'$ **and**

— If the $\mathcal{U}$ flag is not raised, then the current valuation is updated so that it satisfies the current tableau and the current bounds.
assert-sat: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow \neg\ \mathcal{U}\ (assert\ a\ s) \Longrightarrow\ \models\ (assert\ a\ s)$ **and**

— The set of asserted atoms remains equivalent to the bounds in the state.
assert-atoms-equiv-bounds: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow flat\ ats \doteq \mathcal{B}\ s \Longrightarrow flat\ (ats \cup \{a\}) \doteq \mathcal{B}\ (assert\ a\ s)$ **and**

— There is a subset of asserted atoms which remains index-equivalent to the bounds in the state.
assert-atoms-imply-bounds-index: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow ats \models_i \mathcal{BI}\ s \Longrightarrow$
insert a ats $\models_i \mathcal{BI}\ (assert\ a\ s)$ **and**

— If the $\mathcal{U}$ flag is raised, then there is no valuation that satisfies both the current tableau and the current bounds.
assert-unsat: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ index\text{-}valid\ ats\ s]\!] \Longrightarrow \mathcal{U}\ (assert\ a\ s) \Longrightarrow$ *minimal-unsat-state-core* $(assert\ a\ s)$ **and**

assert-index-valid: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow index\text{-}valid\ ats\ s \Longrightarrow index\text{-}valid\ (insert\ a\ ats)\ (assert\ a\ s)$

begin
lemma *assert-tableau-equiv*: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow (v::'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ (assert\ a\ s)$
⟨*proof*⟩

lemma *assert-tableau-normalized*: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow \triangle\ (\mathcal{T}\ (assert\ a\ s))$
⟨*proof*⟩

lemma *assert-tableau-valuated*: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow \nabla\ (assert\ a\ s)$
⟨*proof*⟩
end

locale *AssertAllState′* = *Init init* + *Assert assert* **for**
init :: *tableau* ⇒ ('*i*,'*a*::*lrv*) *state* **and** *assert* :: ('*i*,'*a*) *i-atom* ⇒ ('*i*,'*a*) *state* ⇒ ('*i*,'*a*) *state*
begin

definition *assert-loop* **where**
assert-loop as s ≡ *foldl* (λ *s′ a*. *if* (𝒰 *s′*) *then s′ else assert a s′*) *s as*

definition *assert-all-state* **where** [*simp*]:
assert-all-state t as ≡ *assert-loop as* (*init t*)

lemma *AssertAllState′-precond*:
△ *t* ⟹ △ (𝒯 (*assert-all-state t as*))
∧ (∇ (*assert-all-state t as*))
∧ (¬ 𝒰 (*assert-all-state t as*) ⟶ ⊨ (*assert-all-state t as*))
⟨*proof*⟩

lemma *AssertAllState′Induct*:
assumes
△ *t*
P {} (*init t*)
⋀ *as bs t*. *as* ⊆ *bs* ⟹ *P as t* ⟹ *P bs t*
⋀ *s a as*. ⟦¬ 𝒰 *s*; ⊨ *s*; △ (𝒯 *s*); ∇ *s*; *P as s*; *index-valid as s*⟧ ⟹ *P* (*insert a as*) (*assert a s*)
shows *P* (*set as*) (*assert-all-state t as*)
⟨*proof*⟩

lemma *AssertAllState-index-valid*: △ *t* ⟹ *index-valid* (*set as*) (*assert-all-state t as*)
⟨*proof*⟩

lemma *AssertAllState′-sat-atoms-equiv-bounds*:
△ *t* ⟹ ¬ 𝒰 (*assert-all-state t as*) ⟹ *flat* (*set as*) ≐ ℬ (*assert-all-state t as*)
⟨*proof*⟩

lemma *AssertAllState′-unsat-atoms-implies-bounds*:
assumes △ *t*
shows *set as* $\models_i$ ℬℐ (*assert-all-state t as*)
⟨*proof*⟩

end

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

sublocale *AssertAllState′* < *AssertAllState assert-all-state*
⟨*proof*⟩

6.5 Asserting Single Atoms

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

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

fun *satisfies-bounds-set* :: $'a{::}linorder\ valuation \Rightarrow 'a\ bounds \times 'a\ bounds \Rightarrow var\ set \Rightarrow bool$ **where**
satisfies-bounds-set $v\ (lb,\ ub)\ S \longleftrightarrow (\forall\ x \in S.\ in\text{-}bounds\ x\ v\ (lb,\ ub))$
declare *satisfies-bounds-set.simps* [*simp del*]
syntax
-satisfies-bounds-set :: $(var \Rightarrow 'a{::}linorder) \Rightarrow 'a\ bounds \times 'a\ bounds \Rightarrow var\ set \Rightarrow bool$ $(\text{-} \models_b \text{-} \parallel/ \text{-})$
translations
$v \models_b b \parallel S == CONST\ satisfies\text{-}bounds\text{-}set\ v\ b\ S$
lemma *satisfies-bounds-set-iff*:
$v \models_b (lb,\ ub) \parallel S \equiv (\forall\ x \in S.\ v\ x \geq_{lb} lb\ x \wedge v\ x \leq_{ub} ub\ x)$
⟨*proof*⟩

definition *curr-val-satisfies-no-lhs* $(\models_{nolhs})$ **where**
$\models_{nolhs} s \equiv \langle \mathcal{V}\ s\rangle \models_t (\mathcal{T}\ s) \wedge (\langle \mathcal{V}\ s\rangle \models_b (\mathcal{B}\ s) \parallel (-\ lvars\ (\mathcal{T}\ s)))$
lemma *satisfies-satisfies-no-lhs*:
$\models s \Longrightarrow \models_{nolhs} s$
⟨*proof*⟩

definition *bounds-consistent* :: $('i,'a{::}linorder)\ state \Rightarrow bool\ (\Diamond)$ **where**
$\Diamond\ s \equiv$
$\forall\ x.\ if\ \mathcal{B}_l\ s\ x = None \vee \mathcal{B}_u\ s\ x = None\ then\ True\ else\ the\ (\mathcal{B}_l\ s\ x) \leq the\ (\mathcal{B}_u\ s\ x)$

So, the *assert-bound* function must ensure that the given atom is included in the bounds, that the tableau remains satisfied by the valuation and that all variables except the lhs variables in the tableau are within their bounds. To formalize this, we introduce the notation $v \models_b (lb,\ ub) \parallel S$, and define v

$\models_b$ $(lb,\ ub) \parallel S \equiv \forall\, x{\in}S.\ v\ x \geq_{lb} lb\ x \wedge v\ x \leq_{ub} ub\ x$, and $\models_{nolhs}\ s \equiv \langle \mathcal{V}\ s\rangle \models_t \mathcal{T}\ s \wedge \langle \mathcal{V}\ s\rangle \models_b \mathcal{B}\ s \parallel -\ lvars\ (\mathcal{T}\ s)$. The *assert-bound* function raises the $\mathcal{U}$ flag if and only if lower and upper bounds overlap. This is formalized as $\Diamond\ s \equiv \forall\, x.\ if\ \mathcal{B}_l\ s\ x = None \vee \mathcal{B}_u\ s\ x = None\ then\ True\ else\ the\ (\mathcal{B}_l\ s\ x) \leq the\ (\mathcal{B}_u\ s\ x)$.

lemma *satisfies-bounds-consistent*:
$(v::'a::linorder\ valuation) \models_b \mathcal{B}\ s \longrightarrow \Diamond\ s$
⟨*proof*⟩

lemma *satisfies-consistent*:
$\models s \longrightarrow \Diamond\ s$
⟨*proof*⟩

lemma *bounds-consistent-geq-lb*:
$[\![\Diamond\ s;\ \mathcal{B}_u\ s\ x_i = Some\ c]\!]$
$\Longrightarrow c \geq_{lb} \mathcal{B}_l\ s\ x_i$
⟨*proof*⟩

lemma *bounds-consistent-leq-ub*:
$[\![\Diamond\ s;\ \mathcal{B}_l\ s\ x_i = Some\ c]\!]$
$\Longrightarrow c \leq_{ub} \mathcal{B}_u\ s\ x_i$
⟨*proof*⟩

lemma *bounds-consistent-gt-ub*:
$[\![\Diamond\ s;\ c <_{lb} \mathcal{B}_l\ s\ x\ ]\!] \Longrightarrow \neg\ c >_{ub} \mathcal{B}_u\ s\ x$
⟨*proof*⟩

lemma *bounds-consistent-lt-lb*:
$[\![\Diamond\ s;\ c >_{ub} \mathcal{B}_u\ s\ x\ ]\!] \Longrightarrow \neg\ c <_{lb} \mathcal{B}_l\ s\ x$
⟨*proof*⟩

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

locale *AssertBound* = **fixes** *assert-bound*::$('i,'a::lrv)$ *i-atom* ⇒ $('i,'a)$ *state* ⇒ $('i,'a)$ *state*
assumes
— The tableau remains unchanged and valuated.

assert-bound-tableau: $[\![\neg\ \mathcal{U}\ s;\ \models s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow assert\text{-}bound\ a\ s = s' \Longrightarrow \mathcal{T}\ s' = \mathcal{T}\ s \wedge \nabla\ s'$ **and**

— If the $\mathcal{U}$ flag is not set, all but the lhs variables in the tableau remain within their bounds, the new valuation satisfies the tableau, and bounds do not overlap.
assert-bound-sat: $[\![\neg\ \mathcal{U}\ s;\ \models s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow assert\text{-}bound\ a\ s = s' \Longrightarrow \neg\ \mathcal{U}\ s' \Longrightarrow \models_{nolhs}\ s' \wedge \Diamond\ s'$ **and**

— The set of asserted atoms remains equivalent to the bounds in the state.

assert-bound-atoms-equiv-bounds: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow$
flat ats $\doteq \mathcal{B}\ s \Longrightarrow$ *flat* $(ats \cup \{a\}) \doteq \mathcal{B}$ (*assert-bound a s*) **and**

assert-bound-atoms-imply-bounds-index: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow$
ats $\models_i \mathcal{BI}\ s \Longrightarrow$ *insert a ats* $\models_i \mathcal{BI}$ (*assert-bound a s*) **and**

— $\mathcal{U}$ flag is raised, only if the bounds became inconsistent:

assert-bound-unsat: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow$ *index-valid as s* $\Longrightarrow$ *assert-bound a s* $= s' \Longrightarrow \mathcal{U}\ s' \Longrightarrow$ *minimal-unsat-state-core* s' **and**

assert-bound-index-valid: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow$ *index-valid as s* $\Longrightarrow$ *index-valid* (*insert a as*) (*assert-bound a s*)

begin
lemma *assert-bound-tableau-id*: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow \mathcal{T}$ (*assert-bound a s*) $= \mathcal{T}\ s$
⟨*proof*⟩

lemma *assert-bound-tableau-valuated*: $[\![\neg\ \mathcal{U}\ s;\ \models\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s]\!] \Longrightarrow \nabla$ (*assert-bound a s*)
⟨*proof*⟩

end

locale *AssertBoundNoLhs* =
fixes *assert-bound* :: ('*i*,'*a*::*lrv*) *i-atom* ⇒ ('*i*,'*a*) *state* ⇒ ('*i*,'*a*) *state*
assumes *assert-bound-nolhs-tableau-id*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow \mathcal{T}$ (*assert-bound a s*) $= \mathcal{T}\ s$
assumes *assert-bound-nolhs-sat*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow$
$\neg\ \mathcal{U}$ (*assert-bound a s*) $\Longrightarrow\ \models_{nolhs}$ (*assert-bound a s*) $\wedge\ \Diamond$ (*assert-bound a s*)
assumes *assert-bound-nolhs-atoms-equiv-bounds*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow$
flat ats $\doteq \mathcal{B}\ s \Longrightarrow$ *flat* $(ats \cup \{a\}) \doteq \mathcal{B}$ (*assert-bound a s*)
assumes *assert-bound-nolhs-atoms-imply-bounds-index*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow$
ats $\models_i \mathcal{BI}\ s \Longrightarrow$ *insert a ats* $\models_i \mathcal{BI}$ (*assert-bound a s*)
assumes *assert-bound-nolhs-unsat*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow$
index-valid as s $\Longrightarrow \mathcal{U}$ (*assert-bound a s*) $\Longrightarrow$ *minimal-unsat-state-core* (*assert-bound a s*)
assumes *assert-bound-nolhs-tableau-valuated*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow$
∇ (*assert-bound a s*)
assumes *assert-bound-nolhs-index-valid*: $[\![\neg\ \mathcal{U}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \Diamond\ s]\!] \Longrightarrow$
index-valid as s $\Longrightarrow$ *index-valid* (*insert a as*) (*assert-bound a s*)

sublocale *AssertBoundNoLhs* < *AssertBound*

⟨*proof*⟩

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

locale *Check* = **fixes** $check::('i,'a::lrv)\ state \Rightarrow ('i,'a)\ state$
assumes
— If *check* is called from an inconsistent state, the state is unchanged.

check-unsat-id: $\mathcal{U}\ s \Longrightarrow check\ s = s$ **and**

— The tableau remains equivalent to the previous one, normalized and valuated, the state stays consistent.

check-tableau: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs}\ s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow$
$let\ s' = check\ s\ in\ ((v::'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ s') \wedge \triangle\ (\mathcal{T}\ s') \wedge \nabla\ s' \wedge \models_{nolhs}\ s' \wedge \Diamond\ s'$ **and**

— The bounds remain unchanged.

check-bounds-id: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs}\ s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow \mathcal{B}_i\ (check\ s) = \mathcal{B}_i\ s$ **and**

— If $\mathcal{U}$ flag is not raised, the current valuation $\mathcal{V}$ satisfies both the tableau and the bounds and if it is raised, there is no valuation that satisfies them.

check-sat: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs}\ s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow \neg\ \mathcal{U}\ (check\ s) \Longrightarrow \models\ (check\ s)$ **and**

check-unsat: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs}\ s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow \mathcal{U}\ (check\ s) \Longrightarrow$ *minimal-unsat-state-core* $(check\ s)$

begin

lemma *check-tableau-equiv*: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs}\ s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow$
$(v::'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ (check\ s)$
⟨*proof*⟩

lemma *check-tableau-index-valid*: **assumes** $\neg\ \mathcal{U}\ s\ \ \models_{nolhs}\ s\ \Diamond\ s\ \triangle\ (\mathcal{T}\ s)\ \nabla\ s$
shows *index-valid as* $(check\ s)$ = *index-valid as s*
⟨*proof*⟩

lemma *check-tableau-normalized*: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs}\ s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow \triangle\ (\mathcal{T}\ (check\ s))$

⟨*proof*⟩

lemma *check-bounds-consistent*: **assumes** $\neg\ \mathcal{U}\ s \models_{nolhs} s\ \Diamond\ s\ \triangle\ (\mathcal{T}\ s)\ \nabla\ s$
shows $\Diamond$ (*check s*)
⟨*proof*⟩

lemma *check-tableau-valuated*: $[\![\neg\ \mathcal{U}\ s; \models_{nolhs} s; \Diamond\ s; \triangle\ (\mathcal{T}\ s); \nabla\ s]\!] \Longrightarrow \nabla$ (*check s*)
⟨*proof*⟩

lemma *check-indices-state*: **assumes** $\neg\ \mathcal{U}\ s \Longrightarrow\ \models_{nolhs} s\ \neg\ \mathcal{U}\ s \Longrightarrow \Diamond\ s\ \neg\ \mathcal{U}\ s \Longrightarrow \triangle\ (\mathcal{T}\ s)\ \neg\ \mathcal{U}\ s \Longrightarrow \nabla\ s$
shows *indices-state* (*check s*) = *indices-state s*
⟨*proof*⟩

lemma *check-distinct-indices-state*: **assumes** $\neg\ \mathcal{U}\ s \Longrightarrow\ \models_{nolhs} s\ \neg\ \mathcal{U}\ s \Longrightarrow \Diamond\ s\ \neg\ \mathcal{U}\ s \Longrightarrow \triangle\ (\mathcal{T}\ s)\ \neg\ \mathcal{U}\ s \Longrightarrow \nabla\ s$
shows *distinct-indices-state* (*check s*) = *distinct-indices-state s*
⟨*proof*⟩

end

locale *Assert′* = *AssertBound assert-bound* + *Check check* **for**
assert-bound :: ('*i*,'*a*::*lrv*) *i-atom* ⇒ ('*i*,'*a*) *state* ⇒ ('*i*,'*a*) *state* **and**
check :: ('*i*,'*a*::*lrv*) *state* ⇒ ('*i*,'*a*) *state*
begin
definition *assert* :: ('*i*,'*a*) *i-atom* ⇒ ('*i*,'*a*) *state* ⇒ ('*i*,'*a*) *state* **where**
assert a s ≡ *check* (*assert-bound a s*)

lemma *Assert′Precond*:
assumes $\neg\ \mathcal{U}\ s \models s\ \triangle\ (\mathcal{T}\ s)\ \nabla\ s$
shows
$\triangle$ ($\mathcal{T}$ (*assert-bound a s*))
$\neg\ \mathcal{U}$ (*assert-bound a s*) $\Longrightarrow\ \models_{nolhs}$ (*assert-bound a s*) ∧ $\Diamond$ (*assert-bound a s*)
∇ (*assert-bound a s*)
⟨*proof*⟩
end

sublocale *Assert′* < *Assert assert*
⟨*proof*⟩

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

locale *AssertAllState″* = *Init init* + *AssertBoundNoLhs assert-bound* + *Check check* **for**
init :: *tableau* ⇒ ('*i*,'*a*::*lrv*) *state* **and**

assert-bound :: $('i,'a::lrv)$ *i-atom* $\Rightarrow$ $('i,'a)$ *state* $\Rightarrow$ $('i,'a)$ *state* **and**
check :: $('i,'a::lrv)$ *state* $\Rightarrow$ $('i,'a)$ *state*
begin
definition *assert-bound-loop* **where**
assert-bound-loop ats s $\equiv$ *foldl* $(\lambda\ s'\ a.\ if\ (\mathcal{U}\ s')\ then\ s'\ else\ assert\text{-}bound\ a\ s')\ s$ *ats*
definition *assert-all-state* **where** [*simp*]:
assert-all-state t ats $\equiv$ *check* (*assert-bound-loop ats* (*init t*))

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

assert-bound-loop ats s $\equiv$ *foldl* $(\lambda s'\ a.\ if\ \mathcal{U}\ s'\ then\ s'\ else\ assert\text{-}bound\ a\ s')\ s$ *ats*

assert-all-state t ats $\equiv$ *check* (*assert-bound-loop ats* (*init t*))

Then, the loop consists only of *assert-bound* calls, so *assert-bound* postcondition must imply its precondition. This is not the case, since variables on the lhs may be out of their bounds. Therefore, we make a refinement and specify weaker preconditions (replace $\models s$, by $\models_{nolhs}\ s$ and $\lozenge\ s$) for *assert-bound*, and show that these preconditions are still good enough to prove the correctnes of this alternative *assert-all-state* definition.

lemma *AssertAllState''-precond'*:
assumes $\triangle\ (\mathcal{T}\ s)\ \nabla\ s\ \neg\ \mathcal{U}\ s \longrightarrow\ \models_{nolhs}\ s \wedge \lozenge\ s$
shows *let* s' = *assert-bound-loop ats s in*
$\triangle\ (\mathcal{T}\ s') \wedge \nabla\ s' \wedge (\neg\ \mathcal{U}\ s' \longrightarrow\ \models_{nolhs}\ s' \wedge \lozenge\ s')$
⟨*proof*⟩

lemma *AssertAllState''-precond*:
assumes $\triangle\ t$
shows *let* s' = *assert-bound-loop ats* (*init t*) *in*
$\triangle\ (\mathcal{T}\ s') \wedge \nabla\ s' \wedge (\neg\ \mathcal{U}\ s' \longrightarrow\ \models_{nolhs}\ s' \wedge \lozenge\ s')$
⟨*proof*⟩

lemma *AssertAllState''Induct*:
assumes
$\triangle\ t$
$P\ \{\}\ (init\ t)$
$\bigwedge\ as\ bs\ t.\ as \subseteq bs \Longrightarrow P\ as\ t \Longrightarrow P\ bs\ t$
$\bigwedge\ s\ a\ ats.\ [\![\neg\ \mathcal{U}\ s;\ \ \langle \mathcal{V}\ s\rangle \models_t \mathcal{T}\ s;\ \models_{nolhs}\ s;\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ \lozenge\ s;\ P\ (set\ ats)\ s;$ $index\text{-}valid\ (set\ ats)\ s]\!]$
$\Longrightarrow P\ (insert\ a\ (set\ ats))\ (assert\text{-}bound\ a\ s)$
shows P (*set ats*) (*assert-bound-loop ats* (*init t*))
⟨*proof*⟩

lemma *AssertAllState''-tableau-id*:
$\triangle\ t \Longrightarrow \mathcal{T}\ (assert\text{-}bound\text{-}loop\ ats\ (init\ t)) = \mathcal{T}\ (init\ t)$
⟨*proof*⟩

lemma *AssertAllState''-sat*:
$\triangle\ t \Longrightarrow$
$\neg\ \mathcal{U}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t)) \longrightarrow\ \models_{nolhs}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t))$
$\wedge\ \Diamond\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t))$
⟨*proof*⟩

lemma *AssertAllState''-unsat*:
$\triangle\ t \Longrightarrow \mathcal{U}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t)) \longrightarrow \textit{minimal-unsat-state-core}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t))$
⟨*proof*⟩

lemma *AssertAllState''-sat-atoms-equiv-bounds*:
$\triangle\ t \Longrightarrow \neg\ \mathcal{U}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t)) \longrightarrow \textit{flat}\ (\textit{set ats}) \doteq \mathcal{B}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t))$
⟨*proof*⟩

lemma *AssertAllState''-atoms-imply-bounds-index*:
assumes $\triangle\ t$
shows $\textit{set ats} \models_i \mathcal{BI}\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t))$
⟨*proof*⟩

lemma *AssertAllState''-index-valid*:
$\triangle\ t \Longrightarrow \textit{index-valid}\ (\textit{set ats})\ (\textit{assert-bound-loop ats}\ (\textit{init}\ t))$
⟨*proof*⟩

end

sublocale *AssertAllState''* < *AssertAllState assert-all-state*
⟨*proof*⟩

6.6 Update and Pivot

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

locale *Update* = **fixes** $\textit{update}::\textit{var} \Rightarrow {'a}::\textit{lrv} \Rightarrow ({'i},{'a})\ \textit{state} \Rightarrow ({'i},{'a})\ \textit{state}$
assumes
— Tableau, bounds, and the unsatisfiability flag are preserved.

update-id: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin \textit{lvars}\ (\mathcal{T}\ s)]\!] \Longrightarrow$
$\textit{let}\ s' = \textit{update}\ x\ c\ s\ \textit{in}\ \mathcal{T}\ s' = \mathcal{T}\ s \wedge \mathcal{B}_i\ s' = \mathcal{B}_i\ s \wedge \mathcal{U}\ s' = \mathcal{U}\ s \wedge \mathcal{U}_c\ s' = \mathcal{U}_c\ s$ **and**

— Tableau remains valuated.

update-tableau-valuated: $\llbracket \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin lvars\ (\mathcal{T}\ s)\rrbracket \Longrightarrow \nabla\ (update\ x\ v\ s)$ **and**

— The given variable x in the updated valuation is set to the given value v while all other variables (except those on the lhs of the tableau) are unchanged.

update-valuation-nonlhs: $\llbracket \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin lvars\ (\mathcal{T}\ s)\rrbracket \Longrightarrow x' \notin lvars\ (\mathcal{T}\ s) \longrightarrow look\ (\mathcal{V}\ (update\ x\ v\ s))\ x' = (if\ x = x'\ then\ Some\ v\ else\ look\ (\mathcal{V}\ s)\ x')$ **and**

— Updated valuation continues to satisfy the tableau.

update-satisfies-tableau: $\llbracket \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin lvars\ (\mathcal{T}\ s)\rrbracket \Longrightarrow \langle \mathcal{V}\ s\rangle \models_t \mathcal{T}\ s \longrightarrow \langle \mathcal{V}\ (update\ x\ c\ s)\rangle \models_t \mathcal{T}\ s$

begin
lemma *update-bounds-id*:
  assumes $\triangle\ (\mathcal{T}\ s)\ \nabla\ s\ x \notin lvars\ (\mathcal{T}\ s)$
  shows $\mathcal{B}_i\ (update\ x\ c\ s) = \mathcal{B}_i\ s$
    $\mathcal{BI}\ (update\ x\ c\ s) = \mathcal{BI}\ s$
    $\mathcal{B}_l\ (update\ x\ c\ s) = \mathcal{B}_l\ s$
    $\mathcal{B}_u\ (update\ x\ c\ s) = \mathcal{B}_u\ s$
  ⟨*proof*⟩

lemma *update-indices-state-id*:
  assumes $\triangle\ (\mathcal{T}\ s)\ \nabla\ s\ x \notin lvars\ (\mathcal{T}\ s)$
  shows $indices\text{-}state\ (update\ x\ c\ s) = indices\text{-}state\ s$
  ⟨*proof*⟩

lemma *update-tableau-id*: $\llbracket \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin lvars\ (\mathcal{T}\ s)\rrbracket \Longrightarrow \mathcal{T}\ (update\ x\ c\ s) = \mathcal{T}\ s$
  ⟨*proof*⟩

lemma *update-unsat-id*: $\llbracket \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin lvars\ (\mathcal{T}\ s)\rrbracket \Longrightarrow \mathcal{U}\ (update\ x\ c\ s) = \mathcal{U}\ s$
  ⟨*proof*⟩

lemma *update-unsat-core-id*: $\llbracket \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x \notin lvars\ (\mathcal{T}\ s)\rrbracket \Longrightarrow \mathcal{U}_c\ (update\ x\ c\ s) = \mathcal{U}_c\ s$
  ⟨*proof*⟩

definition *assert-bound′* **where**
  [*simp*]: $assert\text{-}bound'\ dir\ i\ x\ c\ s \equiv$
    $(if\ (\trianglerighteq_{ub}\ (lt\ dir))\ c\ (UB\ dir\ s\ x)\ then\ s$
      $else\ let\ s' = update\mathcal{BI}\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s\ in$
        $if\ (\triangleleft_{lb}\ (lt\ dir))\ c\ ((LB\ dir)\ s\ x)\ then$
          $set\text{-}unsat\ [i,\ ((LI\ dir)\ s\ x)]\ s'$
        $else\ if\ x \notin lvars\ (\mathcal{T}\ s') \wedge (lt\ dir)\ c\ (\langle \mathcal{V}\ s\rangle\ x)\ then$
          $update\ x\ c\ s'$
        $else$

s')

fun *assert-bound* :: ('i,'a::lrv) *i-atom* ⇒ ('i,'a) *state* ⇒ ('i,'a) *state* **where**
assert-bound (i,*Leq x c*) *s* = *assert-bound' Positive i x c s*
| *assert-bound* (i,*Geq x c*) *s* = *assert-bound' Negative i x c s*

lemma *assert-bound'-cases*:
assumes ⟦$\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)⟧ ⟹ *P s*
assumes ⟦¬ ($\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)); $\triangleleft_{lb}$ (*lt dir*) *c* ((*LB dir*) *s x*)⟧ ⟹
P (*set-unsat* [*i*, ((*LI dir*) *s x*)] (*update*$\mathcal{BI}$ (*UBI-upd dir*) *i x c s*))
assumes ⟦*x* ∉ *lvars* ($\mathcal{T}$ *s*); (*lt dir*) *c* (⟨$\mathcal{V}$ *s*⟩ *x*); ¬ ($\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)); ¬ ($\triangleleft_{lb}$ (*lt dir*) *c* ((*LB dir*) *s x*))⟧ ⟹
P (*update x c* (*update*$\mathcal{BI}$ (*UBI-upd dir*) *i x c s*))
assumes ⟦¬ ($\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)); ¬ ($\triangleleft_{lb}$ (*lt dir*) *c* ((*LB dir*) *s x*)); *x* ∈ *lvars* ($\mathcal{T}$ *s*)⟧ ⟹
P (*update*$\mathcal{BI}$ (*UBI-upd dir*) *i x c s*)
assumes ⟦¬ ($\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)); ¬ ($\triangleleft_{lb}$ (*lt dir*) *c* ((*LB dir*) *s x*)); ¬ ((*lt dir*) *c* (⟨$\mathcal{V}$ *s*⟩ *x*))⟧ ⟹
P (*update*$\mathcal{BI}$ (*UBI-upd dir*) *i x c s*)
assumes *dir* = *Positive* ∨ *dir* = *Negative*
shows *P* (*assert-bound' dir i x c s*)
⟨*proof*⟩

lemma *assert-bound-cases*:
assumes ⋀ *c x dir*.
⟦ *dir* = *Positive* ∨ *dir* = *Negative*;
a = *LE dir x c*;
$\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)
⟧ ⟹
P' (*lt dir*) (*UBI dir*) (*LBI dir*) (*UB dir*) (*LB dir*) (*UBI-upd dir*) (*UI dir*) (*LI dir*) (*LE dir*) (*GE dir*) *s*
assumes ⋀ *c x dir*.
⟦*dir* = *Positive* ∨ *dir* = *Negative*;
a = *LE dir x c*;
¬ $\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*); $\triangleleft_{lb}$ (*lt dir*) *c* ((*LB dir*) *s x*)
⟧ ⟹
P' (*lt dir*) (*UBI dir*) (*LBI dir*) (*UB dir*) (*LB dir*) (*UBI-upd dir*) (*UI dir*) (*LI dir*) (*LE dir*) (*GE dir*)
(*set-unsat* [*i*, ((*LI dir*) *s x*)] (*update*$\mathcal{BI}$ (*UBI-upd dir*) *i x c s*))
assumes ⋀ *c x dir*.
⟦ *dir* = *Positive* ∨ *dir* = *Negative*;
a = *LE dir x c*;
x ∉ *lvars* ($\mathcal{T}$ *s*); (*lt dir*) *c* (⟨$\mathcal{V}$ *s*⟩ *x*);
¬ ($\trianglerighteq_{ub}$ (*lt dir*) *c* ((*UB dir*) *s x*)); ¬ ($\triangleleft_{lb}$ (*lt dir*) *c* ((*LB dir*) *s x*))
⟧ ⟹
P' (*lt dir*) (*UBI dir*) (*LBI dir*) (*UB dir*) (*LB dir*) (*UBI-upd dir*) (*UI dir*) (*LI dir*) (*LE dir*) (*GE dir*)
(*update x c* ((*update*$\mathcal{BI}$ (*UBI-upd dir*) *i x c s*)))
assumes ⋀ *c x dir*.

⟦ $dir = Positive \vee dir = Negative$;
$a = LE\ dir\ x\ c$;
$x \in lvars\ (\mathcal{T}\ s)$; $\neg\ (\trianglerighteq_{ub}\ (lt\ dir)\ c\ ((UB\ dir)\ s\ x))$;
$\neg\ (\triangleleft_{lb}\ (lt\ dir)\ c\ ((LB\ dir)\ s\ x))$
⟧ $\Longrightarrow$
$P'\ (lt\ dir)\ (UBI\ dir)\ (LBI\ dir)\ (UB\ dir)\ (LB\ dir)\ (UBI\text{-}upd\ dir)\ (UI\ dir)$
$(LI\ dir)\ (LE\ dir)\ (GE\ dir)$
$((updateBI\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s))$

assumes $\bigwedge\ c\ x\ dir.$
⟦ $dir = Positive \vee dir = Negative$;
$a = LE\ dir\ x\ c$;
$\neg\ (\trianglerighteq_{ub}\ (lt\ dir)\ c\ ((UB\ dir)\ s\ x))$; $\neg\ (\triangleleft_{lb}\ (lt\ dir)\ c\ ((LB\ dir)\ s\ x))$;
$\neg\ ((lt\ dir)\ c\ (\langle \mathcal{V}\ s\rangle\ x))$
⟧ $\Longrightarrow$
$P'\ (lt\ dir)\ (UBI\ dir)\ (LBI\ dir)\ (UB\ dir)\ (LB\ dir)\ (UBI\text{-}upd\ dir)\ (UI\ dir)$
$(LI\ dir)\ (LE\ dir)\ (GE\ dir)$
$((updateBI\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s))$

assumes $\bigwedge\ s.\ P\ s = P'\ (>)\ \mathcal{B}_{il}\ \mathcal{B}_{iu}\ \mathcal{B}_l\ \mathcal{B}_u\ \mathcal{B}_{il}\text{-}update\ \mathcal{I}_l\ \mathcal{I}_u\ Geq\ Leq\ s$
assumes $\bigwedge\ s.\ P\ s = P'\ (<)\ \mathcal{B}_{iu}\ \mathcal{B}_{il}\ \mathcal{B}_u\ \mathcal{B}_l\ \mathcal{B}_{iu}\text{-}update\ \mathcal{I}_u\ \mathcal{I}_l\ Leq\ Geq\ s$
shows $P\ (assert\text{-}bound\ (i,a)\ s)$
⟨*proof*⟩
end

lemma *set-unsat-bounds-id*: $\mathcal{B}\ (set\text{-}unsat\ I\ s) = \mathcal{B}\ s$
⟨*proof*⟩

lemma *decrease-ub-satisfied-inverse*:
assumes *lt*: $\triangleleft_{ub}\ (lt\ dir)\ c\ \ (UB\ dir\ s\ x)$ **and** *dir*: $dir = Positive \vee dir = Negative$
assumes *v*: $v \models_b \mathcal{B}\ (updateBI\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s)$
shows $v \models_b \mathcal{B}\ s$
⟨*proof*⟩

lemma *atoms-equiv-bounds-extend*:
fixes *x c dir*
assumes $dir = Positive \vee dir = Negative\ \ \neg\ \trianglerighteq_{ub}\ (lt\ dir)\ c\ (UB\ dir\ s\ x)\ \ ats \doteq \mathcal{B}\ s$
shows $(ats \cup \{LE\ dir\ x\ c\}) \doteq \mathcal{B}\ (updateBI\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s)$
⟨*proof*⟩

lemma *bounds-updates*: $\mathcal{B}_l\ (\mathcal{B}_{iu}\text{-}update\ u\ s) = \mathcal{B}_l\ s$
$\mathcal{B}_u\ (\mathcal{B}_{il}\text{-}update\ u\ s) = \mathcal{B}_u\ s$
$\mathcal{B}_u\ (\mathcal{B}_{iu}\text{-}update\ (upd\ x\ (i,c))\ s) = (\mathcal{B}_u\ s)\ (x \mapsto c)$
$\mathcal{B}_l\ (\mathcal{B}_{il}\text{-}update\ (upd\ x\ (i,c))\ s) = (\mathcal{B}_l\ s)\ (x \mapsto c)$
⟨*proof*⟩

locale *EqForLVar* =
fixes *eq-idx-for-lvar* :: $tableau \Rightarrow var \Rightarrow nat$

assumes *eq-idx-for-lvar*:
$[\![x \in \textit{lvars } t]\!] \Longrightarrow \textit{eq-idx-for-lvar } t\ x < \textit{length } t \wedge \textit{lhs } (t\ !\ \textit{eq-idx-for-lvar } t\ x) = x$
begin
definition *eq-for-lvar* :: *tableau* ⇒ *var* ⇒ *eq* **where**
$\textit{eq-for-lvar } t\ v \equiv t\ !\ (\textit{eq-idx-for-lvar } t\ v)$
lemma *eq-for-lvar*:
$[\![x \in \textit{lvars } t]\!] \Longrightarrow \textit{eq-for-lvar } t\ x \in \textit{set } t \wedge \textit{lhs } (\textit{eq-for-lvar } t\ x) = x$
⟨*proof*⟩

abbreviation *rvars-of-lvar* **where**
$\textit{rvars-of-lvar } t\ x \equiv \textit{rvars-eq } (\textit{eq-for-lvar } t\ x)$

lemma *rvars-of-lvar-rvars*:
assumes $x \in \textit{lvars } t$
shows $\textit{rvars-of-lvar } t\ x \subseteq \textit{rvars } t$
⟨*proof*⟩

end

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

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

begin

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

abbreviation *update-eq* **where**
$\textit{update-eq } v\ x\ c\ v'\ e \equiv \textit{upd } (\textit{lhs } e)\ (\textit{rhs-eq-val } v\ x\ c\ e)\ v'$

definition *update* :: *var* ⇒ *'a* ⇒ (*'i*,*'a*) *state* ⇒ (*'i*,*'a*) *state* **where**
$\textit{update } x\ c\ s \equiv \mathcal{V}\textit{-update } (\textit{upd } x\ c\ (\textit{foldl } (\textit{update-eq } (\mathcal{V}\ s)\ x\ c)\ (\mathcal{V}\ s)\ (\mathcal{T}\ s)))\ s$

lemma *update-no-set-none*:
shows $\textit{look } (\mathcal{V}\ s)\ y \neq \textit{None} \Longrightarrow$
$\textit{look } (\textit{foldl } (\textit{update-eq } (\mathcal{V}\ s)\ x\ v)\ (\mathcal{V}\ s)\ t)\ y \neq \textit{None}$
⟨*proof*⟩

lemma *update-no-left*:
assumes $y \notin \textit{lvars } t$
shows $\textit{look } (\mathcal{V}\ s)\ y = \textit{look } (\textit{foldl } (\textit{update-eq } (\mathcal{V}\ s)\ x\ v)\ (\mathcal{V}\ s)\ t)\ y$
⟨*proof*⟩

lemma *update-left*:
assumes $y \in \textit{lvars } t$
shows $\exists\ eq \in \textit{set } t.\ \textit{lhs } eq = y \wedge$

$look\ (foldl\ (update\text{-}eq\ (\mathcal{V}\ s)\ x\ v)\ (\mathcal{V}\ s)\ t)\ y = Some\ (rhs\text{-}eq\text{-}val\ (\mathcal{V}\ s)\ x\ v\ eq)$
⟨*proof*⟩

lemma *update-valuate-rhs*:
assumes $e \in set\ (\mathcal{T}\ s)\ \triangle\ (\mathcal{T}\ s)$
shows $rhs\ e\ \{\!\!\{\ \langle \mathcal{V}\ (update\ x\ c\ s)\rangle\ \}\!\!\} = rhs\ e\ \{\!\!\{\ \langle \mathcal{V}\ s\rangle\ (x := c)\ \}\!\!\}$
⟨*proof*⟩

end

sublocale *RhsEqVal* < *Update update*
⟨*proof*⟩

To update the valuation for a variable that is on the lhs of the tableau it should first be swapped with some rhs variable of its equation, in an operation called *pivoting*. Pivoting has the precondition that the tableau is normalized and that it is always called for a lhs variable of the tableau, and a rhs variable in the equation with that lhs variable. The set of rhs variables for the given lhs variable is found using the *rvars-of-lvar* function (specified in a very simple locale *EqForLVar*, that we do not print).

locale *Pivot* = *EqForLVar* + **fixes** $pivot{::}var \Rightarrow var \Rightarrow ('i,'a{::}lrv)\ state \Rightarrow ('i,'a)\ state$
assumes
— Valuation, bounds, and the unsatisfiability flag are not changed.

pivot-id: $[\![\triangle\ (\mathcal{T}\ s);\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$let\ s' = pivot\ x_i\ x_j\ s\ in\ \mathcal{V}\ s' = \mathcal{V}\ s \wedge \mathcal{B}_i\ s' = \mathcal{B}_i\ s \wedge \mathcal{U}\ s' = \mathcal{U}\ s \wedge \mathcal{U}_c\ s' = \mathcal{U}_c\ s$ **and**

— The tableau remains equivalent to the previous one and normalized.

pivot-tableau: $[\![\triangle\ (\mathcal{T}\ s);\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$let\ s' = pivot\ x_i\ x_j\ s\ in\ \ ((v{::}'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ s') \wedge \triangle\ (\mathcal{T}\ s')$ **and**

— x_i and x_j are swapped, while the other variables do not change sides.

pivot-vars': $[\![\triangle\ (\mathcal{T}\ s);\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow let\ s' = pivot\ x_i\ x_j\ s\ in$
$rvars(\mathcal{T}\ s') = rvars(\mathcal{T}\ s) - \{x_j\} \cup \{x_i\}\ \ \wedge\ \ lvars(\mathcal{T}\ s') = lvars(\mathcal{T}\ s) - \{x_i\} \cup \{x_j\}$

begin
lemma *pivot-bounds-id*: $[\![\triangle\ (\mathcal{T}\ s);\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\mathcal{B}_i\ (pivot\ x_i\ x_j\ s) = \mathcal{B}_i\ s$
⟨*proof*⟩

lemma *pivot-bounds-id'*: **assumes** $\triangle\ (\mathcal{T}\ s)\ x_i \in lvars\ (\mathcal{T}\ s)\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}$

s) x_i
shows $\mathcal{BI}$ (*pivot* x_i x_j *s*) = $\mathcal{BI}$ *s* $\mathcal{B}$ (*pivot* x_i x_j *s*) = $\mathcal{B}$ *s* $\mathcal{I}$ (*pivot* x_i x_j *s*) = $\mathcal{I}$ *s*
⟨*proof*⟩

lemma *pivot-valuation-id*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ $\mathcal{V}$ (*pivot* x_i x_j *s*) = $\mathcal{V}$ *s*
⟨*proof*⟩

lemma *pivot-unsat-id*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ $\mathcal{U}$ (*pivot* x_i x_j *s*) = $\mathcal{U}$ *s*
⟨*proof*⟩

lemma *pivot-unsat-core-id*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ $\mathcal{U}_c$ (*pivot* x_i x_j *s*) = $\mathcal{U}_c$ *s*
⟨*proof*⟩

lemma *pivot-tableau-equiv*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹
(*v*::*'a valuation*) $\models_t$ $\mathcal{T}$ *s* = *v* $\models_t$ $\mathcal{T}$ (*pivot* x_i x_j *s*)
⟨*proof*⟩

lemma *pivot-tableau-normalized*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ △ ($\mathcal{T}$ (*pivot* x_i x_j *s*))
⟨*proof*⟩

lemma *pivot-rvars*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ *rvars* ($\mathcal{T}$ (*pivot* x_i x_j *s*)) = *rvars* ($\mathcal{T}$ *s*) − {x_j} ∪ {x_i}
⟨*proof*⟩

lemma *pivot-lvars*: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ *lvars* ($\mathcal{T}$ (*pivot* x_i x_j *s*)) = *lvars* ($\mathcal{T}$ *s*) − {x_i} ∪ {x_j}
⟨*proof*⟩

lemma *pivot-vars*:
⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i⟧ ⟹ *tvars* ($\mathcal{T}$ (*pivot* x_i x_j *s*)) = *tvars* ($\mathcal{T}$ *s*)
⟨*proof*⟩

lemma
pivot-tableau-valuated: ⟦△ ($\mathcal{T}$ *s*); x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-of-lvar* ($\mathcal{T}$ *s*) x_i; ∇ *s*⟧ ⟹ ∇ (*pivot* x_i x_j *s*)
⟨*proof*⟩

end

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

locale *PivotAndUpdate* = *EqForLVar* +
fixes *pivot-and-update* :: *var* ⇒ *var* ⇒ *'a::lrv* ⇒ *('i,'a) state* ⇒ *('i,'a) state*
assumes *pivotandupdate-unsat-id*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\mathcal{U}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \mathcal{U}\ s$
assumes *pivotandupdate-unsat-core-id*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\mathcal{U}_c\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \mathcal{U}_c\ s$
assumes *pivotandupdate-bounds-id*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\mathcal{B}_i\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \mathcal{B}_i\ s$
assumes *pivotandupdate-tableau-normalized*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\triangle\ (\mathcal{T}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s))$
assumes *pivotandupdate-tableau-equiv*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$(v::'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s)$
assumes *pivotandupdate-satisfies-tableau*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\langle\mathcal{V}\ s\rangle \models_t \mathcal{T}\ s \longrightarrow \langle\mathcal{V}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s)\rangle \models_t \mathcal{T}\ s$
assumes *pivotandupdate-rvars*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$rvars\ (\mathcal{T}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s)) = rvars\ (\mathcal{T}\ s) - \{x_j\} \cup \{x_i\}$
assumes *pivotandupdate-lvars*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$lvars\ (\mathcal{T}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s)) = lvars\ (\mathcal{T}\ s) - \{x_i\} \cup \{x_j\}$
assumes *pivotandupdate-valuation-nonlhs*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$x \notin lvars\ (\mathcal{T}\ s) - \{x_i\} \cup \{x_j\} \longrightarrow look\ (\mathcal{V}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s))\ x = (if\ x = x_i\ then\ Some\ c\ else\ look\ (\mathcal{V}\ s)\ x)$
assumes *pivotandupdate-tableau-valuated*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow$
$\nabla\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s)$
begin

lemma *pivotandupdate-bounds-id'*: **assumes** $\triangle\ (\mathcal{T}\ s)\ \nabla\ s\ x_i \in lvars\ (\mathcal{T}\ s)\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i$
shows $\mathcal{BI}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \mathcal{BI}\ s$
$\mathcal{B}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \mathcal{B}\ s$
$\mathcal{I}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \mathcal{I}\ s$
⟨*proof*⟩

lemma *pivotandupdate-valuation-xi*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i]\!] \Longrightarrow look\ (\mathcal{V}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s))\ x_i = Some\ c$
⟨*proof*⟩

lemma *pivotandupdate-valuation-other-nolhs*: $[\![\triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i;\ x \notin lvars\ (\mathcal{T}\ s);\ x \neq x_j]\!] \Longrightarrow look\ (\mathcal{V}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s))\ x = look\ (\mathcal{V}\ s)\ x$
⟨*proof*⟩

lemma *pivotandupdate-nolhs*:
$[\![\ \triangle\ (\mathcal{T}\ s);\ \nabla\ s;\ x_i \in lvars\ (\mathcal{T}\ s);\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i;$
$\models_{nolhs}\ s;\ \Diamond\ s;\ \mathcal{B}_l\ s\ x_i = Some\ c \vee \mathcal{B}_u\ s\ x_i = Some\ c]\!] \Longrightarrow$
$\models_{nolhs}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s)$
⟨*proof*⟩

lemma *pivotandupdate-bounds-consistent*:
assumes $\triangle\ (\mathcal{T}\ s)\ \nabla\ s\ x_i \in lvars\ (\mathcal{T}\ s)\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i$
shows $\Diamond\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s) = \Diamond\ s$
⟨*proof*⟩
end

locale *PivotUpdate* = *Pivot eq-idx-for-lvar pivot* + *Update update* **for**
eq-idx-for-lvar :: *tableau* ⇒ *var* ⇒ *nat* **and**
pivot :: *var* ⇒ *var* ⇒ (*'i,'a::lrv*) *state* ⇒ (*'i,'a*) *state* **and**
update :: *var* ⇒ *'a* ⇒ (*'i,'a*) *state* ⇒ (*'i,'a*) *state*
begin
definition *pivot-and-update* :: *var* ⇒ *var* ⇒ *'a* ⇒ (*'i,'a*) *state* ⇒ (*'i,'a*) *state*
where [*simp*]:
$pivot\text{-}and\text{-}update\ x_i\ x_j\ c\ s \equiv update\ x_i\ c\ (pivot\ x_i\ x_j\ s)$

lemma *pivot-update-precond*:
assumes $\triangle\ (\mathcal{T}\ s)\ x_i \in lvars\ (\mathcal{T}\ s)\ x_j \in rvars\text{-}of\text{-}lvar\ (\mathcal{T}\ s)\ x_i$
shows $\triangle\ (\mathcal{T}\ (pivot\ x_i\ x_j\ s))\ x_i \notin lvars\ (\mathcal{T}\ (pivot\ x_i\ x_j\ s))$
⟨*proof*⟩

end

sublocale *PivotUpdate* < *PivotAndUpdate eq-idx-for-lvar pivot-and-update*
⟨*proof*⟩

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

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

The case of *Geq x c* atoms is analogous (a systematic way to avoid symmetries is discussed in Section 6.8). This implementation satisfies both its specifications.

lemma *indices-state-set-unsat*: *indices-state* (*set-unsat I s*) = *indices-state s*
⟨*proof*⟩

lemma $\mathcal{BI}$*-set-unsat*: $\mathcal{BI}$ (*set-unsat I s*) = $\mathcal{BI}$ *s*
⟨*proof*⟩

lemma *satisfies-tableau-cong*: **assumes** $\bigwedge x.\ x \in tvars\ t \Longrightarrow v\ x = w\ x$
shows $(v \models_t t) = (w \models_t t)$
⟨*proof*⟩

lemma *satisfying-state-valuation-to-atom-tabl*: **assumes** *J*: $J \subseteq$ *indices-state s*
and *model*: $(J,\ v) \models_{ise} s$
and *ivalid*: *index-valid as s*
and *dist*: *distinct-indices-atoms as*
shows $(J,\ v) \models_{iaes}$ *as* $v \models_t \mathcal{T}\ s$
⟨*proof*⟩

Note that in order to ensure minimality of the unsat cores, pivoting is required.

sublocale *AssertAllState* < *AssertAll assert-all*
⟨*proof*⟩

lemma (**in** *Update*) *update-to-assert-bound-no-lhs*: **assumes** *pivot*: *Pivot eqlvar* (*pivot* :: *var* ⇒ *var* ⇒ ('*i*,'*a*) *state* ⇒ ('*i*,'*a*) *state*)
shows *AssertBoundNoLhs assert-bound*
⟨*proof*⟩

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

locale *PivotEq* =
fixes *pivot-eq*::*eq* ⇒ *var* ⇒ *eq*
assumes
— Lhs var of *eq* and x_j are swapped, while the other variables do not change sides.
vars-pivot-eq:
⟦$x_j \in$ *rvars-eq eq*; *lhs eq* ∉ *rvars-eq eq* ⟧ ⟹ *let eq*′ = *pivot-eq eq* x_j *in*
lhs eq′ = x_j ∧ *rvars-eq eq*′ = {*lhs eq*} ∪ (*rvars-eq eq* − {x_j}) **and**

— Pivoting keeps the equation equisatisfiable.

equiv-pivot-eq:
⟦$x_j \in$ *rvars-eq eq*; *lhs eq* ∉ *rvars-eq eq* ⟧ ⟹
(*v*::′*a*::*lrv valuation*) $\models_e$ *pivot-eq eq* x_j ⟷ $v \models_e$ *eq*

begin

lemma *lhs-pivot-eq*:
⟦$x_j \in$ *rvars-eq eq*; *lhs eq* ∉ *rvars-eq eq* ⟧ ⟹ *lhs* (*pivot-eq eq* x_j) = x_j
⟨*proof*⟩

lemma *rvars-pivot-eq*:
$[\![x_j \in \textit{rvars-eq eq}; \textit{lhs eq} \notin \textit{rvars-eq eq}\ ]\!] \Longrightarrow \textit{rvars-eq}\ (\textit{pivot-eq eq}\ x_j) = \{\textit{lhs eq}\} \cup (\textit{rvars-eq eq} - \{x_j\})$
⟨*proof*⟩

end

abbreviation *doublesub* (- ⊆*s* - ⊆*s* - [*50*,*51*,*51*] *50*) **where**
$\textit{doublesub a b c} \equiv a \subseteq b \wedge b \subseteq c$

locale *SubstVar* =
fixes *subst-var*::*var* ⇒ *linear-poly* ⇒ *linear-poly* ⇒ *linear-poly*
assumes
— Effect of *subst-var* x_j *lp′ lp* on *lp* variables.

vars-subst-var′:
$(\textit{vars lp} - \{x_j\}) - \textit{vars lp}' \subseteq s\ \textit{vars}\ (\textit{subst-var}\ x_j\ \textit{lp}'\ \textit{lp}) \subseteq s\ (\textit{vars lp} - \{x_j\}) \cup \textit{vars lp}'$**and**

subst-no-effect: $x_j \notin \textit{vars lp} \Longrightarrow \textit{subst-var}\ x_j\ \textit{lp}'\ \textit{lp} = \textit{lp}$ **and**

subst-with-effect: $x_j \in \textit{vars lp} \Longrightarrow x \in \textit{vars lp}' - \textit{vars lp} \Longrightarrow x \in \textit{vars}\ (\textit{subst-var}\ x_j\ \textit{lp}'\ \textit{lp})$ **and**

— Effect of *subst-var* x_j *lp′ lp* on *lp* value.

equiv-subst-var:
$(v::'a :: \textit{lrv valuation})\ x_j = \textit{lp}'\ \{\!|v|\!\} \longrightarrow \textit{lp}\ \{\!|v|\!\} = (\textit{subst-var}\ x_j\ \textit{lp}'\ \textit{lp})\ \{\!|v|\!\}$

begin

lemma *vars-subst-var*:
$\textit{vars}\ (\textit{subst-var}\ x_j\ \textit{lp}'\ \textit{lp}) \subseteq (\textit{vars lp} - \{x_j\}) \cup \textit{vars lp}'$
⟨*proof*⟩

lemma *vars-subst-var-supset*:
$\textit{vars}\ (\textit{subst-var}\ x_j\ \textit{lp}'\ \textit{lp}) \supseteq (\textit{vars lp} - \{x_j\}) - \textit{vars lp}'$
⟨*proof*⟩

definition *subst-var-eq* :: *var* ⇒ *linear-poly* ⇒ *eq* ⇒ *eq* **where**
$\textit{subst-var-eq}\ v\ \textit{lp}'\ \textit{eq} \equiv (\textit{lhs eq}, \textit{subst-var}\ v\ \textit{lp}'\ (\textit{rhs eq}))$

lemma *rvars-eq-subst-var-eq*:
shows $\textit{rvars-eq}\ (\textit{subst-var-eq}\ x_j\ \textit{lp eq}) \subseteq (\textit{rvars-eq eq} - \{x_j\}) \cup \textit{vars lp}$
⟨*proof*⟩

lemma *rvars-eq-subst-var-eq-supset*:
$rvars\text{-}eq\ (subst\text{-}var\text{-}eq\ x_j\ lp\ eq) \supseteq (rvars\text{-}eq\ eq) - \{x_j\} - (vars\ lp)$
⟨*proof*⟩

lemma *equiv-subst-var-eq*:
assumes $(v{::}'a\ valuation) \models_e (x_j,\ lp')$
shows $v \models_e eq \longleftrightarrow v \models_e subst\text{-}var\text{-}eq\ x_j\ lp'\ eq$
⟨*proof*⟩
end

locale *Pivot′* = *EqForLVar* + *PivotEq* + *SubstVar*
begin
definition *pivot-tableau′* :: *var* ⇒ *var* ⇒ *tableau* ⇒ *tableau* **where**
pivot-tableau′ x_i x_j *t* ≡
  let x_i*-idx* = *eq-idx-for-lvar t* x_i; *eq* = *t* ! x_i*-idx*; *eq′* = *pivot-eq eq* x_j *in*
  map (λ *idx*. *if idx* = x_i*-idx then*
    eq′
   else
    subst-var-eq x_j (*rhs eq′*) (*t* ! *idx*)
) [*0*..<*length t*]

definition *pivot′* :: *var* ⇒ *var* ⇒ (′*i*,′*a*::*lrv*) *state* ⇒ (′*i*,′*a*) *state* **where**
pivot′ x_i x_j *s* ≡ $\mathcal{T}$*-update* (*pivot-tableau′* x_i x_j ($\mathcal{T}$ *s*)) *s*

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

definition *pivot-tableau* :: *var* ⇒ *var* ⇒ *tableau* ⇒ *tableau* **where**
pivot-tableau x_i x_j *t* ≡ *let eq* = *eq-for-lvar t* x_i; *eq′* = *pivot-eq eq* x_j *in*
  map (λ *e*. *if lhs e* = *lhs eq then eq′ else subst-var-eq* x_j (*rhs eq′*) *e*) *t*

definition *pivot* :: *var* ⇒ *var* ⇒ (′*i*,′*a*::*lrv*) *state* ⇒ (′*i*,′*a*) *state* **where**
pivot x_i x_j *s* ≡ $\mathcal{T}$*-update* (*pivot-tableau* x_i x_j ($\mathcal{T}$ *s*)) *s*

lemma *pivot-tableau′pivot-tableau*:
assumes △ *t* x_i ∈ *lvars t*
shows *pivot-tableau′* x_i x_j *t* = *pivot-tableau* x_i x_j *t*
⟨*proof*⟩

lemma *pivot′pivot*: **fixes** *s* :: (′*i*,′*a*::*lrv*)*state*
assumes △ ($\mathcal{T}$ *s*) x_i ∈ *lvars* ($\mathcal{T}$ *s*)
shows *pivot′* x_i x_j *s* = *pivot* x_i x_j *s*
⟨*proof*⟩
end

sublocale *Pivot′* < *Pivot eq-idx-for-lvar pivot*
⟨*proof*⟩

6.7 Check implementation

The *check* function is called when all rhs variables are in bounds, and it checks if there is a lhs variable that is not. If there is no such variable, then satisfiability is detected and *check* succeeds. If there is a lhs variable x_i out of its bounds, a rhs variable x_j is sought which allows pivoting with x_i and updating x_i to its violated bound. If x_i is under its lower bound it must be increased, and if x_j has a positive coefficient it must be increased so it must be under its upper bound and if it has a negative coefficient it must be decreased so it must be above its lower bound. The case when x_i is above its upper bound is symmetric (avoiding symmetries is discussed in Section 6.8). If there is no such x_j, unsatisfiability is detected and *check* fails. The procedure is recursively repeated, until it either succeeds or fails. To ensure termination, variables x_i and x_j must be chosen with respect to a fixed variable ordering. For choosing these variables auxiliary functions *min-lvar-not-in-bounds*, *min-rvar-inc* and *min-rvar-dec* are specified (each in its own locale). For, example:

locale *MinLVarNotInBounds* = **fixes** *min-lvar-not-in-bounds*::('*i*,'*a*::*lrv*) *state* ⇒ *var option*
assumes

min-lvar-not-in-bounds-None: *min-lvar-not-in-bounds s = None* ⟶ (∀ *x*∈*lvars* ($\mathcal{T}$ *s*). *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}$ *s*)) **and**

min-lvar-not-in-bounds-Some': *min-lvar-not-in-bounds s = Some* x_i ⟶ x_i∈*lvars* ($\mathcal{T}$ *s*) ∧ ¬*in-bounds* x_i ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}$ *s*)
∧ (∀ *x*∈*lvars* ($\mathcal{T}$ *s*). $x < x_i$ ⟶ *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}$ *s*))

begin
lemma *min-lvar-not-in-bounds-None'*:
min-lvar-not-in-bounds s = None ⟶ (⟨$\mathcal{V}$ *s*⟩ $\models_b$ $\mathcal{B}$ *s* ∥ *lvars* ($\mathcal{T}$ *s*))
⟨*proof*⟩

lemma *min-lvar-not-in-bounds-lvars*:*min-lvar-not-in-bounds s = Some* x_i ⟶ x_i ∈ *lvars* ($\mathcal{T}$ *s*)
⟨*proof*⟩

lemma *min-lvar-not-in-bounds-Some*: *min-lvar-not-in-bounds s = Some* x_i ⟶ ¬ *in-bounds* x_i ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}$ *s*)
⟨*proof*⟩

lemma *min-lvar-not-in-bounds-Some-min*: *min-lvar-not-in-bounds s = Some* x_i ⟶ (∀ *x* ∈ *lvars* ($\mathcal{T}$ *s*). $x < x_i$ ⟶ *in-bounds x* ⟨$\mathcal{V}$ *s*⟩ ($\mathcal{B}$ *s*))
⟨*proof*⟩

end

abbreviation *reasable-var* **where**
reasable-var dir x eq s ≡
(*coeff* (*rhs eq*) *x* > *0* ∧ $\lhd_{ub}$ (*lt dir*) (⟨𝒱 *s*⟩ *x*) (*UB dir s x*)) ∨
(*coeff* (*rhs eq*) *x* < *0* ∧ $\rhd_{lb}$ (*lt dir*) (⟨𝒱 *s*⟩ *x*) (*LB dir s x*))

locale *MinRVarsEq* =
fixes *min-rvar-incdec-eq* :: (′*i*,′*a*) *Direction* ⇒ (′*i*,′*a*::*lrv*) *state* ⇒ *eq* ⇒ ′*i list* + *var*
assumes *min-rvar-incdec-eq-None*:
min-rvar-incdec-eq dir s eq = *Inl is* ⟹
(∀ *x* ∈ *rvars-eq eq*. ¬ *reasable-var dir x eq s*) ∧
(*set is* = {*LI dir s* (*lhs eq*)} ∪ {*LI dir s x* | *x*. *x* ∈ *rvars-eq eq* ∧ *coeff* (*rhs eq*) *x* < *0*}
∪ {*UI dir s x* | *x*. *x* ∈ *rvars-eq eq* ∧ *coeff* (*rhs eq*) *x* > *0*}) ∧
((*dir* = *Positive* ∨ *dir* = *Negative*) ⟶ *LI dir s* (*lhs eq*) ∈ *indices-state s* ⟶ *set is* ⊆ *indices-state s*)
assumes *min-rvar-incdec-eq-Some-rvars*:
min-rvar-incdec-eq dir s eq = *Inr* x_j ⟹ x_j ∈ *rvars-eq eq*
assumes *min-rvar-incdec-eq-Some-incdec*:
min-rvar-incdec-eq dir s eq = *Inr* x_j ⟹ *reasable-var dir* x_j *eq s*
assumes *min-rvar-incdec-eq-Some-min*:
min-rvar-incdec-eq dir s eq = *Inr* x_j ⟹
(∀ *x* ∈ *rvars-eq eq*. *x* < x_j ⟶ ¬ *reasable-var dir x eq s*)
begin
lemma *min-rvar-incdec-eq-None*′:
assumes ∗: *dir* = *Positive* ∨ *dir* = *Negative*
and *min*: *min-rvar-incdec-eq dir s eq* = *Inl is*
and *sub*: *I* = *set is*
and *Iv*: (*I*,*v*) $\models_{ib}$ ℬℐ *s*
shows *le* (*lt dir*) ((*rhs eq*) ⦃*v*⦄) ((*rhs eq*) ⦃⟨𝒱 *s*⟩⦄)
⟨*proof*⟩
end

locale *MinRVars* = *EqForLVar* + *MinRVarsEq min-rvar-incdec-eq*
for *min-rvar-incdec-eq* :: (′*i*, ′*a* :: *lrv*) *Direction* ⇒ -
begin
abbreviation *min-rvar-incdec* :: (′*i*,′*a*) *Direction* ⇒ (′*i*,′*a*) *state* ⇒ *var* ⇒ ′*i list* + *var* **where**
min-rvar-incdec dir s x_i ≡ *min-rvar-incdec-eq dir s* (*eq-for-lvar* (𝒯 *s*) x_i)
end

locale *MinVars* = *MinLVarNotInBounds min-lvar-not-in-bounds* + *MinRVars eq-idx-for-lvar min-rvar-incdec-eq*
for *min-lvar-not-in-bounds* :: (′*i*,′*a*::*lrv*) *state* ⇒ - **and**
eq-idx-for-lvar **and**

min-rvar-incdec-eq :: (*'i*, *'a* :: *lrv*) *Direction* ⇒ -

locale *PivotUpdateMinVars* =
PivotAndUpdate eq-idx-for-lvar pivot-and-update +
MinVars min-lvar-not-in-bounds eq-idx-for-lvar min-rvar-incdec-eq **for**
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*
begin

definition *check'* **where**
check' dir x_i *s* ≡
  let l_i = *the* (*LB dir s* x_i);
    $x_j{}'$ = *min-rvar-incdec dir s* x_i
  in case $x_j{}'$ *of*
    Inl I ⇒ *set-unsat I s*
  | *Inr* x_j ⇒ *pivot-and-update* x_i x_j l_i *s*

lemma *check'-cases*:
assumes ⋀ *I*. ⟦*min-rvar-incdec dir s* x_i = *Inl I*; *check' dir* x_i *s* = *set-unsat I s*⟧
⟹ *P* (*set-unsat I s*)
assumes ⋀ x_j l_i. ⟦*min-rvar-incdec dir s* x_i = *Inr* x_j;
  l_i = *the* (*LB dir s* x_i);
  check' dir x_i *s* = *pivot-and-update* x_i x_j l_i *s*⟧ ⟹
  P (*pivot-and-update* x_i x_j l_i *s*)
shows *P* (*check' dir* x_i *s*)
⟨*proof*⟩

partial-function (*tailrec*) *check* **where**
check s =
  (*if* 𝒰 *s then s*
  else let $x_i{}'$ = *min-lvar-not-in-bounds s*
    in case $x_i{}'$ *of*
      None ⇒ *s*
    | *Some* x_i ⇒ *let dir* = *if* ⟨𝒱 *s*⟩ x_i $<_{lb}$ $\mathcal{B}_l$ *s* x_i *then Positive*
          else Negative
      in check (*check' dir* x_i *s*))
declare *check.simps*[*code*]

inductive *check-dom* **where**
step: ⟦⋀x_i. ⟦¬ 𝒰 *s*; *Some* x_i = *min-lvar-not-in-bounds s*; ⟨𝒱 *s*⟩ x_i $<_{lb}$ $\mathcal{B}_l$ *s* x_i⟧
  ⟹ *check-dom* (*check' Positive* x_i *s*);
⋀x_i. ⟦¬ 𝒰 *s*; *Some* x_i = *min-lvar-not-in-bounds s*; ¬ ⟨𝒱 *s*⟩ x_i $<_{lb}$ $\mathcal{B}_l$ *s* x_i⟧
  ⟹ *check-dom* (*check' Negative* x_i *s*)⟧
⟹ *check-dom s*

The definition of *check* can be given by:

$check\ s \equiv if\ \mathcal{U}\ s\ then\ s$
$\quad else\ let\ x_i{}' = min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s\ in$
$\quad\quad case\ x_i{}'\ of\ None \Rightarrow s$
$\quad\quad\quad |\ Some\ x_i \Rightarrow if\ \langle \mathcal{V}\ s\rangle\ x_i <_{lb} \mathcal{B}_l\ s\ x_i\ then\ check\ (check\text{-}inc\ x_i\ s)$
$\quad\quad\quad\quad else\ check\ (check\text{-}dec\ x_i\ s)$

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

The definition of *check-dec* is analogous. It is shown (mainly by induction) that this definition satisfies the *check* specification. Note that this definition uses general recursion, so its termination is non-trivial. It has been shown that it terminates for all states satisfying the check preconditions. The proof is based on the proof outline given in [1]. It is very technically involved, but conceptually uninteresting so we do not discuss it in more details.

lemma *pivotandupdate-check-precond*:
assumes
$dir = (if\ \langle \mathcal{V}\ s\rangle\ x_i <_{lb} \mathcal{B}_l\ s\ x_i\ then\ Positive\ else\ Negative)$
$min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = Some\ x_i$
$min\text{-}rvar\text{-}incdec\ dir\ s\ x_i = Inr\ x_j$
$l_i = the\ (LB\ dir\ s\ x_i)$
$\nabla\ s\ \triangle\ (\mathcal{T}\ s)\ \models_{nolhs}\ s\ \ \Diamond\ s$
shows $\triangle\ (\mathcal{T}\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s)) \wedge \models_{nolhs} (pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s) \wedge \Diamond\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s) \wedge \nabla\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s)$
$\langle proof\rangle$

abbreviation *gt-state′* **where**
$gt\text{-}state'\ dir\ s\ s'\ x_i\ x_j\ l_i \equiv$
$min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = Some\ x_i\ \wedge$
$l_i = the\ (LB\ dir\ s\ x_i)\ \wedge$
$min\text{-}rvar\text{-}incdec\ dir\ s\ x_i = Inr\ x_j\ \wedge$
$s' = pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s$

definition *gt-state* :: $('i,'a)\ state \Rightarrow ('i,'a)\ state \Rightarrow bool$ (**infixl** $\succ_x$ *100*) **where**
$s \succ_x s' \equiv$
$\exists\ x_i\ x_j\ l_i.$
$let\ dir = if\ \langle \mathcal{V}\ s\rangle\ x_i <_{lb} \mathcal{B}_l\ s\ x_i\ then\ Positive\ else\ Negative\ in$
$gt\text{-}state'\ dir\ s\ s'\ x_i\ x_j\ l_i$

abbreviation *succ* :: $('i,'a)\ state \Rightarrow ('i,'a)\ state \Rightarrow bool$ (**infixl** $\succ$ *100*) **where**
$s \succ s' \equiv \triangle\ (\mathcal{T}\ s) \wedge \Diamond\ s \wedge \models_{nolhs}\ s \wedge \nabla\ s \wedge s \succ_x s' \wedge \mathcal{B}_i\ s' = \mathcal{B}_i\ s \wedge \mathcal{U}_c\ s' = \mathcal{U}_c\ s$

abbreviation *succ-rel* :: $('i,'a)$ *state rel* **where**
 succ-rel $\equiv \{(s,\ s').\ s \succ s'\}$

abbreviation *succ-rel-trancl* :: $('i,'a)$ *state* $\Rightarrow$ $('i,'a)$ *state* $\Rightarrow$ *bool* (**infixl** $\succ^+$ *100*)
where
 $s \succ^+ s' \equiv (s,\ s') \in$ *succ-rel*$^+$

abbreviation *succ-rel-rtrancl* :: $('i,'a)$ *state* $\Rightarrow$ $('i,'a)$ *state* $\Rightarrow$ *bool* (**infixl** $\succ^*$ *100*)
where
 $s \succ^* s' \equiv (s,\ s') \in$ *succ-rel**

lemma *succ-vars*:
 assumes $s \succ s'$
 obtains x_i x_j **where**
  $x_i \in$ *lvars* $(\mathcal{T}\ s)$
  $x_j \in$ *rvars-of-lvar* $(\mathcal{T}\ s)$ x_i $x_j \in$ *rvars* $(\mathcal{T}\ s)$
  lvars $(\mathcal{T}\ s') =$ *lvars* $(\mathcal{T}\ s) - \{x_i\} \cup \{x_j\}$
  rvars $(\mathcal{T}\ s') =$ *rvars* $(\mathcal{T}\ s) - \{x_j\} \cup \{x_i\}$
⟨*proof*⟩

lemma *succ-vars-id*:
 assumes $s \succ s'$
 shows *lvars* $(\mathcal{T}\ s) \cup$ *rvars* $(\mathcal{T}\ s) =$
   lvars $(\mathcal{T}\ s') \cup$ *rvars* $(\mathcal{T}\ s')$
 ⟨*proof*⟩

lemma *succ-inv*:
 assumes $s \succ s'$
 shows $\triangle$ $(\mathcal{T}\ s')$ ∇ s' $\Diamond$ s' $\mathcal{B}_i\ s = \mathcal{B}_i\ s'$
  $(v{::}'a\ valuation) \models_t (\mathcal{T}\ s) \longleftrightarrow v \models_t (\mathcal{T}\ s')$
⟨*proof*⟩

lemma *succ-rvar-valuation-id*:
 assumes $s \succ s'$ $x \in$ *rvars* $(\mathcal{T}\ s)$ $x \in$ *rvars* $(\mathcal{T}\ s')$
 shows $\langle\mathcal{V}\ s\rangle\ x = \langle\mathcal{V}\ s'\rangle\ x$
⟨*proof*⟩

lemma *succ-min-lvar-not-in-bounds*:
 assumes $s \succ s'$
  $xr \in$ *lvars* $(\mathcal{T}\ s)$ $xr \in$ *rvars* $(\mathcal{T}\ s')$
 shows $\neg$ *in-bounds* xr $(\langle\mathcal{V}\ s\rangle)$ $(\mathcal{B}\ s)$
  $\forall\ x \in$ *lvars* $(\mathcal{T}\ s).\ x < xr \longrightarrow$ *in-bounds* x $(\langle\mathcal{V}\ s\rangle)$ $(\mathcal{B}\ s)$
⟨*proof*⟩

lemma *succ-min-rvar*:
 assumes $s \succ s'$
  $xs \in$ *lvars* $(\mathcal{T}\ s)$ $xs \in$ *rvars* $(\mathcal{T}\ s')$
  $xr \in$ *rvars* $(\mathcal{T}\ s)$ $xr \in$ *lvars* $(\mathcal{T}\ s')$

$eq = eq\text{-}for\text{-}lvar\ (\mathcal{T}\ s)\ xs$ **and**
dir: $dir = Positive \vee dir = Negative$
shows
$\neg \trianglerighteq_{lb}\ (lt\ dir)\ (\langle \mathcal{V}\ s\rangle\ xs)\ (LB\ dir\ s\ xs) \longrightarrow$
$reasable\text{-}var\ dir\ xr\ eq\ s \wedge (\forall\ x \in rvars\text{-}eq\ eq.\ x < xr \longrightarrow \neg\ reasable\text{-}var$
$dir\ x\ eq\ s)$
$\langle proof\rangle$

lemma *succ-set-on-bound*:
assumes
$s \succ s'\ x_i \in lvars\ (\mathcal{T}\ s)\ x_i \in rvars\ (\mathcal{T}\ s')$ **and**
dir: $dir = Positive \vee dir = Negative$
shows
$\neg \trianglerighteq_{lb}\ (lt\ dir)\ (\langle \mathcal{V}\ s\rangle\ x_i)\ (LB\ dir\ s\ x_i) \longrightarrow \langle \mathcal{V}\ s'\rangle\ x_i = the\ (LB\ dir\ s\ x_i)$
$\langle \mathcal{V}\ s'\rangle\ x_i = the\ (\mathcal{B}_l\ s\ x_i) \vee \langle \mathcal{V}\ s'\rangle\ x_i = the\ (\mathcal{B}_u\ s\ x_i)$
$\langle proof\rangle$

lemma *succ-rvar-valuation*:
assumes
$s \succ s'\ x \in rvars\ (\mathcal{T}\ s')$
shows
$\langle \mathcal{V}\ s'\rangle\ x = \langle \mathcal{V}\ s\rangle\ x \vee \langle \mathcal{V}\ s'\rangle\ x = the\ (\mathcal{B}_l\ s\ x) \vee \langle \mathcal{V}\ s'\rangle\ x = the\ (\mathcal{B}_u\ s\ x)$
$\langle proof\rangle$

lemma *succ-no-vars-valuation*:
assumes
$s \succ s'\ x \notin tvars\ (\mathcal{T}\ s')$
shows $look\ (\mathcal{V}\ s')\ x = look\ (\mathcal{V}\ s)\ x$
$\langle proof\rangle$

lemma *succ-valuation-satisfies*:
assumes $s \succ s'\ \langle \mathcal{V}\ s\rangle \models_t \mathcal{T}\ s$
shows $\langle \mathcal{V}\ s'\rangle \models_t \mathcal{T}\ s'$
$\langle proof\rangle$

lemma *succ-tableau-valuated*:
assumes $s \succ s'\ \nabla\ s$
shows $\nabla\ s'$
$\langle proof\rangle$

abbreviation *succ-chain* **where**
$succ\text{-}chain\ l \equiv rel\text{-}chain\ l\ succ\text{-}rel$

lemma *succ-chain-induct*:
assumes $*$: $succ\text{-}chain\ l\ i \le j\ j < length\ l$
assumes $base$: $\bigwedge\ i.\ P\ i\ i$
assumes $step$: $\bigwedge\ i.\ l\ !\ i \succ (l\ !\ (i + 1)) \Longrightarrow P\ i\ (i + 1)$
assumes $trans$: $\bigwedge\ i\ j\ k.\ [\![P\ i\ j;\ P\ j\ k;\ i < j;\ j \le k]\!] \Longrightarrow P\ i\ k$

shows $P\ i\ j$
⟨*proof*⟩

lemma *succ-chain-bounds-id*:
assumes $succ\text{-}chain\ l\ i \leq j\ j < length\ l$
shows $\mathcal{B}_i\ (l\ !\ i) = \mathcal{B}_i\ (l\ !\ j)$
⟨*proof*⟩

lemma *succ-chain-vars-id'*:
assumes $succ\text{-}chain\ l\ i \leq j\ j < length\ l$
shows $lvars\ (\mathcal{T}\ (l\ !\ i)) \cup rvars\ (\mathcal{T}\ (l\ !\ i)) =$
$lvars\ (\mathcal{T}\ (l\ !\ j)) \cup rvars\ (\mathcal{T}\ (l\ !\ j))$
⟨*proof*⟩

lemma *succ-chain-vars-id*:
assumes $succ\text{-}chain\ l\ i < length\ l\ j < length\ l$
shows $lvars\ (\mathcal{T}\ (l\ !\ i)) \cup rvars\ (\mathcal{T}\ (l\ !\ i)) =$
$lvars\ (\mathcal{T}\ (l\ !\ j)) \cup rvars\ (\mathcal{T}\ (l\ !\ j))$
⟨*proof*⟩

lemma *succ-chain-tableau-equiv'*:
assumes $succ\text{-}chain\ l\ i \leq j\ j < length\ l$
shows $(v::'a\ valuation) \models_t \mathcal{T}\ (l\ !\ i) \longleftrightarrow v \models_t \mathcal{T}\ (l\ !\ j)$
⟨*proof*⟩

lemma *succ-chain-tableau-equiv*:
assumes $succ\text{-}chain\ l\ \ i < length\ l\ j < length\ l$
shows $(v::'a\ valuation) \models_t \mathcal{T}\ (l\ !\ i) \longleftrightarrow v \models_t \mathcal{T}\ (l\ !\ j)$
⟨*proof*⟩

lemma *succ-chain-no-vars-valuation*:
assumes $succ\text{-}chain\ l\ \ i \leq j\ j < length\ l$
shows $\forall\ x.\ x \notin tvars\ (\mathcal{T}\ (l\ !\ i)) \longrightarrow look\ (\mathcal{V}\ (l\ !\ i))\ x = look\ (\mathcal{V}\ (l\ !\ j))\ x$ (**is** *?P i j*)
⟨*proof*⟩

lemma *succ-chain-rvar-valuation*:
assumes $succ\text{-}chain\ l\ i \leq j\ j < length\ l$
shows $\forall x{\in}rvars\ (\mathcal{T}\ (l\ !\ j)).$
$\langle\mathcal{V}\ (l\ !\ j)\rangle\ x = \langle\mathcal{V}\ (l\ !\ i)\rangle\ x\ \vee$
$\langle\mathcal{V}\ (l\ !\ j)\rangle\ x = the\ (\mathcal{B}_l\ (l\ !\ i)\ x)\ \vee$
$\langle\mathcal{V}\ (l\ !\ j)\rangle\ x = the\ (\mathcal{B}_u\ (l\ !\ i)\ x)$ (**is** *?P i j*)
⟨*proof*⟩

lemma *succ-chain-valuation-satisfies*:
assumes $succ\text{-}chain\ l\ \ i \leq j\ j < length\ l$
shows $\langle\mathcal{V}\ (l\ !\ i)\rangle \models_t \mathcal{T}\ (l\ !\ i) \longrightarrow \langle\mathcal{V}\ (l\ !\ j)\rangle \models_t \mathcal{T}\ (l\ !\ j)$
⟨*proof*⟩

lemma *succ-chain-tableau-valuated*:
assumes *succ-chain l* $i \leq j$ $j < $ *length l*
shows ∇ $(l\ !\ i) \longrightarrow \nabla$ $(l\ !\ j)$
⟨*proof*⟩

abbreviation *swap-lr* **where**
swap-lr l i x $\equiv$ $i + 1 <$ *length l* $\wedge$ $x \in$ *lvars* $(\mathcal{T}\ (l\ !\ i)) \wedge x \in$ *rvars* $(\mathcal{T}\ (l\ !\ (i + 1)))$

abbreviation *swap-rl* **where**
swap-rl l i x $\equiv$ $i + 1 <$ *length l* $\wedge$ $x \in$ *rvars* $(\mathcal{T}\ (l\ !\ i)) \wedge x \in$ *lvars* $(\mathcal{T}\ (l\ !\ (i + 1)))$

abbreviation *always-r* **where**
always-r l i j x $\equiv \forall\ k.\ i \leq k \wedge k \leq j \longrightarrow x \in$ *rvars* $(\mathcal{T}\ (l\ !\ k))$

lemma *succ-chain-always-r-valuation-id*:
assumes *succ-chain l* $i \leq j$ $j <$ *length l*
shows *always-r l i j x* $\longrightarrow \langle\mathcal{V}\ (l\ !\ i)\rangle\ x = \langle\mathcal{V}\ (l\ !\ j)\rangle\ x$ (**is** *?P i j*)
⟨*proof*⟩

lemma *succ-chain-swap-rl-exists*:
assumes *succ-chain l* $i < j$ $j <$ *length l*
$x \in$ *rvars* $(\mathcal{T}\ (l\ !\ i))$ $x \in$ *lvars* $(\mathcal{T}\ (l\ !\ j))$
shows $\exists\ k.\ i \leq k \wedge k < j \wedge$ *swap-rl l k x*
⟨*proof*⟩

lemma *succ-chain-swap-lr-exists*:
assumes *succ-chain l* $i < j$ $j <$ *length l*
$x \in$ *lvars* $(\mathcal{T}\ (l\ !\ i))$ $x \in$ *rvars* $(\mathcal{T}\ (l\ !\ j))$
shows $\exists\ k.\ i \leq k \wedge k < j \wedge$ *swap-lr l k x*
⟨*proof*⟩

lemma *finite-tableaus-aux*:
shows *finite* $\{t.$ *lvars* $t = L \wedge$ *rvars* $t = V - L \wedge \triangle\ t \wedge (\forall\ v{::}'a$ *valuation*. $v \models_t t = v \models_t t0)\}$ (**is** *finite* (*?Al L*))
⟨*proof*⟩

lemma *finite-tableaus*:
assumes *finite V*
shows *finite* $\{t.$ *tvars* $t = V \wedge \triangle\ t \wedge (\forall\ v{::}'a$ *valuation*. $v \models_t t = v \models_t t0)\}$ (**is** *finite ?A*)
⟨*proof*⟩

lemma *finite-accessible-tableaus*:
shows *finite* $(\mathcal{T}\ `\ \{s'.\ s \succ^* s'\})$
⟨*proof*⟩

abbreviation *check-valuation* **where**
check-valuation $(v::'a\ valuation)\ v0\ bl0\ bu0\ t0\ V \equiv$
$\exists\ t.\ tvars\ t = V \wedge \triangle\ t \wedge (\forall\ v::'a\ valuation.\ v \models_t t = v \models_t t0) \wedge v \models_t t \wedge$
$(\forall\ x \in rvars\ t.\ v\ x = v0\ x \vee v\ x = bl0\ x \vee v\ x = bu0\ x) \wedge$
$(\forall\ x.\ x \notin V \longrightarrow v\ x = v0\ x)$

lemma *finite-valuations*:
assumes *finite V*
shows *finite* $\{v::'a\ valuation.\ check\text{-}valuation\ v\ v0\ bl0\ bu0\ t0\ V\}$ (**is** *finite ?A*)
⟨*proof*⟩

lemma *finite-accessible-valuations*:
shows *finite* $(\mathcal{V}\ `\ \{s'.\ s \succ^* s'\})$
⟨*proof*⟩

lemma *accessible-bounds*:
shows $\mathcal{B}_i\ `\ \{s'.\ s \succ^* s'\} = \{\mathcal{B}_i\ s\}$
⟨*proof*⟩

lemma *accessible-unsat-core*:
shows $\mathcal{U}_c\ `\ \{s'.\ s \succ^* s'\} = \{\mathcal{U}_c\ s\}$
⟨*proof*⟩

lemma *state-eqI*:
$\mathcal{B}_{il}\ s = \mathcal{B}_{il}\ s' \Longrightarrow \mathcal{B}_{iu}\ s = \mathcal{B}_{iu}\ s' \Longrightarrow$
$\mathcal{T}\ s = \mathcal{T}\ s' \Longrightarrow \mathcal{V}\ s = \mathcal{V}\ s' \Longrightarrow$
$\mathcal{U}\ s = \mathcal{U}\ s' \Longrightarrow \mathcal{U}_c\ s = \mathcal{U}_c\ s' \Longrightarrow$
$s = s'$
⟨*proof*⟩

lemma *finite-accessible-states*:
shows *finite* $\{s'.\ s \succ^* s'\}$ (**is** *finite ?A*)
⟨*proof*⟩

lemma *acyclic-suc-rel*: *acyclic succ-rel*
⟨*proof*⟩

lemma *check-unsat-terminates*:
assumes $\mathcal{U}\ s$
shows *check-dom s*
⟨*proof*⟩

lemma *check-sat-terminates'-aux*:
assumes

dir: $dir = (if\ \langle\mathcal{V}\ s\rangle\ x_i <_{lb} \mathcal{B}_l\ s\ x_i\ then\ Positive\ else\ Negative)$ **and**
$*$: $\bigwedge s'. [\![s \succ s'; \nabla\ s'; \triangle\ (\mathcal{T}\ s'); \Diamond\ s'; \models_{nolhs}\ s'\ ]\!] \Longrightarrow check\text{-}dom\ s'$ **and**
$\nabla\ s\ \triangle\ (\mathcal{T}\ s)\ \Diamond\ s \models_{nolhs} s$
$\neg\ \mathcal{U}\ s\ min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = Some\ x_i$
$\lhd_{lb}\ (lt\ dir)\ (\langle\mathcal{V}\ s\rangle\ x_i)\ (LB\ dir\ s\ x_i)$
shows *check-dom*
$(case\ min\text{-}rvar\text{-}incdec\ dir\ s\ x_i\ of\ Inl\ I \Rightarrow set\text{-}unsat\ I\ s$
$\mid Inr\ x_j \Rightarrow pivot\text{-}and\text{-}update\ x_i\ x_j\ (the\ (LB\ dir\ s\ x_i))\ s)$
⟨*proof*⟩

lemma *check-sat-terminates′*:
assumes $\nabla\ s\ \triangle\ (\mathcal{T}\ s)\ \Diamond\ s \models_{nolhs} s\ s_0 \succ^* s$
shows *check-dom s*
⟨*proof*⟩

lemma *check-sat-terminates*:
assumes $\nabla\ s\ \triangle\ (\mathcal{T}\ s)\ \Diamond\ s \models_{nolhs} s$
shows *check-dom s*
⟨*proof*⟩

lemma *check-cases*:
assumes $\mathcal{U}\ s \Longrightarrow P\ s$
assumes $[\![\neg\ \mathcal{U}\ s; min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = None]\!] \Longrightarrow P\ s$
assumes $\bigwedge x_i\ dir\ I.$
$[\![dir = Positive \lor dir = Negative;$
$\neg\ \mathcal{U}\ s; min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = Some\ x_i;$
$\lhd_{lb}\ (lt\ dir)\ (\langle\mathcal{V}\ s\rangle\ x_i)\ (LB\ dir\ s\ x_i);$
$min\text{-}rvar\text{-}incdec\ dir\ s\ x_i = Inl\ I]\!] \Longrightarrow$
$P\ (set\text{-}unsat\ I\ s)$
assumes $\bigwedge x_i\ x_j\ l_i\ dir.$
$[\![dir = (if\ \langle\mathcal{V}\ s\rangle\ x_i <_{lb} \mathcal{B}_l\ s\ x_i\ then\ Positive\ else\ Negative);$
$\neg\ \mathcal{U}\ s; min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = Some\ x_i;$
$\lhd_{lb}\ (lt\ dir)\ (\langle\mathcal{V}\ s\rangle\ x_i)\ (LB\ dir\ s\ x_i);$
$min\text{-}rvar\text{-}incdec\ dir\ s\ x_i = Inr\ x_j;$
$l_i = the\ (LB\ dir\ s\ x_i);$
$check'\ dir\ x_i\ s = pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s]\!] \Longrightarrow$
$P\ (check\ (pivot\text{-}and\text{-}update\ x_i\ x_j\ l_i\ s))$
assumes $\triangle\ (\mathcal{T}\ s)\ \Diamond\ s \models_{nolhs} s$
shows $P\ (check\ s)$
⟨*proof*⟩

lemma *check-induct*:
fixes $s :: ('i,'a)\ state$
assumes $*$: $\nabla\ s\ \triangle\ (\mathcal{T}\ s) \models_{nolhs} s\ \Diamond\ s$
assumes $**$:
$\bigwedge s.\ \mathcal{U}\ s \Longrightarrow P\ s\ s$
$\bigwedge s.\ [\![\neg\ \mathcal{U}\ s; min\text{-}lvar\text{-}not\text{-}in\text{-}bounds\ s = None]\!] \Longrightarrow P\ s\ s$

$\bigwedge$ *s* x_i *dir I*. ⟦*dir* = *Positive* ∨ *dir* = *Negative*; ¬ $\mathcal{U}$ *s*; *min-lvar-not-in-bounds* *s* = *Some* x_i;
$\lhd_{lb}$ (*lt dir*) (⟨$\mathcal{V}$ *s*⟩ x_i) (*LB dir s* x_i); *min-rvar-incdec dir s* x_i = *Inl I*⟧
⟹ *P s* (*set-unsat I s*)
assumes *step′*: $\bigwedge$ *s* x_i x_j l_i. ⟦△ ($\mathcal{T}$ *s*); ∇ *s*; x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-eq* (*eq-for-lvar* ($\mathcal{T}$ *s*) x_i)⟧ ⟹ *P s* (*pivot-and-update* x_i x_j l_i *s*)
assumes *trans′*: $\bigwedge$ *si sj sk*. ⟦*P si sj*; *P sj sk*⟧ ⟹ *P si sk*
shows *P s* (*check s*)
⟨*proof*⟩

lemma *check-induct′*:
fixes *s* :: (′*i*,′*a*) *state*
assumes ∇ *s* △ ($\mathcal{T}$ *s*) $\models_{nolhs}$ *s* ◊ *s*
assumes $\bigwedge$ *s* x_i *dir I*. ⟦*dir* = *Positive* ∨ *dir* = *Negative*; ¬ $\mathcal{U}$ *s*; *min-lvar-not-in-bounds* *s* = *Some* x_i;
$\lhd_{lb}$ (*lt dir*) (⟨$\mathcal{V}$ *s*⟩ x_i) (*LB dir s* x_i); *min-rvar-incdec dir s* x_i = *Inl I*; *P s*⟧
⟹ *P* (*set-unsat I s*)
assumes $\bigwedge$ *s* x_i x_j l_i. ⟦△ ($\mathcal{T}$ *s*); ∇ *s*; x_i ∈ *lvars* ($\mathcal{T}$ *s*); x_j ∈ *rvars-eq* (*eq-for-lvar* ($\mathcal{T}$ *s*) x_i); *P s*⟧ ⟹ *P* (*pivot-and-update* x_i x_j l_i *s*)
assumes *P s*
shows *P* (*check s*)
⟨*proof*⟩

lemma *check-induct′′*:
fixes *s* :: (′*i*,′*a*) *state*
assumes ∗: ∇ *s* △ ($\mathcal{T}$ *s*) $\models_{nolhs}$ *s* ◊ *s*
assumes ∗∗:
$\mathcal{U}$ *s* ⟹ *P s*
$\bigwedge$ *s*. ⟦∇ *s*; △ ($\mathcal{T}$ *s*); $\models_{nolhs}$ *s*; ◊ *s*; ¬ $\mathcal{U}$ *s*; *min-lvar-not-in-bounds s* = *None*⟧ ⟹ *P s*
$\bigwedge$ *s* x_i *dir I*. ⟦*dir* = *Positive* ∨ *dir* = *Negative*; ∇ *s*; △ ($\mathcal{T}$ *s*); $\models_{nolhs}$ *s*; ◊ *s*; ¬ $\mathcal{U}$ *s*;
min-lvar-not-in-bounds s = *Some* x_i; $\lhd_{lb}$ (*lt dir*) (⟨$\mathcal{V}$ *s*⟩ x_i) (*LB dir s* x_i);
min-rvar-incdec dir s x_i = *Inl I*⟧
⟹ *P* (*set-unsat I s*)
shows *P* (*check s*)
⟨*proof*⟩

end

lemma *poly-eval-update*: (*p* ⦃ *v* (*x* := *c* :: ′*a* :: *lrv*) ⦄) = (*p* ⦃ *v* ⦄) + *coeff p x* ∗*R* (*c* − *v x*)
⟨*proof*⟩

lemma *bounds-consistent-set-unsat*[*simp*]: ◊ (*set-unsat I s*) = ◊ *s*
⟨*proof*⟩

lemma *curr-val-satisfies-no-lhs-set-unsat*[*simp*]: ($\models_{nolhs}$ (*set-unsat I s*)) = ($\models_{nolhs}$ *s*)
  ⟨*proof*⟩

context *PivotUpdateMinVars*
begin
context
  fixes *rhs-eq-val* :: (*var*, *'a*::*lrv*) *mapping* ⇒ *var* ⇒ *'a* ⇒ *eq* ⇒ *'a*
  assumes *RhsEqVal rhs-eq-val*
begin

lemma *check-minimal-unsat-state-core*:
  assumes ∗: $\neg\, \mathcal{U}\ s\ \models_{nolhs}\ s\ \Diamond\ s\ \triangle\ (\mathcal{T}\ s)\ \nabla\ s$
shows $\mathcal{U}$ (*check s*) ⟶ *minimal-unsat-state-core* (*check s*)
  (**is** *?P* (*check s*))
⟨*proof*⟩

lemma *Check-check*: *Check check*
⟨*proof*⟩
end
end

6.8 Symmetries

Simplex algorithm exhibits many symmetric cases. For example, *assert-bound* treats atoms *Leq x c* and *Geq x c* in a symmetric manner, *check-inc* and *check-dec* are symmetric, etc. These symmetric cases differ only in several aspects: order relations between numbers ($<$ vs $>$ and $\leq$ vs $\geq$), the role of lower and upper bounds ($\mathcal{B}_l$ vs $\mathcal{B}_u$) and their updating functions, comparisons with bounds (e.g., $\geq_{ub}$ vs $\leq_{lb}$ or $<_{lb}$ vs $>_{ub}$), and atom constructors (*Leq* and *Geq*). These can be attributed to two different orientations (positive and negative) of rational axis. To avoid duplicating definitions and proofs, *assert-bound* definition cases for *Leq* and *Geq* are replaced by a call to a newly introduced function parametrized by a *Direction* — a record containing minimal set of aspects listed above that differ in two definition cases such that other aspects can be derived from them (e.g., only $<$ need to be stored while $\leq$ can be derived from it). Two constants of the type *Direction* are defined: *Positive* (with $<$, $\leq$ orders, $\mathcal{B}_l$ for lower and $\mathcal{B}_u$ for upper bounds and their corresponding updating functions, and *Leq* constructor) and *Negative* (completely opposite from the previous one). Similarly, *check-inc* and *check-dec* are replaced by a new function *check-incdec* parametrized by a *Direction*. All lemmas, previously repeated for each symmetric instance, were replaced by a more abstract one, again parametrized by a *Direction* parameter.

6.9 Concrete implementation

It is easy to give a concrete implementation of the initial state constructor, which satisfies the specification of the *Init* locale. For example:

definition *init-state* :: - ⇒ ('i,'a :: zero)state **where**
init-state t = State t Mapping.empty Mapping.empty (Mapping.tabulate (vars-list t) (λ v. 0)) False None

interpretation *Init init-state* :: - ⇒ ('i,'a :: lrv)state
⟨*proof*⟩

definition *min-lvar-not-in-bounds* :: ('i,'a::{linorder,zero}) state ⇒ var option **where**
min-lvar-not-in-bounds s ≡
min-satisfying (λ x. ¬ *in-bounds x* (⟨𝒱 s⟩) (ℬ s)) (*map lhs* (𝒯 s))

interpretation *MinLVarNotInBounds min-lvar-not-in-bounds* :: ('i,'a::lrv) state ⇒ -
⟨*proof*⟩
definition *unsat-indices* :: ('i,'a :: linorder) Direction ⇒ ('i,'a) state ⇒ var list ⇒ eq ⇒ 'i list **where**
unsat-indices dir s vs eq = (let r = rhs eq; li = LI dir s; ui = UI dir s in
remdups (li (lhs eq) # map (λ x. if coeff r x < 0 then li x else ui x) vs))

definition *min-rvar-incdec-eq* :: ('i,'a) Direction ⇒ ('i,'a::lrv) state ⇒ eq ⇒ 'i list + var **where**
min-rvar-incdec-eq dir s eq = (let rvars = Abstract-Linear-Poly.vars-list (rhs eq)
in case min-satisfying (λ x. reasable-var dir x eq s) rvars of
None ⇒ Inl (unsat-indices dir s rvars eq)
| Some x_j ⇒ *Inr* x_j)

interpretation *MinRVarsEq min-rvar-incdec-eq* :: ('i,'a :: lrv) Direction ⇒ -
⟨*proof*⟩

primrec *eq-idx-for-lvar-aux* :: tableau ⇒ var ⇒ nat ⇒ nat **where**
eq-idx-for-lvar-aux [] x i = i
| *eq-idx-for-lvar-aux (eq # t) x i =*
(if lhs eq = x then i else eq-idx-for-lvar-aux t x (i+1))

definition *eq-idx-for-lvar* **where**
eq-idx-for-lvar t x ≡ eq-idx-for-lvar-aux t x 0

lemma *eq-idx-for-lvar-aux*:
  assumes $x \in lvars\ t$
  shows *let* $idx = eq\text{-}idx\text{-}for\text{-}lvar\text{-}aux\ t\ x\ i$ *in*
    $i \leq idx \wedge idx < i + length\ t \wedge lhs\ (t\ !\ (idx - i)) = x$
  ⟨*proof*⟩

global-interpretation *EqForLVarDefault*: *EqForLVar eq-idx-for-lvar*
  defines *eq-for-lvar-code* = *EqForLVarDefault.eq-for-lvar*
⟨*proof*⟩

definition *pivot-eq* :: *eq* ⇒ *var* ⇒ *eq* **where**
  $pivot\text{-}eq\ e\ y \equiv let\ cy = coeff\ (rhs\ e)\ y\ in$
    $(y, (-1/cy) *R ((rhs\ e) - cy *R (Var\ y)) + (1/cy) *R (Var\ (lhs\ e)))$

lemma *pivot-eq-satisfies-eq*:
  assumes $y \in rvars\text{-}eq\ e$
  shows $v \models_e e = v \models_e pivot\text{-}eq\ e\ y$
  ⟨*proof*⟩

lemma *pivot-eq-rvars*:
  assumes $x \in vars\ (rhs\ (pivot\text{-}eq\ e\ v))\ x \neq lhs\ e\ coeff\ (rhs\ e)\ v \neq 0\ v \neq lhs\ e$
  shows $x \in vars\ (rhs\ e)$
⟨*proof*⟩

interpretation *PivotEq pivot-eq*
⟨*proof*⟩

definition *subst-var*:: *var* ⇒ *linear-poly* ⇒ *linear-poly* ⇒ *linear-poly* **where**
  $subst\text{-}var\ v\ lp'\ lp \equiv lp + (coeff\ lp\ v) *R\ lp' - (coeff\ lp\ v) *R\ (Var\ v)$

definition *subst-var-eq-code* = *SubstVar.subst-var-eq subst-var*

global-interpretation *SubstVar subst-var* **rewrites**
  SubstVar.subst-var-eq subst-var = *subst-var-eq-code*
⟨*proof*⟩

definition *rhs-eq-val* **where**
rhs-eq-val $v\ x_i\ c\ e \equiv$ *let* $x_j = $ *lhs* e; $a_{ij} = $ *coeff* (*rhs* e) x_i *in*
$\langle v\rangle\ x_j + a_{ij} *R\ (c - \langle v\rangle\ x_i)$

definition *update-code = RhsEqVal.update rhs-eq-val*
definition *assert-bound′-code = Update.assert-bound′ update-code*
definition *assert-bound-code = Update.assert-bound update-code*

global-interpretation *RhsEqValDefault′*: *RhsEqVal rhs-eq-val*
rewrites
RhsEqVal.update rhs-eq-val = update-code **and**
Update.assert-bound update-code = assert-bound-code **and**
Update.assert-bound′ update-code = assert-bound′-code
⟨*proof*⟩

sublocale *PivotUpdateMinVars < Check check*
⟨*proof*⟩

definition *pivot-code = Pivot′.pivot eq-idx-for-lvar pivot-eq subst-var*
definition *pivot-tableau-code = Pivot′.pivot-tableau eq-idx-for-lvar pivot-eq subst-var*

global-interpretation *Pivot′Default*: *Pivot′ eq-idx-for-lvar pivot-eq subst-var*
rewrites
Pivot′.pivot eq-idx-for-lvar pivot-eq subst-var = pivot-code **and**
Pivot′.pivot-tableau eq-idx-for-lvar pivot-eq subst-var = pivot-tableau-code **and**
SubstVar.subst-var-eq subst-var = subst-var-eq-code
⟨*proof*⟩

definition *pivot-and-update-code = PivotUpdate.pivot-and-update pivot-code update-code*

global-interpretation *PivotUpdateDefault*: *PivotUpdate eq-idx-for-lvar pivot-code update-code*
rewrites
PivotUpdate.pivot-and-update pivot-code update-code = pivot-and-update-code
⟨*proof*⟩

sublocale *Update < AssertBoundNoLhs assert-bound*
⟨*proof*⟩

definition *check-code = PivotUpdateMinVars.check eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq pivot-and-update-code*
definition *check′-code = PivotUpdateMinVars.check′ eq-idx-for-lvar min-rvar-incdec-eq pivot-and-update-code*

global-interpretation *PivotUpdateMinVarsDefault*: *PivotUpdateMinVars eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq pivot-and-update-code*
rewrites
PivotUpdateMinVars.check eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq

pivot-and-update-code = *check-code* **and**
PivotUpdateMinVars.check′ eq-idx-for-lvar min-rvar-incdec-eq pivot-and-update-code = *check′-code*
⟨*proof*⟩

definition *assert-code* = *Assert′.assert assert-bound-code check-code*

global-interpretation *Assert′Default*: *Assert′ assert-bound-code check-code*
rewrites
Assert′.assert assert-bound-code check-code = *assert-code*
⟨*proof*⟩

definition *assert-bound-loop-code* = *AssertAllState″.assert-bound-loop assert-bound-code*
definition *assert-all-state-code* = *AssertAllState″.assert-all-state init-state assert-bound-code check-code*
definition *assert-all-code* = *AssertAllState.assert-all assert-all-state-code*

global-interpretation *AssertAllStateDefault*: *AssertAllState″ init-state assert-bound-code check-code*
rewrites
AssertAllState″.assert-bound-loop assert-bound-code = *assert-bound-loop-code*
and
AssertAllState″.assert-all-state init-state assert-bound-code check-code = *assert-all-state-code* **and**
AssertAllState.assert-all assert-all-state-code = *assert-all-code*
⟨*proof*⟩

primrec
monom-to-atom:: *QDelta ns-constraint* ⇒ *QDelta atom* **where**
monom-to-atom (*LEQ-ns l r*) = (*if* (*monom-coeff l* < *0*) *then*
(*Geq* (*monom-var l*) (*r* /*R monom-coeff l*))
else
(*Leq* (*monom-var l*) (*r* /*R monom-coeff l*)))
| *monom-to-atom* (*GEQ-ns l r*) = (*if* (*monom-coeff l* < *0*) *then*
(*Leq* (*monom-var l*) (*r* /*R monom-coeff l*))
else
(*Geq* (*monom-var l*) (*r* /*R monom-coeff l*)))

primrec
qdelta-constraint-to-atom:: *QDelta ns-constraint* ⇒ *var* ⇒ *QDelta atom* **where**
qdelta-constraint-to-atom (*LEQ-ns l r*) *v* = (*if* (*is-monom l*) *then* (*monom-to-atom* (*LEQ-ns l r*)) *else* (*Leq v r*))
| *qdelta-constraint-to-atom* (*GEQ-ns l r*) *v* = (*if* (*is-monom l*) *then* (*monom-to-atom* (*GEQ-ns l r*)) *else* (*Geq v r*))

primrec
  qdelta-constraint-to-atom':: *QDelta ns-constraint* ⇒ *var* ⇒ *QDelta atom* **where**
  qdelta-constraint-to-atom' (*LEQ-ns l r*) *v* = (*Leq v r*)
| *qdelta-constraint-to-atom'* (*GEQ-ns l r*) *v* = (*Geq v r*)

fun *linear-poly-to-eq*:: *linear-poly* ⇒ *var* ⇒ *eq* **where**
  linear-poly-to-eq p v = (*v*, *p*)

datatype *'i istate* = *IState*
  (*FirstFreshVariable*: *var*)
  (*Tableau*: *tableau*)
  (*Atoms*: (*'i*,*QDelta*) *i-atom list*)
  (*Poly-Mapping*: *linear-poly* ⇀ *var*)
  (*UnsatIndices*: *'i list*)

primrec *zero-satisfies* :: *'a* :: *lrv ns-constraint* ⇒ *bool* **where**
  zero-satisfies (*LEQ-ns l r*) ⟷ *0* ≤ *r*
| *zero-satisfies* (*GEQ-ns l r*) ⟷ *0* ≥ *r*

lemma *zero-satisfies*: *poly c* = *0* ⟹ *zero-satisfies c* ⟹ $v \models_{ns} c$
  ⟨*proof*⟩

lemma *not-zero-satisfies*: *poly c* = *0* ⟹ ¬ *zero-satisfies c* ⟹ ¬ $v \models_{ns} c$
  ⟨*proof*⟩

fun
  preprocess' :: (*'i*,*QDelta*) *i-ns-constraint list* ⇒ *var* ⇒ *'i istate* **where**
  preprocess' [] *v* = *IState v* [] [] (λ *p*. *None*) []
| *preprocess'* ((*i*,*h*) # *t*) *v* = (*let s'* = *preprocess' t v*; *p* = *poly h*; *is-monom-h* = *is-monom p*;
    v' = *FirstFreshVariable s'*;
    t' = *Tableau s'*;
    a' = *Atoms s'*;
    m' = *Poly-Mapping s'*;
    u' = *UnsatIndices s' in*
    if is-monom-h then IState v' t'
      ((*i*,*qdelta-constraint-to-atom h v'*) # *a'*) *m' u'*
    else if p = *0 then*
      if zero-satisfies h then s' else
        IState v' t' a' m' (*i* # *u'*)
    else (*case m' p of Some v* ⇒
        IState v' t' ((*i*,*qdelta-constraint-to-atom h v*) # *a'*) *m' u'*
      | *None* ⇒ *IState* (*v'* + *1*) (*linear-poly-to-eq p v'* # *t'*)
      ((*i*,*qdelta-constraint-to-atom h v'*) # *a'*) (*m'* (*p* ↦ *v'*)) *u'*)
)

lemma *preprocess'-simps*: *preprocess'* ((*i*,*h*) # *t*) *v* = (*let s'* = *preprocess' t v*; *p*

= *poly h*; *is-monom-h* = *is-monom p*;
v′ = *FirstFreshVariable s′*;
t′ = *Tableau s′*;
a′ = *Atoms s′*;
m′ = *Poly-Mapping s′*;
u′ = *UnsatIndices s′ in*
if is-monom-h then IState v′ t′
((*i*,*monom-to-atom h*) # *a′*) *m′ u′*
else if p = *0 then*
if zero-satisfies h then s′ else
IState v′ t′ a′ m′ (*i* # *u′*)
else (*case m′ p of Some v* ⇒
IState v′ t′ ((*i*,*qdelta-constraint-to-atom′ h v*) # *a′*) *m′ u′*
| *None* ⇒ *IState* (*v′* + *1*) (*linear-poly-to-eq p v′* # *t′*)
((*i*,*qdelta-constraint-to-atom′ h v′*) # *a′*) (*m′* (*p* ↦ *v′*)) *u′*)
) ⟨*proof*⟩

lemmas *preprocess′-code* = *preprocess′.simps*(*1*) *preprocess′-simps*
declare *preprocess′-code*[*code*]

Normalization of constraints helps to identify same polynomials, e.g., the constraints $x + y \leq 5$ and $-2x - 2y \leq -12$ will be normalized to $x + y \leq 5$ and $x + y \geq 6$, so that only one slack-variable will be introduced for the polynomial $x + y$, and not another one for $-2x - 2y$. Normalization will take care that the max-var of the polynomial in the constraint will have coefficient 1 (if the polynomial is non-zero)

fun *normalize-ns-constraint* :: *′a* :: *lrv ns-constraint* ⇒ *′a ns-constraint* **where**
normalize-ns-constraint (*LEQ-ns l r*) = (*let v* = *max-var l*; *c* = *coeff l v in*
if c = *0 then LEQ-ns l r else*
let ic = *inverse c in if c* < *0 then GEQ-ns* (*ic* ∗*R l*) (*scaleRat ic r*) *else LEQ-ns* (*ic* ∗*R l*) (*scaleRat ic r*))
| *normalize-ns-constraint* (*GEQ-ns l r*) = (*let v* = *max-var l*; *c* = *coeff l v in*
if c = *0 then GEQ-ns l r else*
let ic = *inverse c in if c* < *0 then LEQ-ns* (*ic* ∗*R l*) (*scaleRat ic r*) *else GEQ-ns* (*ic* ∗*R l*) (*scaleRat ic r*))

lemma *normalize-ns-constraint*[*simp*]: *v* $\models_{ns}$ (*normalize-ns-constraint c*) ⟷ *v* $\models_{ns}$ (*c* :: *′a* :: *lrv ns-constraint*)
⟨*proof*⟩

declare *normalize-ns-constraint.simps*[*simp del*]

lemma *i-satisfies-normalize-ns-constraint*[*simp*]: *Iv* $\models_{inss}$ (*map-prod id normalize-ns-constraint ‘ cs*)
⟷ *Iv* $\models_{inss}$ *cs*
⟨*proof*⟩

abbreviation *max-var*:: *QDelta ns-constraint* ⇒ *var* **where**

max-var C ≡ *Abstract-Linear-Poly.max-var* (*poly C*)

fun
start-fresh-variable :: ($'i$,*QDelta*) *i-ns-constraint list* ⇒ *var* **where**
start-fresh-variable [] = *0*
| *start-fresh-variable* ((*i*,*h*)#*t*) = *max* (*max-var h* + *1*) (*start-fresh-variable t*)

definition
preprocess-part-1 :: ($'i$,*QDelta*) *i-ns-constraint list* ⇒ *tableau* × (($'i$,*QDelta*) *i-atom list*) × $'i$ *list* **where**
preprocess-part-1 l ≡ *let start* = *start-fresh-variable l*; *is* = *preprocess′ l start in* (*Tableau is*, *Atoms is*, *UnsatIndices is*)

lemma *lhs-linear-poly-to-eq* [*simp*]:
lhs (*linear-poly-to-eq h v*) = *v*
⟨*proof*⟩

lemma *rvars-eq-linear-poly-to-eq* [*simp*]:
rvars-eq (*linear-poly-to-eq h v*) = *vars h*
⟨*proof*⟩

lemma *fresh-var-monoinc*:
FirstFreshVariable (*preprocess′ cs start*) ≥ *start*
⟨*proof*⟩

abbreviation *vars-constraints* **where**
vars-constraints cs ≡ ⋃ (*set* (*map vars* (*map poly cs*)))

lemma *start-fresh-variable-fresh*:
∀ *var* ∈ *vars-constraints* (*flat-list cs*). *var* < *start-fresh-variable cs*
⟨*proof*⟩

lemma *vars-tableau-vars-constraints*:
rvars (*Tableau* (*preprocess′ cs start*)) ⊆ *vars-constraints* (*flat-list cs*)
⟨*proof*⟩

lemma *lvars-tableau-ge-start*:
∀ *var* ∈ *lvars* (*Tableau* (*preprocess′ cs start*)). *var* ≥ *start*
⟨*proof*⟩

lemma *rhs-no-zero-tableau-start*:
0 ∉ *rhs* ‘ *set* (*Tableau* (*preprocess′ cs start*))
⟨*proof*⟩

lemma *first-fresh-variable-not-in-lvars*:
∀ *var* ∈ *lvars* (*Tableau* (*preprocess′ cs start*)). *FirstFreshVariable* (*preprocess′ cs start*) > *var*
⟨*proof*⟩

lemma *sat-atom-sat-eq-sat-constraint-non-monom*:
assumes $v \models_a$ *qdelta-constraint-to-atom h var* $v \models_e$ *linear-poly-to-eq* (*poly h*) *var*
$\neg$ *is-monom* (*poly h*)
shows $v \models_{ns} h$
⟨*proof*⟩

lemma *qdelta-constraint-to-atom-monom*:
assumes *is-monom* (*poly h*)
shows $v \models_a$ *qdelta-constraint-to-atom h var* $\longleftrightarrow v \models_{ns} h$
⟨*proof*⟩

lemma *preprocess′-Tableau-Poly-Mapping-None*: (*Poly-Mapping* (*preprocess′ cs start*))
p = *None*
$\Longrightarrow$ *linear-poly-to-eq p v* $\notin$ *set* (*Tableau* (*preprocess′ cs start*))
⟨*proof*⟩

lemma *preprocess′-Tableau-Poly-Mapping-Some*: (*Poly-Mapping* (*preprocess′ cs start*))
p = *Some v*
$\Longrightarrow$ *linear-poly-to-eq p v* $\in$ *set* (*Tableau* (*preprocess′ cs start*))
⟨*proof*⟩

lemma *preprocess′-Tableau-Poly-Mapping-Some′*: (*Poly-Mapping* (*preprocess′ cs start*)) *p* = *Some v*
$\Longrightarrow \exists$ *h*. *poly h* = *p* $\wedge \neg$ *is-monom* (*poly h*) $\wedge$ *qdelta-constraint-to-atom h v* $\in$
flat (*set* (*Atoms* (*preprocess′ cs start*)))
⟨*proof*⟩

lemma *not-one-le-zero-qdelta*: $\neg$ (*1* $\leq$ (*0* :: *QDelta*)) ⟨*proof*⟩

lemma *one-zero-contra*[*dest*,*consumes 2*]: *1* $\leq$ *x* $\Longrightarrow$ (*x* :: *QDelta*) $\leq$ *0* $\Longrightarrow$ *False*

⟨*proof*⟩

lemma *i-preprocess′-sat*:
assumes (*I*,*v*) $\models_{ias}$ *set* (*Atoms* (*preprocess′ s start*)) $v \models_t$ *Tableau* (*preprocess′ s start*)
I $\cap$ *set* (*UnsatIndices* (*preprocess′ s start*)) = {}
shows (*I*,*v*) $\models_{inss}$ *set s*
⟨*proof*⟩

lemma *preprocess′-sat*:
assumes $v \models_{as}$ *flat* (*set* (*Atoms* (*preprocess′ s start*))) $v \models_t$ *Tableau* (*preprocess′ s start*) *set* (*UnsatIndices* (*preprocess′ s start*)) = {}
shows $v \models_{nss}$ *flat* (*set s*)
⟨*proof*⟩

lemma *sat-constraint-valuation*:
assumes $\forall$ *var* $\in$ *vars* (*poly c*). *v1 var* = *v2 var*

shows $v1 \models_{ns} c \longleftrightarrow v2 \models_{ns} c$
⟨*proof*⟩

lemma *atom-var-first*:
assumes $a \in$ *flat* (*set* (*Atoms* (*preprocess′ cs start*))) $\forall$ *var* $\in$ *vars-constraints* (*flat-list cs*). *var* $<$ *start*
shows *atom-var a* $<$ *FirstFreshVariable* (*preprocess′ cs start*)
⟨*proof*⟩

lemma *satisfies-tableau-satisfies-tableau*:
assumes $v1 \models_t t$ $\forall$ *var* $\in$ *tvars t*. *v1 var* $=$ *v2 var*
shows $v2 \models_t t$
⟨*proof*⟩

lemma *preprocess′-unsat-indices*:
assumes $i \in$ *set* (*UnsatIndices* (*preprocess′ s start*))
shows $\neg$ $(\{i\},v) \models_{inss}$ *set s*
⟨*proof*⟩

lemma *preprocess′-unsat*:
assumes $(I,v) \models_{inss}$ *set s vars-constraints* (*flat-list s*) $\subseteq V$ $\forall$ *var* $\in V$. *var* $<$ *start*
shows $\exists v'$. ($\forall$ *var* $\in V$. *v var* $=$ *v′ var*)
$\wedge$ $v' \models_{as}$ *restrict-to I* (*set* (*Atoms* (*preprocess′ s start*)))
$\wedge$ $v' \models_t$ *Tableau* (*preprocess′ s start*)
⟨*proof*⟩

lemma *lvars-distinct*:
distinct (*map lhs* (*Tableau* (*preprocess′ cs start*)))
⟨*proof*⟩

lemma *normalized-tableau-preprocess′*: $\triangle$ (*Tableau* (*preprocess′ cs* (*start-fresh-variable cs*)))
⟨*proof*⟩

Improved preprocessing: Deletion. An equation x = p can be deleted from the tableau, if x does not occur in the atoms.

lemma *delete-lhs-var*: **assumes** *norm*: $\triangle$ *t* **and** *t*: *t* $=$ *t1* @ (*x*,*p*) # *t2*
and *t′*: *t′* $=$ *t1* @ *t2*
and *tv*: *tv* $=$ (λ *v*. *upd x* (*p* ⦃ ⟨*v*⟩ ⦄) *v*)
and *x-as*: $x \notin$ *atom-var* ‘ *snd* ‘ *set as*
shows $\triangle$ *t′* — new tableau is normalized
$\langle w\rangle \models_t t' \Longrightarrow \langle tv\ w\rangle \models_t t$ — solution of new tableau is translated to solution of old tableau
$(I, \langle w\rangle) \models_{ias}$ *set as* $\Longrightarrow$ $(I, \langle tv\ w\rangle) \models_{ias}$ *set as* — solution translation also works for bounds
$v \models_t t \Longrightarrow v \models_t t'$ — solution of old tableau is also solution for new tableau
⟨*proof*⟩

definition *pivot-tableau-eq* :: *tableau* ⇒ *eq* ⇒ *tableau* ⇒ *var* ⇒ *tableau* × *eq* × *tableau* **where**
pivot-tableau-eq t1 eq t2 x ≡ *let eq′* = *pivot-eq eq x*; *m* = *map* (λ *e*. *subst-var-eq x* (*rhs eq′*) *e*) *in*
(*m t1*, *eq′*, *m t2*)

lemma *pivot-tableau-eq*: **assumes** *t*: *t* = *t1* @ *eq* # *t2* *t′* = *t1′* @ *eq′* # *t2′*
and *x*: *x* ∈ *rvars-eq eq* **and** *norm*: △ *t* **and** *pte*: *pivot-tableau-eq t1 eq t2 x* = (*t1′*,*eq′*,*t2′*)
shows △ *t′* *lhs eq′* = *x* (*v* :: ′*a* :: *lrv valuation*) $\models_t$ *t′* ⟷ *v* $\models_t$ *t*
⟨*proof*⟩

function *preprocess-opt* :: *var set* ⇒ *tableau* ⇒ *tableau* ⇒ *tableau* × ((*var*,′*a* :: *lrv*)*mapping* ⇒ (*var*,′*a*)*mapping*) **where**
preprocess-opt X t1 [] = (*t1*,*id*)
| *preprocess-opt X t1* ((*x*,*p*) # *t2*) = (*if x* ∉ *X then*
case preprocess-opt X t1 t2 of (*t*,*tv*) ⇒ (*t*, (λ *v*. *upd x* (*p* ⦃ ⟨*v*⟩ ⦄) *v*) *o tv*)
else case find (λ *x*. *x* ∉ *X*) (*Abstract-Linear-Poly.vars-list p*) *of*
None ⇒ *preprocess-opt X* ((*x*,*p*) # *t1*) *t2*
| *Some y* ⇒ *case pivot-tableau-eq t1* (*x*,*p*) *t2 y of*
(*tt1*,(*z*,*q*),*tt2*) ⇒ *case preprocess-opt X tt1 tt2 of* (*t*,*tv*) ⇒ (*t*, (λ *v*. *upd z* (*q* ⦃ ⟨*v*⟩ ⦄) *v*) *o tv*))
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *preprocess-opt*: **assumes** *X* = *atom-var* ‘ *snd* ‘ *set as*
preprocess-opt X t1 t2 = (*t′*,*tv*) △ *t t* = *rev t1* @ *t2*
shows △ *t′*
(⟨*w*⟩ :: ′*a* :: *lrv valuation*) $\models_t$ *t′* ⟹ ⟨*tv w*⟩ $\models_t$ *t*
(*I*, ⟨*w*⟩) $\models_{ias}$ *set as* ⟹ (*I*, ⟨*tv w*⟩) $\models_{ias}$ *set as*
v $\models_t$ *t* ⟹ (*v* :: ′*a valuation*) $\models_t$ *t′*
⟨*proof*⟩

definition *preprocess-part-2 as t* = *preprocess-opt* (*atom-var* ‘ *snd* ‘ *set as*) [] *t*

lemma *preprocess-part-2*: **assumes** *preprocess-part-2 as t* = (*t′*,*tv*) △ *t*
shows △ *t′*
(⟨*w*⟩ :: ′*a* :: *lrv valuation*) $\models_t$ *t′* ⟹ ⟨*tv w*⟩ $\models_t$ *t*
(*I*, ⟨*w*⟩) $\models_{ias}$ *set as* ⟹ (*I*, ⟨*tv w*⟩) $\models_{ias}$ *set as*
v $\models_t$ *t* ⟹ (*v* :: ′*a valuation*) $\models_t$ *t′*
⟨*proof*⟩

definition *preprocess* :: (′*i*,*QDelta*) *i-ns-constraint list* ⇒ - × - × (- ⇒ (*var*,*QDelta*)*mapping*) × ′*i list* **where**
preprocess l = (*case preprocess-part-1* (*map* (*map-prod id normalize-ns-constraint*) *l*) *of*
(*t*,*as*,*ui*) ⇒ *case preprocess-part-2 as t of* (*t*,*tv*) ⇒ (*t*,*as*,*tv*,*ui*))

lemma *preprocess*:
assumes *id*: *preprocess cs* = (*t*, *as*, *trans-v*, *ui*)
shows $\triangle$ *t*
fst ‘ set as ∪ *set ui* ⊆ *fst ‘ set cs*
distinct-indices-ns (*set cs*) ⟹ *distinct-indices-atoms* (*set as*)
I ∩ *set ui* = {} ⟹ (*I*, ⟨*v*⟩) $\models_{ias}$ *set as* ⟹
⟨*v*⟩ $\models_t$ *t* ⟹ (*I*, ⟨*trans-v v*⟩) $\models_{inss}$ *set cs*
i ∈ *set ui* ⟹ ∄ *v*. ({*i*}, *v*) $\models_{inss}$ *set cs*
∃ *v*. (*I*,*v*) $\models_{inss}$ *set cs* ⟹ ∃ *v*′. (*I*,*v*′) $\models_{ias}$ *set as* ∧ *v*′ $\models_t$ *t*
⟨*proof*⟩

interpretation *PreprocessDefault*: *Preprocess preprocess*
⟨*proof*⟩

global-interpretation *Solve-exec-ns′Default*: *Solve-exec-ns′ preprocess assert-all-code*
defines *solve-exec-ns-code* = *Solve-exec-ns′Default.solve-exec-ns*
⟨*proof*⟩

primrec
constraint-to-qdelta-constraint:: *constraint* ⇒ *QDelta ns-constraint list* **where**
constraint-to-qdelta-constraint (*LT l r*) = [*LEQ-ns l* (*QDelta.QDelta r* (−*1*))]
| *constraint-to-qdelta-constraint* (*GT l r*) = [*GEQ-ns l* (*QDelta.QDelta r 1*)]
| *constraint-to-qdelta-constraint* (*LEQ l r*) = [*LEQ-ns l* (*QDelta.QDelta r 0*)]
| *constraint-to-qdelta-constraint* (*GEQ l r*) = [*GEQ-ns l* (*QDelta.QDelta r 0*)]
| *constraint-to-qdelta-constraint* (*EQ l r*) = [*LEQ-ns l* (*QDelta.QDelta r 0*), *GEQ-ns l* (*QDelta.QDelta r 0*)]

primrec
i-constraint-to-qdelta-constraint:: ′*i i-constraint* ⇒ (′*i*,*QDelta*) *i-ns-constraint list* **where**
i-constraint-to-qdelta-constraint (*i*,*c*) = *map* (*Pair i*) (*constraint-to-qdelta-constraint c*)

definition
to-ns :: ′*i i-constraint list* ⇒ (′*i*,*QDelta*) *i-ns-constraint list* **where**
to-ns l ≡ *concat* (*map i-constraint-to-qdelta-constraint l*)

primrec
δ0-val :: *QDelta ns-constraint* ⇒ *QDelta valuation* ⇒ *rat* **where**
δ0-val (*LEQ-ns lll rrr*) *vl* = *δ0 lll*⦃*vl*⦄ *rrr*
| *δ0-val* (*GEQ-ns lll rrr*) *vl* = *δ0 rrr lll*⦃*vl*⦄

primrec
δ0-val-min :: *QDelta ns-constraint list* ⇒ *QDelta valuation* ⇒ *rat* **where**
δ0-val-min [] *vl* = *1*

| *δ0-val-min* (*h#t*) *vl* = *min* (*δ0-val-min t vl*) (*δ0-val h vl*)

abbreviation *vars-list-constraints* **where**
vars-list-constraints cs ≡ *remdups* (*concat* (*map Abstract-Linear-Poly.vars-list* (*map poly cs*)))

definition
from-ns ::(*var*, *QDelta*) *mapping* ⇒ *QDelta ns-constraint list* ⇒ (*var*, *rat*) *mapping* **where**
from-ns vl cs ≡ *let δ* = *δ0-val-min cs* ⟨*vl*⟩ *in*
Mapping.tabulate (*vars-list-constraints cs*) (*λ var. val* (⟨*vl*⟩ *var*) *δ*)

global-interpretation *SolveExec′Default*: *SolveExec′ to-ns from-ns solve-exec-ns-code*
defines *solve-exec-code* = *SolveExec′Default.solve-exec*
and *solve-code* = *SolveExec′Default.solve*
⟨*proof*⟩

hide-const (**open**) *le lt LE GE LB UB LI UI LBI UBI UBI-upd le-rat*
inv zero Var add flat flat-list restrict-to look upd

Simplex version with indexed constraints as input

definition *simplex-index* :: *′i i-constraint list* ⇒ *′i list* + (*var*, *rat*) *mapping* **where**
simplex-index = *solve-exec-code*

lemma *simplex-index*:
simplex-index cs = *Unsat I* ⟹ *set I* ⊆ *fst ‘ set cs* ∧ ¬ (∃ *v*. (*set I*, *v*) $\models_{ics}$ *set cs*) ∧
(*distinct-indices cs* ⟶ (∀ *J* ⊂ *set I*. (∃ *v*. (*J*, *v*) $\models_{ics}$ *set cs*))) — minimal unsat core
simplex-index cs = *Sat v* ⟹ ⟨*v*⟩ $\models_{cs}$ (*snd ‘ set cs*) — satisfying assingment
⟨*proof*⟩

Simplex version where indices will be created

definition *simplex* **where** *simplex cs* = *simplex-index* (*zip* [*0..<length cs*] *cs*)

lemma *simplex*:
simplex cs = *Unsat I* ⟹ ¬ (∃ *v*. *v* $\models_{cs}$ *set cs*) — unsat of original constraints
simplex cs = *Unsat I* ⟹ *set I* ⊆ {*0..<length cs*} ∧ ¬ (∃ *v*. *v* $\models_{cs}$ {*cs* ! *i* | *i*. *i* ∈ *set I*})
∧ (∀ *J*⊂*set I*. ∃ *v*. *v* $\models_{cs}$ {*cs* ! *i* |*i*. *i* ∈ *J*}) — minimal unsat core
simplex cs = *Sat v* ⟹ ⟨*v*⟩ $\models_{cs}$ *set cs* — satisfying assignment
⟨*proof*⟩

check executability

lemma *case simplex* [*LT* (*lp-monom 2 1* − *lp-monom 3 2* + *lp-monom 3 0*) *0*, *GT* (*lp-monom 1 1*) *5*]
of Sat - ⇒ *True* | *Unsat -* ⇒ *False*

⟨*proof*⟩

check unsat core

lemma
case simplex-index [
(*1* :: *int*, *LT* (*lp-monom 1 1*) *4*),
(*2*, *GT* (*lp-monom 2 1* − *lp-monom 1 2*) *0*),
(*3*, *EQ* (*lp-monom 1 1* − *lp-monom 2 2*) *0*),
(*4*, *GT* (*lp-monom 2 2*) *5*),
(*5*, *GT* (*lp-monom 3 0*) *7*)]
of Sat - ⇒ False | *Unsat I ⇒ set I* = {*1*,*3*,*4*} — Constraints 1,3,4 are unsat core
⟨*proof*⟩

end

7 The Incremental Simplex Algorithm

In this theory we specify operations which permit to incrementally add constraints. To this end, first an indexed list of potential constraints is used to construct the initial state, and then one can activate indices, extract solutions or unsat cores, do backtracking, etc.

theory *Simplex-Incremental*
imports *Simplex*
begin

7.1 Lowest Layer: Fixed Tableau and Incremental Atoms

Interface

locale *Incremental-Atom-Ops* = **fixes**
init-s :: *tableau* ⇒ *′s* **and**
assert-s :: (*′i*,*′a* :: *lrv*) *i-atom* ⇒ *′s* ⇒ *′i list* + *′s* **and**
check-s :: *′s* ⇒ *′s* × (*′i list option*) **and**
solution-s :: *′s* ⇒ (*var*, *′a*) *mapping* **and**
checkpoint-s :: *′s* ⇒ *′c* **and**
backtrack-s :: *′c* ⇒ *′s* ⇒ *′s* **and**
precond-s :: *tableau* ⇒ *bool* **and**
weak-invariant-s :: *tableau* ⇒ (*′i*,*′a*) *i-atom set* ⇒ *′s* ⇒ *bool* **and**
invariant-s :: *tableau* ⇒ (*′i*,*′a*) *i-atom set* ⇒ *′s* ⇒ *bool* **and**
checked-s :: *tableau* ⇒ (*′i*,*′a*) *i-atom set* ⇒ *′s* ⇒ *bool*
assumes
assert-s-ok: *invariant-s t as s* ⟹ *assert-s a s* = *Inr s′* ⟹
invariant-s t (*insert a as*) *s′* **and**
assert-s-unsat: *invariant-s t as s* ⟹ *assert-s a s* = *Unsat I* ⟹
minimal-unsat-core-tabl-atoms (*set I*) *t* (*insert a as*) **and**
check-s-ok: *invariant-s t as s* ⟹ *check-s s* = (*s′*, *None*) ⟹
checked-s t as s′ **and**

check-s-unsat: *invariant-s t as s* $\Longrightarrow$ *check-s s* = (*s*′,*Some I*) $\Longrightarrow$
weak-invariant-s t as s′ ∧ *minimal-unsat-core-tabl-atoms* (*set I*) *t as* **and**
init-s: *precond-s t* $\Longrightarrow$ *checked-s t* {} (*init-s t*) **and**
solution-s: *checked-s t as s* $\Longrightarrow$ *solution-s s* = *v* $\Longrightarrow$ ⟨*v*⟩ $\models_t$ *t* ∧ ⟨*v*⟩ $\models_{as}$ *Simplex.flat as* **and**
backtrack-s: *checked-s t as s* $\Longrightarrow$ *checkpoint-s s* = *c*
$\Longrightarrow$ *weak-invariant-s t bs s*′ $\Longrightarrow$ *backtrack-s c s*′ = *s*′′ $\Longrightarrow$ *as* ⊆ *bs* $\Longrightarrow$ *invariant-s t as s*′′ **and**
weak-invariant-s: *invariant-s t as s* $\Longrightarrow$ *weak-invariant-s t as s* **and**
checked-invariant-s: *checked-s t as s* $\Longrightarrow$ *invariant-s t as s*
begin

fun *assert-all-s* :: (′*i*,′*a*) *i-atom list* ⇒ ′*s* ⇒ ′*i list* + ′*s* **where**
assert-all-s [] *s* = *Inr s*
| *assert-all-s* (*a* # *as*) *s* = (*case assert-s a s of Unsat I* ⇒ *Unsat I*
| *Inr s*′ ⇒ *assert-all-s as s*′)

lemma *assert-all-s-ok*: *invariant-s t as s* $\Longrightarrow$ *assert-all-s bs s* = *Inr s*′ $\Longrightarrow$
invariant-s t (*set bs* ∪ *as*) *s*′
⟨*proof*⟩

lemma *assert-all-s-unsat*: *invariant-s t as s* $\Longrightarrow$ *assert-all-s bs s* = *Unsat I* $\Longrightarrow$
minimal-unsat-core-tabl-atoms (*set I*) *t* (*as* ∪ *set bs*)
⟨*proof*⟩

end

Implementation of the interface via the Simplex operations init, check, and assert-bound.

locale *Incremental-State-Ops-Simplex* = *AssertBoundNoLhs assert-bound* + *Init init* + *Check check*
for *assert-bound* :: (′*i*,′*a*::*lrv*) *i-atom* ⇒ (′*i*,′*a*) *state* ⇒ (′*i*,′*a*) *state* **and**
init :: *tableau* ⇒ (′*i*,′*a*) *state* **and**
check :: (′*i*,′*a*) *state* ⇒ (′*i*,′*a*) *state*
begin

definition *weak-invariant-s* **where**
weak-invariant-s t (*as* :: (′*i*,′*a*)*i-atom set*) *s* =
($\models_{nolhs}$ *s* ∧
△ ($\mathcal{T}$ *s*) ∧
∇ *s* ∧
◊ *s* ∧
(∀ *v* :: (*var* ⇒ ′*a*). *v* $\models_t$ $\mathcal{T}$ *s* ⟷ *v* $\models_t$ *t*) ∧
index-valid as s ∧
Simplex.flat as ≐ $\mathcal{B}$ *s* ∧
as $\models_i$ $\mathcal{BI}$ *s*)

definition *invariant-s* **where**

invariant-s t (*as* :: ($'i$,$'a$)*i-atom set*) *s* =
(*weak-invariant-s t as s* ∧ ¬ $\mathcal{U}$ *s*)

definition *checked-s* **where**
checked-s t as s = (*invariant-s t as s* ∧ ⊨ *s*)

definition *assert-s* **where** *assert-s a s* = (*let* s' = *assert-bound a s in*
if $\mathcal{U}$ s' *then Inl* (*the* ($\mathcal{U}_c$ s')) *else Inr* s')

definition *check-s* **where** *check-s s* = (*let* s' = *check s in*
if $\mathcal{U}$ s' *then* (s', *Some* (*the* ($\mathcal{U}_c$ s'))) *else* (s', *None*))

definition *checkpoint-s* **where** *checkpoint-s s* = $\mathcal{B}_i$ *s*

fun *backtrack-s* :: - ⇒ ($'i$, $'a$) *state* ⇒ ($'i$, $'a$) *state*
where *backtrack-s* (*bl*, *bu*) (*State t bl-old bu-old v u uc*) = *State t bl bu v False None*

lemmas *invariant-defs* = *weak-invariant-s-def invariant-s-def checked-s-def*

lemma *invariant-sD*: **assumes** *invariant-s t as s*
shows ¬ $\mathcal{U}$ *s* $\models_{nolhs}$ *s* △ ($\mathcal{T}$ *s*) ∇ *s* ◇ *s*
Simplex.flat as $\doteq$ $\mathcal{B}$ *s as* $\models_i$ $\mathcal{BI}$ *s index-valid as s*
(∀ *v* :: (*var* ⇒ $'a$). *v* $\models_t$ $\mathcal{T}$ *s* ⟷ *v* $\models_t$ *t*)
⟨*proof*⟩

lemma *weak-invariant-sD*: **assumes** *weak-invariant-s t as s*
shows $\models_{nolhs}$ *s* △ ($\mathcal{T}$ *s*) ∇ *s* ◇ *s*
Simplex.flat as $\doteq$ $\mathcal{B}$ *s as* $\models_i$ $\mathcal{BI}$ *s index-valid as s*
(∀ *v* :: (*var* ⇒ $'a$). *v* $\models_t$ $\mathcal{T}$ *s* ⟷ *v* $\models_t$ *t*)
⟨*proof*⟩

lemma *minimal-unsat-state-core-translation*: **assumes**
unsat: *minimal-unsat-state-core* (*s* :: ($'i$,$'a$::*lrv*)*state*) **and**
tabl: ∀(*v* :: $'a$ *valuation*). *v* $\models_t$ $\mathcal{T}$ *s* = *v* $\models_t$ *t* **and**
index: *index-valid as s* **and**
imp: *as* $\models_i$ $\mathcal{BI}$ *s* **and**
I: *I* = *the* ($\mathcal{U}_c$ *s*)
shows *minimal-unsat-core-tabl-atoms* (*set I*) *t as*
⟨*proof*⟩

sublocale *Incremental-Atom-Ops*
init assert-s check-s $\mathcal{V}$ *checkpoint-s backtrack-s* △ *weak-invariant-s invariant-s checked-s*
⟨*proof*⟩

end

7.2 Intermediate Layer: Incremental Non-Strict Constraints

Interface

locale *Incremental-NS-Constraint-Ops* = **fixes**
init-nsc :: ('i,'a :: lrv) *i-ns-constraint list* ⇒ 's **and**
assert-nsc :: 'i ⇒ 's ⇒ 'i *list* + 's **and**
check-nsc :: 's ⇒ 's × ('i *list option*) **and**
solution-nsc :: 's ⇒ (*var*, 'a) *mapping* **and**
checkpoint-nsc :: 's ⇒ 'c **and**
backtrack-nsc :: 'c ⇒ 's ⇒ 's **and**
weak-invariant-nsc :: ('i,'a) *i-ns-constraint list* ⇒ 'i *set* ⇒ 's ⇒ *bool* **and**
invariant-nsc :: ('i,'a) *i-ns-constraint list* ⇒ 'i *set* ⇒ 's ⇒ *bool* **and**
checked-nsc :: ('i,'a) *i-ns-constraint list* ⇒ 'i *set* ⇒ 's ⇒ *bool*
assumes
assert-nsc-ok: *invariant-nsc nsc J s* ⟹ *assert-nsc j s* = *Inr s'* ⟹
invariant-nsc nsc (*insert j J*) *s'* **and**
assert-nsc-unsat: *invariant-nsc nsc J s* ⟹ *assert-nsc j s* = *Unsat I* ⟹
set I ⊆ *insert j J* ∧ *minimal-unsat-core-ns* (*set I*) (*set nsc*) **and**
check-nsc-ok: *invariant-nsc nsc J s* ⟹ *check-nsc s* = (*s'*, *None*) ⟹
checked-nsc nsc J s' **and**
check-nsc-unsat: *invariant-nsc nsc J s* ⟹ *check-nsc s* = (*s'*,*Some I*) ⟹
set I ⊆ *J* ∧ *weak-invariant-nsc nsc J s'* ∧ *minimal-unsat-core-ns* (*set I*) (*set nsc*) **and**
init-nsc: *checked-nsc nsc* {} (*init-nsc nsc*) **and**
solution-nsc: *checked-nsc nsc J s* ⟹ *solution-nsc s* = *v* ⟹ (*J*, ⟨*v*⟩) $\models_{inss}$ *set nsc* **and**
backtrack-nsc: *checked-nsc nsc J s* ⟹ *checkpoint-nsc s* = *c*
⟹ *weak-invariant-nsc nsc K s'* ⟹ *backtrack-nsc c s'* = *s''* ⟹ *J* ⊆ *K* ⟹ *invariant-nsc nsc J s''* **and**
weak-invariant-nsc: *invariant-nsc nsc J s* ⟹ *weak-invariant-nsc nsc J s* **and**
checked-invariant-nsc: *checked-nsc nsc J s* ⟹ *invariant-nsc nsc J s*

Implementation via the Simplex operation preprocess and the incremental operations for atoms.

fun *create-map* :: ('i × 'a)*list* ⇒ ('i, ('i × 'a) *list*)*mapping* **where**
create-map [] = *Mapping.empty*
| *create-map* ((*i*,*a*) # *xs*) = (*let m* = *create-map xs in*
case Mapping.lookup m i of
None ⇒ *Mapping.update i* [(*i*,*a*)] *m*
| *Some ias* ⇒ *Mapping.update i* ((*i*,*a*) # *ias*) *m*)

definition *list-map-to-fun* :: ('i, ('i × 'a) *list*)*mapping* ⇒ 'i ⇒ ('i × 'a) *list* **where**
list-map-to-fun m i = (*case Mapping.lookup m i of None* ⇒ [] | *Some ias* ⇒ *ias*)

lemma *list-map-to-fun-create-map*: *set* (*list-map-to-fun* (*create-map ias*) *i*) = *set ias* ∩ {*i*} × *UNIV*
⟨*proof*⟩

fun *prod-wrap* :: ('c ⇒ 's ⇒ 's × ('i *list option*))

$\Rightarrow$ $'c \times 's \Rightarrow ('c \times 's) \times ('i$ *list option*) **where**
prod-wrap f (*asi*,*s*) = (*case f asi s of* (s', *info*) $\Rightarrow$ ((*asi*,s'), *info*))

lemma *prod-wrap-def'*: *prod-wrap f* (*asi*,*s*) = *map-prod* (*Pair asi*) *id* (*f asi s*)
⟨*proof*⟩

locale *Incremental-Atom-Ops-For-NS-Constraint-Ops* =
Incremental-Atom-Ops init-s assert-s check-s solution-s checkpoint-s backtrack-s
△
weak-invariant-s invariant-s checked-s
\+ *Preprocess preprocess*
for
init-s :: *tableau* $\Rightarrow$ $'s$ **and**
assert-s :: ($'i$ :: *linorder*,$'a$ :: *lrv*) *i-atom* $\Rightarrow$ $'s$ $\Rightarrow$ $'i$ *list* + $'s$ **and**
check-s :: $'s \Rightarrow 's \times 'i$ *list option* **and**
solution-s :: $'s$ $\Rightarrow$ (*var*, $'a$) *mapping* **and**
checkpoint-s :: $'s \Rightarrow 'c$ **and**
backtrack-s :: $'c \Rightarrow 's \Rightarrow 's$ **and**
weak-invariant-s :: *tableau* $\Rightarrow$ ($'i$,$'a$) *i-atom set* $\Rightarrow$ $'s$ $\Rightarrow$ *bool* **and**
invariant-s :: *tableau* $\Rightarrow$ ($'i$,$'a$) *i-atom set* $\Rightarrow$ $'s$ $\Rightarrow$ *bool* **and**
checked-s :: *tableau* $\Rightarrow$ ($'i$,$'a$) *i-atom set* $\Rightarrow$ $'s$ $\Rightarrow$ *bool* **and**
preprocess :: ($'i$,$'a$) *i-ns-constraint list* $\Rightarrow$ *tableau* $\times$ ($'i$,$'a$) *i-atom list* $\times$ ((*var*,$'a$)*mapping* $\Rightarrow$ (*var*,$'a$)*mapping*) $\times$ $'i$ *list*
begin

definition *check-nsc* **where** *check-nsc* = *prod-wrap* (λ *asitv*. *check-s*)

definition *assert-nsc* **where** *assert-nsc i* = (λ ((*asi*,*tv*,*ui*),*s*).
if i $\in$ *set ui then Unsat* [*i*] *else*
case assert-all-s (*list-map-to-fun asi i*) *s of Unsat I* $\Rightarrow$ *Unsat I* | *Inr* s' $\Rightarrow$ *Inr* ((*asi*,*tv*,*ui*),s'))

fun *checkpoint-nsc* **where** *checkpoint-nsc* (*asi-tv-ui*,*s*) = *checkpoint-s s*
fun *backtrack-nsc* **where** *backtrack-nsc c* (*asi-tv-ui*,*s*) = (*asi-tv-ui*, *backtrack-s c s*)
fun *solution-nsc* **where** *solution-nsc* ((*asi*,*tv*,*ui*),*s*) = *tv* (*solution-s s*)

definition *init-nsc nsc* = (*case preprocess nsc of* (*t*,*as*,*trans-v*,*ui*) $\Rightarrow$
((*create-map as*, *trans-v*, *remdups ui*), *init-s t*))

fun *invariant-as-asi* **where** *invariant-as-asi as asi tc* tc' *ui* ui' = (*tc* = tc' $\wedge$ *set ui* = *set* ui' $\wedge$
($\forall$ *i*. *set* (*list-map-to-fun asi i*) = (*as* $\cap$ ({*i*} $\times$ *UNIV*))))

fun *weak-invariant-nsc* **where**
weak-invariant-nsc nsc J ((*asi*,*tv*,*ui*),*s*) = (*case preprocess nsc of* (*t*,*as*,tv',ui') $\Rightarrow$ *invariant-as-asi* (*set as*) *asi tv* tv' *ui* ui' $\wedge$
weak-invariant-s t (*set as* $\cap$ (*J* $\times$ *UNIV*)) *s* $\wedge$ *J* $\cap$ *set ui* = {})

fun *invariant-nsc* **where**
invariant-nsc nsc J $((asi,tv,ui),s) = ($*case preprocess nsc of* $(t,as,tv',ui') \Rightarrow$ *invariant-as-asi* $($*set as*$)$ *asi tv tv' ui ui'* $\wedge$
invariant-s t $($*set as* $\cap (J \times UNIV))$ *s* $\wedge J \cap$ *set ui* $= \{\})$

fun *checked-nsc* **where**
checked-nsc nsc J $((asi,tv,ui),s) = ($*case preprocess nsc of* $(t,as,tv',ui') \Rightarrow$ *invariant-as-asi* $($*set as*$)$ *asi tv tv' ui ui'* $\wedge$
checked-s t $($*set as* $\cap (J \times UNIV))$ *s* $\wedge J \cap$ *set ui* $= \{\})$

lemma *i-satisfies-atom-set-inter-right*: $((I, v) \models_{ias} (ats \cap (J \times UNIV))) \longleftrightarrow ((I \cap J, v) \models_{ias} ats)$
⟨*proof*⟩

lemma *ns-constraints-ops*: *Incremental-NS-Constraint-Ops init-nsc assert-nsc check-nsc solution-nsc checkpoint-nsc backtrack-nsc weak-invariant-nsc invariant-nsc checked-nsc*
⟨*proof*⟩

end

7.3 Highest Layer: Incremental Constraints

Interface

locale *Incremental-Simplex-Ops* = **fixes**
init-cs :: *'i i-constraint list* $\Rightarrow$ *'s* **and**
assert-cs :: *'i* $\Rightarrow$ *'s* $\Rightarrow$ *'i list* + *'s* **and**
check-cs :: *'s* $\Rightarrow$ *'s* $\times$ *'i list option* **and**
solution-cs :: *'s* $\Rightarrow$ *rat valuation* **and**
checkpoint-cs :: *'s* $\Rightarrow$ *'c* **and**
backtrack-cs :: *'c* $\Rightarrow$ *'s* $\Rightarrow$ *'s* **and**
weak-invariant-cs :: *'i i-constraint list* $\Rightarrow$ *'i set* $\Rightarrow$ *'s* $\Rightarrow$ *bool* **and**
invariant-cs :: *'i i-constraint list* $\Rightarrow$ *'i set* $\Rightarrow$ *'s* $\Rightarrow$ *bool* **and**
checked-cs :: *'i i-constraint list* $\Rightarrow$ *'i set* $\Rightarrow$ *'s* $\Rightarrow$ *bool*
assumes
assert-cs-ok: *invariant-cs cs J s* $\Longrightarrow$ *assert-cs j s* = *Inr s'* $\Longrightarrow$
invariant-cs cs $($*insert j J*$)$ *s'* **and**
assert-cs-unsat: *invariant-cs cs J s* $\Longrightarrow$ *assert-cs j s* = *Unsat I* $\Longrightarrow$
set I $\subseteq$ *insert j J* $\wedge$ *minimal-unsat-core* $($*set I*$)$ *cs* **and**
check-cs-ok: *invariant-cs cs J s* $\Longrightarrow$ *check-cs s* = $(s', None) \Longrightarrow$
checked-cs cs J s' **and**
check-cs-unsat: *invariant-cs cs J s* $\Longrightarrow$ *check-cs s* = $(s', Some\ I) \Longrightarrow$
weak-invariant-cs cs J s' $\wedge$ *set I* $\subseteq J \wedge$ *minimal-unsat-core* $($*set I*$)$ *cs* **and**
init-cs: *checked-cs cs* $\{\}$ $($*init-cs cs*$)$ **and**
solution-cs: *checked-cs cs J s* $\Longrightarrow$ *solution-cs s* = $v \Longrightarrow (J, v) \models_{ics}$ *set cs* **and**
backtrack-cs: *checked-cs cs J s* $\Longrightarrow$ *checkpoint-cs s* = *c*
$\Longrightarrow$ *weak-invariant-cs cs K s'* $\Longrightarrow$ *backtrack-cs c s'* = $s'' \Longrightarrow J \subseteq K \Longrightarrow$

invariant-cs cs J s'' **and**
weak-invariant-cs: *invariant-cs cs J s* $\Longrightarrow$ *weak-invariant-cs cs J s* **and**
checked-invariant-cs: *checked-cs cs J s* $\Longrightarrow$ *invariant-cs cs J s*

Implementation via the Simplex-operation To-Ns and the Incremental Operations for Non-Strict Constraints

locale *Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex* =
Incremental-NS-Constraint-Ops init-nsc assert-nsc check-nsc solution-nsc checkpoint-nsc backtrack-nsc
weak-invariant-nsc invariant-nsc checked-nsc + *To-ns to-ns from-ns*
for
init-nsc :: ('*i*,'*a* :: *lrv*) *i-ns-constraint list* ⇒ '*s* **and**
assert-nsc :: '*i* ⇒ '*s* ⇒ '*i list* + '*s* **and**
check-nsc :: '*s* ⇒ '*s* × '*i list option* **and**
solution-nsc :: '*s* ⇒ (*var*, '*a*) *mapping* **and**
checkpoint-nsc :: '*s* ⇒ '*c* **and**
backtrack-nsc :: '*c* ⇒ '*s* ⇒ '*s* **and**
weak-invariant-nsc :: ('*i*,'*a*) *i-ns-constraint list* ⇒ '*i set* ⇒ '*s* ⇒ *bool* **and**
invariant-nsc :: ('*i*,'*a*) *i-ns-constraint list* ⇒ '*i set* ⇒ '*s* ⇒ *bool* **and**
checked-nsc :: ('*i*,'*a*) *i-ns-constraint list* ⇒ '*i set* ⇒ '*s* ⇒ *bool* **and**
to-ns :: '*i i-constraint list* ⇒ ('*i*,'*a*) *i-ns-constraint list* **and**
from-ns :: (*var*, '*a*) *mapping* ⇒ '*a ns-constraint list* ⇒ (*var*, *rat*) *mapping*
begin

fun *assert-cs* **where** *assert-cs i* (*cs*,*s*) = (*case assert-nsc i s of*
Unsat I ⇒ *Unsat I*
| *Inr s'* ⇒ *Inr* (*cs*, *s'*))

definition *init-cs cs* = (*let tons-cs* = *to-ns cs in* (*map snd* (*tons-cs*), *init-nsc tons-cs*))

definition *check-cs s* = *prod-wrap* (λ *cs*. *check-nsc*) *s*
fun *checkpoint-cs* **where** *checkpoint-cs* (*cs*,*s*) = (*checkpoint-nsc s*)
fun *backtrack-cs* **where** *backtrack-cs c* (*cs*,*s*) = (*cs*, *backtrack-nsc c s*)
fun *solution-cs* **where** *solution-cs* (*cs*,*s*) = (⟨*from-ns* (*solution-nsc s*) *cs*⟩)

fun *weak-invariant-cs* **where**
weak-invariant-cs cs J (*ds*,*s*) = (*ds* = *map snd* (*to-ns cs*) ∧ *weak-invariant-nsc* (*to-ns cs*) *J s*)
fun *invariant-cs* **where**
invariant-cs cs J (*ds*,*s*) = (*ds* = *map snd* (*to-ns cs*) ∧ *invariant-nsc* (*to-ns cs*) *J s*)
fun *checked-cs* **where**
checked-cs cs J (*ds*,*s*) = (*ds* = *map snd* (*to-ns cs*) ∧ *checked-nsc* (*to-ns cs*) *J s*)

sublocale *Incremental-Simplex-Ops*
init-cs
assert-cs
check-cs

solution-cs
checkpoint-cs
backtrack-cs
weak-invariant-cs
invariant-cs
checked-cs
⟨*proof*⟩

end

7.4 Concrete Implementation

7.4.1 Connecting all the locales

global-interpretation *Incremental-State-Ops-Simplex-Default*:
Incremental-State-Ops-Simplex assert-bound-code init-state check-code
defines *assert-s = Incremental-State-Ops-Simplex-Default.assert-s* **and**
check-s = Incremental-State-Ops-Simplex-Default.check-s **and**
backtrack-s = Incremental-State-Ops-Simplex-Default.backtrack-s **and**
checkpoint-s = Incremental-State-Ops-Simplex-Default.checkpoint-s **and**
weak-invariant-s = Incremental-State-Ops-Simplex-Default.weak-invariant-s
and
invariant-s = Incremental-State-Ops-Simplex-Default.invariant-s **and**
checked-s = Incremental-State-Ops-Simplex-Default.checked-s **and**
assert-all-s = Incremental-State-Ops-Simplex-Default.assert-all-s
⟨*proof*⟩

lemma *Incremental-State-Ops-Simplex-Default-assert-all-s*[*simp*]:
Incremental-State-Ops-Simplex-Default.assert-all-s = assert-all-s
⟨*proof*⟩

lemmas *assert-all-s-code = Incremental-State-Ops-Simplex-Default.assert-all-s.simps*[*unfolded*

Incremental-State-Ops-Simplex-Default-assert-all-s]

declare *assert-all-s-code*[*code*]

global-interpretation *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default*:
Incremental-Atom-Ops-For-NS-Constraint-Ops init-state assert-s check-s $\mathcal{V}$
checkpoint-s backtrack-s weak-invariant-s invariant-s checked-s preprocess
defines
init-nsc = Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.init-nsc **and**
check-nsc = Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.check-nsc
and
assert-nsc = Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.assert-nsc
and
checkpoint-nsc = Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.checkpoint-nsc
and

solution-nsc = *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.solution-nsc*
and
backtrack-nsc = *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.backtrack-nsc*
and
invariant-nsc = *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.invariant-nsc*
and
weak-invariant-nsc = *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.weak-invariant-nsc*
and
checked-nsc = *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.checked-nsc*

⟨*proof*⟩

type-synonym *'i simplex-state'* = *QDelta ns-constraint list*
× ((*'i*, (*'i* × *QDelta atom*) *list*) *mapping* × ((*var*,*QDelta*)*mapping* ⇒ (*var*,*QDelta*)*mapping*)
× *'i list*)
× (*'i*, *QDelta*) *state*

global-interpretation *Incremental-Simplex*:
Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex
init-nsc assert-nsc check-nsc solution-nsc checkpoint-nsc backtrack-nsc
weak-invariant-nsc invariant-nsc checked-nsc to-ns from-ns
defines
init-simplex' = *Incremental-Simplex.init-cs* **and**
assert-simplex' = *Incremental-Simplex.assert-cs* **and**
check-simplex' = *Incremental-Simplex.check-cs* **and**
backtrack-simplex' = *Incremental-Simplex.backtrack-cs* **and**
checkpoint-simplex' = *Incremental-Simplex.checkpoint-cs* **and**
solution-simplex' = *Incremental-Simplex.solution-cs* **and**
weak-invariant-simplex' = *Incremental-Simplex.weak-invariant-cs* **and**
invariant-simplex' = *Incremental-Simplex.invariant-cs* **and**
checked-simplex' = *Incremental-Simplex.checked-cs*
⟨*proof*⟩

7.4.2 An implementation which encapsulates the state

In principle, we now already have a complete implementation of the incremental simplex algorithm with *init-simplex'*, *assert-simplex'*, etc. However, this implementation results in code where the interal type *'i simplex-state'* becomes visible. Therefore, we now define all operations on a new type which encapsulates the internal construction.

datatype *'i simplex-state* = *Simplex-State 'i simplex-state'*
datatype *'i simplex-checkpoint* = *Simplex-Checkpoint* (*nat*, *'i* × *QDelta*) *mapping*
× (*nat*, *'i* × *QDelta*) *mapping*

fun *init-simplex* **where** *init-simplex cs* =
(*let tons-cs* = *to-ns cs*
in Simplex-State (*map snd tons-cs*,

case preprocess tons-cs of (*t*, *as*, *trans-v*, *ui*) ⇒ ((*create-map as*, *trans-v*, *remdups ui*), *init-state t*)))

fun *assert-simplex* **where** *assert-simplex i* (*Simplex-State* (*cs*, (*asi*, *tv*, *ui*), *s*)) =
(*if i* ∈ *set ui then Inl* [*i*] *else*
case assert-all-s (*list-map-to-fun asi i*) *s of*
Inl y ⇒ *Inl y* | *Inr s′* ⇒ *Inr* (*Simplex-State* (*cs*, (*asi*, *tv*, *ui*), *s′*)))

fun *check-simplex* **where**
check-simplex (*Simplex-State* (*cs*, *asi-tv*, *s*)) = (*case check-s s of* (*s′*, *res*) ⇒
(*Simplex-State* (*cs*, *asi-tv*, *s′*), *res*))

fun *solution-simplex* **where**
solution-simplex (*Simplex-State* (*cs*, (*asi*, *tv*, *ui*), *s*)) = ⟨*from-ns* (*tv* ($\mathcal{V}$ *s*)) *cs*⟩

fun *checkpoint-simplex* **where** *checkpoint-simplex* (*Simplex-State* (*cs*, *asi-tv*, *s*)) = *Simplex-Checkpoint* (*checkpoint-s s*)

fun *backtrack-simplex* **where**
backtrack-simplex (*Simplex-Checkpoint c*) (*Simplex-State* (*cs*, *asi-tv*, *s*)) = *Simplex-State* (*cs*, *asi-tv*, *backtrack-s c s*)

7.4.3 Soundness of the incremental simplex implementation

First link the unprimed constants against their primed counterparts.

lemma *init-simplex′*: *init-simplex cs* = *Simplex-State* (*init-simplex′ cs*)
⟨*proof*⟩

lemma *assert-simplex′*: *assert-simplex i* (*Simplex-State s*) = *map-sum id Simplex-State* (*assert-simplex′ i s*)
⟨*proof*⟩

lemma *check-simplex′*: *check-simplex* (*Simplex-State s*) = *map-prod Simplex-State id* (*check-simplex′ s*)
⟨*proof*⟩

lemma *solution-simplex′*: *solution-simplex* (*Simplex-State s*) = *solution-simplex′ s*

⟨*proof*⟩

lemma *checkpoint-simplex′*: *checkpoint-simplex* (*Simplex-State s*) = *Simplex-Checkpoint* (*checkpoint-simplex′ s*)
⟨*proof*⟩

lemma *backtrack-simplex′*: *backtrack-simplex* (*Simplex-Checkpoint c*) (*Simplex-State s*) = *Simplex-State* (*backtrack-simplex′ c s*)
⟨*proof*⟩

fun *invariant-simplex* **where**

invariant-simplex cs J (*Simplex-State s*) = *invariant-simplex′ cs J s*

fun *weak-invariant-simplex* **where**
weak-invariant-simplex cs J (*Simplex-State s*) = *weak-invariant-simplex′ cs J s*

fun *checked-simplex* **where**
checked-simplex cs J (*Simplex-State s*) = *checked-simplex′ cs J s*

Hide implementation

declare *init-simplex.simps*[*simp del*]
declare *assert-simplex.simps*[*simp del*]
declare *check-simplex.simps*[*simp del*]
declare *solution-simplex.simps*[*simp del*]
declare *checkpoint-simplex.simps*[*simp del*]
declare *backtrack-simplex.simps*[*simp del*]

Soundness lemmas

lemma *init-simplex*: *checked-simplex cs* {} (*init-simplex cs*)
⟨*proof*⟩

lemma *assert-simplex-ok*:
invariant-simplex cs J s ⟹ *assert-simplex j s* = *Inr s′* ⟹ *invariant-simplex cs* (*insert j J*) *s′*
⟨*proof*⟩

lemma *assert-simplex-unsat*:
invariant-simplex cs J s ⟹ *assert-simplex j s* = *Inl I* ⟹
set I ⊆ *insert j J* ∧ *minimal-unsat-core* (*set I*) *cs*
⟨*proof*⟩

lemma *check-simplex-ok*:
invariant-simplex cs J s ⟹ *check-simplex s* = (*s′*,*None*) ⟹ *checked-simplex cs J s′*
⟨*proof*⟩

lemma *check-simplex-unsat*:
invariant-simplex cs J s ⟹ *check-simplex s* = (*s′*,*Some I*) ⟹
weak-invariant-simplex cs J s′ ∧ *set I* ⊆ *J* ∧ *minimal-unsat-core* (*set I*) *cs*
⟨*proof*⟩

lemma *solution-simplex*:
checked-simplex cs J s ⟹ *solution-simplex s* = *v* ⟹ (*J*, *v*) $\models_{ics}$ *set cs*
⟨*proof*⟩

lemma *backtrack-simplex*:
checked-simplex cs J s ⟹
checkpoint-simplex s = *c* ⟹
weak-invariant-simplex cs K s′ ⟹
backtrack-simplex c s′ = *s′′* ⟹

$J \subseteq K \Longrightarrow$
invariant-simplex cs J s''
⟨*proof*⟩

lemma *weak-invariant-simplex*:
invariant-simplex cs J s ⟹ *weak-invariant-simplex cs J s*
⟨*proof*⟩

lemma *checked-invariant-simplex*:
checked-simplex cs J s ⟹ *invariant-simplex cs J s*
⟨*proof*⟩

declare *checked-simplex.simps*[*simp del*]
declare *invariant-simplex.simps*[*simp del*]
declare *weak-invariant-simplex.simps*[*simp del*]

From this point onwards, one should not look into the types *'i simplex-state* and *'i simplex-checkpoint*.

For convenience: an assert-all function which takes multiple indices.

fun *assert-all-simplex* :: *'i list* ⇒ *'i simplex-state* ⇒ *'i list* + *'i simplex-state* **where**
assert-all-simplex [] *s* = *Inr s*
| *assert-all-simplex* (*j* # *J*) *s* = (*case assert-simplex j s of Unsat I* ⇒ *Unsat I*
| *Inr s'* ⇒ *assert-all-simplex J s'*)

lemma *assert-all-simplex-ok*: *invariant-simplex cs J s* ⟹ *assert-all-simplex K s* = *Inr s'* ⟹
invariant-simplex cs (*J* ∪ *set K*) *s'*
⟨*proof*⟩

lemma *assert-all-simplex-unsat*: *invariant-simplex cs J s* ⟹ *assert-all-simplex K s* = *Unsat I* ⟹
set I ⊆ *set K* ∪ *J* ∧ *minimal-unsat-core* (*set I*) *cs*
⟨*proof*⟩

The collection of soundness lemmas for the incremental simplex algorithm.

lemmas *incremental-simplex* =
init-simplex
assert-simplex-ok
assert-simplex-unsat
assert-all-simplex-ok
assert-all-simplex-unsat
check-simplex-ok
check-simplex-unsat
solution-simplex
backtrack-simplex
checked-invariant-simplex
weak-invariant-simplex

7.5 Test Executability and Example for Incremental Interface

value (*code*) *let cs* = [
(*1* :: *int*, *LT* (*lp-monom 1 1*) *4*), — $x_1 < 4$
(*2*, *GT* (*lp-monom 2 1* − *lp-monom 1 2*) *0*), — $2x_1 - x_2 > 0$
(*3*, *EQ* (*lp-monom 1 1* − *lp-monom 2 2*) *0*), — $x_1 - 2x_2 = 0$
(*4*, *GT* (*lp-monom 2 2*) *5*), — $2x_2 > 5$
(*5*, *GT* (*lp-monom 3 0*) *7*), — $3x_0 > 7$
(*6*, *GT* (*lp-monom 3 3* + *lp-monom* (*1/3*) *2*) *2*)]; — $3x_3 + 1/3x_2 > 2$
s1 = *init-simplex cs*; — initialize
s2 = (*case assert-all-simplex* [*1*,*2*,*3*] *s1 of Inr s* ⇒ *s* | *Unsat* - ⇒ *undefined*); — assert 1,2,3
s3 = (*case check-simplex s2 of* (*s*,*None*) ⇒ *s* | - ⇒ *undefined*); — check that 1,2,3 are sat.
c123 = *checkpoint-simplex s3*; — after check, store checkpoint for backtracking
s4 = (*case assert-simplex 4 s2 of Inr s* ⇒ *s* | *Unsat* - ⇒ *undefined*); — assert 4
(*s5*,*I*) = (*case check-simplex s4 of* (*s*,*Some I*) ⇒ (*s*,*I*) | - ⇒ *undefined*); — checking detects unsat-core 1,3,4
s6 = *backtrack-simplex c123 s5*; — backtrack to constraints 1,2,3
s7 = (*case assert-all-simplex* [*5*,*6*] *s6 of Inr s* ⇒ *s* | *Unsat* - ⇒ *undefined*); — assert 5,6
s8 = (*case check-simplex s7 of* (*s*,*None*) ⇒ *s* | - ⇒ *undefined*); — check that 1,2,3,5,6 are sat.
sol = *solution-simplex s8* — solution for 1,2,3,5,6
in (*I*, *map* (λ *x*. (''*x-*'', *x*, ''*=*'', *sol x*)) [*0*,*1*,*2*,*3*]) — output unsat core and solution
end

References

[1] B. Dutertre and L. 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.

[2] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP'13*, volume 7998 of *LNCS*, pages 100–115, 2013.

[3] 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.