

An Incremental Simplex Algorithm with Unsatisfiable Core Generation^{*}

Filip Marić Mirko Spasić René Thiemann

March 19, 2025

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

1	Introduction	2
2	Auxiliary Results	3
3	Linearly Ordered Rational Vectors	14
4	Linear Polynomials and Constraints	20
5	Rational Numbers Extended with Infinitesimal Element	37
6	The Simplex Algorithm	41
6.1	Procedure Specification	43
6.2	Handling Strict Inequalities	44
6.3	Preprocessing	46
6.4	Incrementally Asserting Atoms	55
6.5	Asserting Single Atoms	71

^{*}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.

6.6	Update and Pivot	83
6.7	Check implementation	121
6.8	Symmetries	179
6.9	Concrete implementation	180
7	The Incremental Simplex Algorithm	217
7.1	Lowest Layer: Fixed Tableau and Incremental Atoms	218
7.2	Intermediate Layer: Incremental Non-Strict Constraints	225
7.3	Highest Layer: Incremental Constraints	230
7.4	Concrete Implementation	233
7.4.1	Connecting all the locales	233
7.4.2	An implementation which encapsulates the state	235
7.4.3	Soundness of the incremental simplex implementation	235
7.5	Test Executability and Example for Incremental Interface	239

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 "i < length l. g (l ! i) = f i"
  shows "map f [0.. l] = map g l"
  using assms
  by (induct l rule: rev-induct) (auto simp add: nth-append split: if-splits)

lemma map-parametrize-idx:
  map f l = map (λi. f (l ! i)) [0.. l]
  by (induct l rule: rev-induct) (auto simp add: nth-append)

lemma last-tl:
  assumes "length l > 1"
  shows "last (tl l) = last l"
  using assms
  by (induct l) auto

lemma hd-tl:
  assumes "length l > 1"
  shows "hd (tl l) = l ! 1"
  using assms
  by (induct l) (auto simp add: hd-conv-nth)

lemma butlast-empty-conv-length:
  shows "(butlast l = []) = (length l ≤ 1)"
  by (induct l) (auto split: if-splits)

lemma butlast-nth:
  assumes "n + 1 < length l"
  shows "butlast l ! n = l ! n"
  using assms
  by (induct l rule: rev-induct) (auto simp add: nth-append)

lemma last-take-conv-nth:
  assumes "0 < n n ≤ length l"
  shows "last (take n l) = l ! (n - 1)"
  using assms
  by (cases l = []) (auto simp add: last-conv-nth min-def)

lemma tl-nth:

```

```

assumes l ≠ []
shows tl l ! n = l ! (n + 1)
using assms
by (induct l) auto

lemma interval-3split:
assumes i < n
shows [0..<n] = [0..<i] @ [i] @ [i+1..<n]
proof-
have [0..<n] = [0..<i + 1] @ [i + 1..<n]
using upt-add-eq-append[of 0 i + 1 n - i - 1]
using ‹i < n›
by (auto simp del: upt-Suc)
then show ?thesis
by simp
qed

abbreviation list-min l ≡ foldl min (hd l) (tl l)
lemma list-min-Min[simp]: l ≠ [] ⟹ list-min l = Min (set l)
proof (induct l rule: rev-induct)
case (snoc a l')
then show ?case
by (cases l' = []) (auto simp add: ac-simps)
qed simp

```

```

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 ∈ set l. ¬ P x)
unfolding min-satisfying-def Let-def
by (simp add: filter-empty-conv)

lemma min-satisfying-Some:
min-satisfying P l = Some x ⟶
x ∈ set l ∧ P x ∧ (∀ x' ∈ set l. x' < x ⟶ ¬ P x')
proof (safe)
let ?xs = filter P l
assume min-satisfying P l = Some x
then have set ?xs ≠ {} x = Min (set ?xs)
unfolding min-satisfying-def Let-def
by (auto split: if-splits simp add: filter-empty-conv)
then show x ∈ set l P x
using Min-in[of set ?xs]

```

```

by simp-all
fix x'
assume x' ∈ set l P x' x' < x
have x' ∉ set ?xs
proof (rule ccontr)
  assume ¬ ?thesis
  then have x' ≥ x
  using ⟨x = Min (set ?xs)⟩
  by simp
  then show False
  using ⟨x' < x⟩
  by simp
qed
then show False
using ⟨x' ∈ set l. ⟨P x'⟩
by simp
qed

```

```

lemma min-element:
  fixes k :: nat
  assumes ∃ (m::nat). P m
  shows ∃ mm. P mm ∧ (∀ m'. m' < mm → ¬ P m')
proof –
  from assms obtain m where P m
  by auto
  show ?thesis
  proof (cases ∀ m' < m. ¬ P m')
    case True
    then show ?thesis
    using ⟨P m⟩
    by auto
  next
    case False
    then show ?thesis
    proof (induct m)
      case 0
      then show ?case
      by auto
    next
      case (Suc m')
      then show ?case
      by (cases ¬ (∀ m'a < m'. ¬ P m'a)) auto
    qed
  qed
qed

```

```

lemma finite-fun-args:
  assumes finite A  $\forall a \in A. \text{finite}(B a)$ 
  shows finite {f. ( $\forall a. \text{if } a \in A \text{ then } f a \in B a \text{ else } f a = f_0 a$ )} (is finite (?F A))
  using assms
proof (induct)
  case empty
  have ?F {} = { $\lambda x. f_0 x$ }
    by auto
  then show ?case
    by auto
next
  case (insert a A')
  then have finite (?F A')
    by auto
  let ?f =  $\lambda f. \{f'. (\forall a'. \text{if } a = a' \text{ then } f' a \in B a \text{ else } f' a' = f a')\}$ 
  have  $\forall f \in ?F A'. \text{finite}(\text{?ff})$ 
  proof
    fix f
    assume f ∈ ?F A'
    then have ?ff =  $(\lambda b. f(a := b))`B a$ 
      by (force split: if-splits)
    then show finite (?ff)
      using  $\forall a \in \text{insert } a A'. \text{finite}(B a)$ 
      by auto
  qed
  then have finite ( $\bigcup (\text{?f}`(?F A'))$ )
    using finite (?F A')
    by auto
moreover
  have ?F (insert a A') =  $\bigcup (\text{?f}`(?F A'))$ 
  proof
    show ?F (insert a A') ⊆  $\bigcup (\text{?f}`(?F A'))$ 
    proof
      fix f
      assume f ∈ ?F (insert a A')
      then have f ∈ ?f (f(a := f_0 a)) f(a := f_0 a) ∈ ?F A'
        using  $a \notin A'$ 
        by auto
      then show f ∈  $\bigcup (\text{?f}`(?F A'))$ 
        by blast
    qed
  next
    show  $\bigcup (\text{?f}`(?F A')) \subseteq ?F (\text{insert } a A')$ 
    proof

```

```

fix f
assume f ∈ ∪ (?f ‘ (?F A'))
then obtain f0 where f0 ∈ ?F A' f ∈ ?f f0
  by auto
then show f ∈ ?F (insert a A')
  using `a ∉ A'
  by (force split: if-splits)
qed
qed
ultimately
show ?case
  by simp
qed

lemma foldl-mapping-update:
assumes X ∈ set l distinct (map f l)
shows Mapping.lookup (foldl (λm a. Mapping.update (f a) (g a) m) i l) (f X) =
Some (g X)
using assms
proof(induct l rule:rev-induct)
case Nil
then show ?case
  by simp
next
case (snoc h t)
show ?case
proof (cases f h = f X)
case True
then show ?thesis using snoc by (auto simp: lookup-update)
next
case False
show ?thesis by (simp add: lookup-update' False, rule snoc, insert False snoc,
auto)
qed
qed

end

theory Rel-Chain
imports
  Simplex-Auxiliary
begin

definition
rel-chain :: 'a list ⇒ ('a × 'a) set ⇒ bool

```

where
 $\text{rel-chain } l \ r = (\forall k < \text{length } l - 1. (l ! k, l ! (k + 1)) \in r)$

lemma *rel-chain-Nil*: $\text{rel-chain } [] \ r$ **and**
rel-chain-Cons: $\text{rel-chain } (x \ # \ xs) \ r = (\text{if } xs = [] \text{ then True else } ((x, \text{hd } xs) \in r) \wedge \text{rel-chain } xs \ r)$
by (auto simp add: rel-chain-def hd-conv-nth nth-Cons split: nat.split-asm nat.split)

lemma *rel-chain-drop*:
 $\text{rel-chain } l \ R ==> \text{rel-chain } (\text{drop } n \ l) \ R$
unfolding rel-chain-def
by simp

lemma *rel-chain-take*:
 $\text{rel-chain } l \ R ==> \text{rel-chain } (\text{take } n \ l) \ R$
unfolding rel-chain-def
by simp

lemma *rel-chain-butlast*:
 $\text{rel-chain } l \ R ==> \text{rel-chain } (\text{butlast } l) \ R$
unfolding rel-chain-def
by (auto simp add: butlast-nth)

lemma *rel-chain-tl*:
 $\text{rel-chain } l \ R ==> \text{rel-chain } (\text{tl } l) \ R$
unfolding rel-chain-def
by (cases l = []) (auto simp add: tl-nth)

lemma *rel-chain-append*:
assumes $\text{rel-chain } l \ R \ \text{rel-chain } l' \ R \ (\text{last } l, \text{hd } l') \in R$
shows $\text{rel-chain } (l @ l') \ R$
using assms
by (induct l) (auto simp add: rel-chain-Cons split: if-splits)

lemma *rel-chain-appendD*:
assumes $\text{rel-chain } (l @ l') \ R$
shows $\text{rel-chain } l \ R \ \text{rel-chain } l' \ R \ l \neq [] \wedge l' \neq [] \longrightarrow (\text{last } l, \text{hd } l') \in R$
using assms
by (induct l) (auto simp add: rel-chain-Cons rel-chain-Nil split: if-splits)

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

proof
assume ?lhs
then show ?rhs
by (induct rule: converse-rtrancl-induct) (auto simp add: rel-chain-Cons)

next

```

assume ?rhs
then obtain l where l ≠ [] hd l = x last l = y rel-chain l R
  by auto
then show ?lhs
  by (induct l arbitrary: x) (auto simp add: rel-chain-Cons, force)
qed

lemma trancr-rel-chain:
  (x, y) ∈ R+  $\longleftrightarrow$  (exists l. l ≠ [] ∧ length l > 1 ∧ hd l = x ∧ last l = y ∧ rel-chain l R) (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  then obtain z where (x, z) ∈ R (z, y) ∈ R*
    by (auto dest: trancrD)
  then obtain l where l ≠ [] ∧ hd l = z ∧ last l = y ∧ rel-chain l R
    by (auto simp add: rtrancr-rel-chain)
  then show ?rhs
    using ⟨(x, z) ∈ R⟩
    by (rule-tac x=x # l in exI) (auto simp add: rel-chain-Cons)
next
  assume ?rhs
  then obtain l where 1 < length l l ≠ [] hd l = x last l = y rel-chain l R
    by auto
  then obtain l' where
    l' ≠ [] l = x # l' (x, hd l') ∈ R rel-chain l' R
    using ⟨1 < length l⟩
    by (cases l) (auto simp add: rel-chain-Cons)
  then have (x, hd l') ∈ R (hd l', y) ∈ R*
    using ⟨last l = y⟩
    by (auto simp add: rtrancr-rel-chain)
  then show ?lhs
    by auto
qed

lemma rel-chain-elems-rtrancr:
  assumes rel-chain l R i ≤ j j < length l
  shows (l ! i, l ! j) ∈ R*
proof (cases i = j)
  case True
  then show ?thesis
  by simp
next
  case False
  then have i < j
  using ⟨i ≤ j⟩
  by simp
  then have l ≠ []
  using ⟨j < length l⟩
  by auto

```

```

let ?l = drop i (take (j + 1) l)

have ?l ≠ []
  using ⟨i < j⟩ ⟨j < length l⟩
  by simp
moreover
have hd ?l = l ! i
  using ⟨?l ≠ []⟩ ⟨i < j⟩
  by (auto simp add: hd-conv-nth)
moreover
have last ?l = l ! j
  using ⟨?l ≠ []⟩ ⟨l ≠ []⟩ ⟨i < j⟩ ⟨j < length l⟩
  by (cases length l = j + 1) (auto simp add: last-conv-nth min-def)
moreover
have rel-chain ?l R
  using ⟨rel-chain l R⟩
  by (auto intro: rel-chain-drop rel-chain-take)
ultimately
show ?thesis
  by (subst rtrancl-rel-chain) blast
qed

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-chain l' r length l' = length l - 1
    ∀ i. i + 1 < length l' →
      (exists j. j + 1 < length l ∧ l' ! i = l ! j ∧ l' ! (i + 1) = l ! (j + 1))
proof-
have l ≠ []
  using ⟨length l > 2⟩
  by auto

have length (tl l) > 1 tl l ≠ []
  using ⟨length l > 2⟩
  by (auto simp add: length-0-conv[THEN sym])

let ?l' = if sl = 0 then
  tl l
  else
    drop (sl + 1) l @ tl (take (sl + 1) l)

have hd ?l' = l ! (sl + 1)
proof (cases sl > 0, simp-all)
show hd (tl l) = l ! (Suc 0)
  using ⟨tl l ≠ []⟩ ⟨l ≠ []⟩
  by (simp add: hd-conv-nth tl-nth)

```

```

next
assume  $0 < sl$ 
show  $hd (drop (Suc sl) l @ tl (take (Suc sl) l)) = l ! (Suc sl)$ 
using  $\langle sl + 1 < length l \rangle \langle l \neq [] \rangle$ 
by (auto simp add: hd-append hd-drop-conv-nth)
qed

moreover

have  $last ?l' = l ! sl$ 
proof (cases  $sl > 0$ , simp-all)
show  $last (tl l) = l ! 0$ 
using  $\langle l \neq [] \rangle \langle last l = s \rangle \langle hd l = s \rangle \langle length l > 2 \rangle$ 
by (simp add: hd-conv-nth last-tl)
next
assume  $sl > 0$ 
then show  $last (drop (Suc sl) l @ tl (take (Suc sl) l)) = l ! sl$ 
using  $\langle l \neq [] \rangle \langle tl l \neq [] \rangle \langle sl + 1 < length l \rangle$ 
by (auto simp add: last-append drop-Suc tl-take last-take-conv-nth tl-nth)
qed

moreover

have  $rel-chain ?l' r$ 
proof (cases  $sl = 0$ , simp-all)
case True
show  $rel-chain (tl l) r$ 
using  $\langle rel-chain l r \rangle$ 
by (auto intro: rel-chain-tl)
next
assume  $sl > 0$ 
show  $rel-chain (drop (Suc sl) l @ tl (take (Suc sl) l)) r$ 
proof (rule rel-chain-append)
show  $rel-chain (drop (Suc sl) l) r$ 
using  $\langle rel-chain l r \rangle$ 
by (auto intro: rel-chain-drop)
next
show  $rel-chain (tl (take (Suc sl) l)) r$ 
using  $\langle rel-chain l r \rangle$ 
by (auto intro: rel-chain-tl rel-chain-take)
next
have  $last (drop (sl + 1) l) = l ! 0$ 
using  $\langle sl + 1 < length l \rangle \langle last l = s \rangle \langle hd l = s \rangle \langle l \neq [] \rangle$ 
by (auto simp add: hd-conv-nth)
moreover
have  $sl > 0 \longrightarrow tl (take (sl + 1) l) \neq []$ 
using  $\langle sl + 1 < length l \rangle \langle l \neq [] \rangle \langle tl l \neq [] \rangle$ 
by (auto simp add: take-Suc)
then have  $sl > 0 \longrightarrow hd (tl (take (sl + 1) l)) = l ! 1$ 

```

```

using ‹l ≠ []›
by (auto simp add: hd-conv-nth take-Suc tl-nth)
ultimately
show (last (drop (Suc sl) l), hd (tl (take (Suc sl) l))) ∈ r
  using ‹rel-chain l r› ‹length l > 2› ‹sl > 0›
  unfolding rel-chain-def
  by simp
qed
qed

moreover

have length ?l' = length l - 1
  by auto

ultimately

obtain l' where *: l' = ?l' hd l' = l ! (sl + 1) last l' = l ! sl rel-chain l' r length
l' = length l - 1
  by auto

have l'-l: ∀ i. i + 1 < length l' →
  (exists j. j + 1 < length l ∧ l' ! i = l ! j ∧ l' ! (i + 1) = l ! (j + 1))
proof (safe)
fix i
assume i + 1 < length l'
show ∃ j. j + 1 < length l ∧ l' ! i = l ! j ∧ l' ! (i + 1) = l ! (j + 1)
proof (cases sl = 0)
case True
then show ?thesis
  using ‹i + 1 < length l'›
  using ‹l' = ?l'› ‹l ≠ []›
  by (force simp add: tl-nth)
next
case False
then have length l' = length l - 1
  using ‹l' = ?l'› ‹sl + 1 < length l›
  by (simp add: min-def)
then have i + 2 < length l
  using ‹i + 1 < length l'›
  by simp

show ?thesis
proof (cases i + 1 < length (drop (sl + 1) l))
case True
then show ?thesis
  using ‹sl ≠ 0› ‹l' = ?l'›
  by (force simp add: nth-append)
next

```

```

case False
show ?thesis
proof (cases i + 1 > length (drop (sl + 1) l))
  case True
    then have i + 1 > length l - (sl + 1)
      by auto
    have
      l' ! i = l ! Suc (i - (length l - Suc sl))
      l' ! (i + 1) = l ! Suc (Suc i - (length l - Suc sl))
      using <i + 2 < length l> <sl + 1 < length l>
      using <i + 1 > length l - (sl + 1)>
      using <sl ≠ 0> <l' = ?l'> <l ≠ []>
      using tl-nth[of take (sl + 1) l i - (length l - Suc sl)]
      using tl-nth[of take (sl + 1) l Suc i - (length l - Suc sl)]
      by (auto simp add: nth-append)

    have Suc (i - (length l - Suc sl)) = i + sl + 1 - length l + 1
      Suc (Suc i - (length l - Suc sl)) = (i + sl + 1 - length l + 1) + 1
      i + sl + 1 - length l + 1 + 1 < length l
      using <sl + 1 < length l>
      using <i + 1 > length l - (sl + 1)>
      using <i + 2 < length l>
      by auto

    have l' ! i = l ! (i + sl + 1 - length l + 1)
      using <l' ! i = l ! Suc (i - (length l - Suc sl))>
      by (subst <Suc (i - (length l - Suc sl)) = i + sl + 1 - length l +
        1>[THEN sym])
    moreover
      have l' ! (i + 1) = l ! ((i + sl + 1 - length l + 1) + 1)
        using <l' ! (i + 1) = l ! Suc (Suc i - (length l - Suc sl))>
        by (subst <Suc (Suc i - (length l - Suc sl)) = (i + sl + 1 - length l +
          1) + 1>[THEN sym])
    ultimately
      show ?thesis
        using <i + sl + 1 - length l + 1 + 1 < length l>
        by force
    next
      case False
        then have i + 1 = length l - sl - 1
          using <¬ i + 1 < length (drop (sl + 1) l)>
          by simp
        then have length l - 1 = sl + i + 1
          by auto
        then have l ! Suc (sl + i) = last l
          using last-conv-nth[of l, THEN sym] <l ≠ []>
          by simp
        then show ?thesis
          using <i + 1 = length l - sl - 1>

```

```

    using ⟨l' = ?l'⟩ ⟨sl ≠ 0⟩ ⟨l ≠ []⟩
    using tl-nth[of take (sl + 1) l 0]
    using ⟨hd l = s⟩ ⟨last l = s⟩
    by (force simp add: nth-append hd-conv-nth)
qed
qed
qed
qed

then show thesis
  using * l'-l
  apply -
  ..
qed

end

```

3 Linearly Ordered Rational Vectors

```

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

class scaleRat =
  fixes scaleRat :: rat ⇒ 'a ⇒ 'a (infixr ⟨*R⟩ 75)
begin

abbreviation
  divideRat :: 'a ⇒ rat ⇒ '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-space scaleRat :: rat ⇒ 'a ⇒ 'a::rational-vector
  by (unfold-locales) (simp-all add: scaleRat-right-distrib scaleRat-left-distrib scaleRat-scaleRat
  scaleRat-one)

class ordered-rational-vector = rational-vector + order

```

```

class linordered-rational-vector = ordered-rational-vector + linorder +
assumes plus-less:  $(a::'a) < b \implies a + c < b + c$  and
scaleRat-less1:  $\llbracket (a::'a) < b; k > 0 \rrbracket \implies (k *R a) < (k *R b)$  and
scaleRat-less2:  $\llbracket (a::'a) < b; k < 0 \rrbracket \implies (k *R a) > (k *R b)$ 
begin

lemma scaleRat-leq1:  $\llbracket a \leq b; k > 0 \rrbracket \implies k *R a \leq k *R b$ 
  unfolding le-less
  using scaleRat-less1[of a b k]
  by auto

lemma scaleRat-leq2:  $\llbracket a \leq b; k < 0 \rrbracket \implies k *R a \geq k *R b$ 
  unfolding le-less
  using scaleRat-less2[of a b k]
  by auto

lemma zero-scaleRat
  [simp]:  $0 *R v = zero$ 
  using scaleRat-left-distrib[of 0 0 v]
  by auto

lemma scaleRat-zero
  [simp]:  $a *R (0::'a) = 0$ 
  using scaleRat-right-distrib[of a 0 0]
  by auto

lemma scaleRat-uminus [simp]:
   $-1 *R x = - (x :: 'a)$ 
proof-
  have  $0 = -1 *R x + x$ 
    using scaleRat-left-distrib[of -1 1 x]
    by (simp add: scaleRat-one)
  have  $-x = 0 - x$ 
    by simp
  then have  $-x = -1 *R x + x - x$ 
    using ‹0 = -1 *R x + x›
    by simp
  then show ?thesis
    by (simp add: add-assoc)
qed

lemma minus-lt:  $(a::'a) < b \longleftrightarrow a - b < 0$ 
  using plus-less[of a b -b]
  using plus-less[of a - b 0 b]
  by (auto simp add: add-assoc)

lemma minus-gt:  $(a::'a) < b \longleftrightarrow 0 < b - a$ 
  using plus-less[of a b -a]
  using plus-less[of 0 b-a a]

```

```
by (auto simp add: add-assoc)
```

```
lemma minus-leq:
  (a::'a) ≤ b ⟷ a - b ≤ 0
proof-
  have *: a ≤ b ⟹ a - b ≤ (0 :: 'a)
    using minus-gt[of a b]
    using scaleRat-less2[of 0 b-a -1]
    by (auto simp add: not-less-iff-gr-or-eq)
  have **: a - b ≤ 0 ⟹ a ≤ b
  proof-
    assume a - b ≤ 0
    show ?thesis
    proof(cases a - b < 0)
      case True
      then show ?thesis
        using plus-less[of a - b 0 b]
        by (simp add: add-assoc )
    next
      case False
      then show ?thesis
        using <a - b ≤ 0>
        by (simp add:antisym-conv1)
    qed
  qed
  show ?thesis
    using * **
    by auto
qed
```

```
lemma minus-geq: (a::'a) ≤ b ⟷ 0 ≤ b - a
proof-
  have *: a ≤ b ⟹ 0 ≤ b - a
    using minus-gt[of a b]
    by (auto simp add: not-less-iff-gr-or-eq)
  have **: 0 ≤ b - a ⟹ a ≤ b
  proof-
    assume 0 ≤ b - a
    show ?thesis
    proof(cases 0 < b - a)
      case True
      then show ?thesis
        using plus-less[of 0 b - a a]
        by (simp add: add-assoc )
    next
      case False
      then show ?thesis
        using <0 ≤ b - a>
        using order.eq-iff[of b - a 0]
```

```

    by auto
qed
qed
show ?thesis
using * **
by auto
qed

lemma divide-lt:
   $\llbracket c *R (a::'a) < b; (c::rat) > 0 \rrbracket \implies a < (1/c) *R b$ 
  using scaleRat-less1[of c *R a b 1/c]
  by (simp add: scaleRat-one scaleRat-scaleRat)

lemma divide-gt:
   $\llbracket c *R (a::'a) > b; (c::rat) > 0 \rrbracket \implies a > (1/c) *R b$ 
  using scaleRat-less1[of b c *R a 1/c]
  by (simp add: scaleRat-one scaleRat-scaleRat)

lemma divide-leq:
   $\llbracket c *R (a::'a) \leq b; (c::rat) > 0 \rrbracket \implies a \leq (1/c) *R b$ 
proof(cases c *R a < b)
assume c > 0
case True
then show ?thesis
  using divide-lt[of c a b]
  using <c > 0>
  by simp
next
assume c *R a \leq b c > 0
case False
then have *: c *R a = b
  using <c *R a \leq b>
  by simp
then show ?thesis
  using <c > 0>
  by (auto simp add: scaleRat-one scaleRat-scaleRat)
qed

lemma divide-geq:
   $\llbracket c *R (a::'a) \geq b; (c::rat) > 0 \rrbracket \implies a \geq (1/c) *R b$ 
proof(cases c *R a > b)
assume c > 0
case True
then show ?thesis
  using divide-gt[of b c a]
  using <c > 0>
  by simp
next
assume c *R a \geq b c > 0

```

```

case False
then have *:  $c *R a = b$ 
  using ⟨ $c *R a \geq bby simp
then show ?thesis
  using ⟨ $c > 0by (auto simp add: scaleRat-one scaleRat-scaleRat)
qed

lemma divide-lt1:
   $\llbracket c *R (a::'a) < b; (c::rat) < 0 \rrbracket \implies a > (1/c) *R b$ 
  using scaleRat-less2[of  $c *R a$   $b$   $1/c$ ]
  by (simp add: scaleRat-scaleRat scaleRat-one)

lemma divide-gt1:
   $\llbracket c *R (a::'a) > b; (c::rat) < 0 \rrbracket \implies a < (1/c) *R b$ 
  using scaleRat-less2[of  $b$   $c *R a$   $1/c$ ]
  by (simp add: scaleRat-scaleRat scaleRat-one)

lemma divide-leq1:
   $\llbracket c *R (a::'a) \leq b; (c::rat) < 0 \rrbracket \implies a \geq (1/c) *R b$ 
proof(cases  $c *R a < b$ )
  assume  $c < 0$ 
  case True
  then show ?thesis
    using divide-lt1[of  $c$   $a$   $b$ ]
    using ⟨ $c < 0by simp
next
  assume  $c *R a \leq b$   $c < 0$ 
  case False
  then have *:  $c *R a = b$ 
    using ⟨ $c *R a \leq bby simp
  then show ?thesis
    using ⟨ $c < 0by (auto simp add: scaleRat-one scaleRat-scaleRat)
qed

lemma divide-geq1:
   $\llbracket c *R (a::'a) \geq b; (c::rat) < 0 \rrbracket \implies a \leq (1/c) *R b$ 
proof(cases  $c *R a > b$ )
  assume  $c < 0$ 
  case True
  then show ?thesis
    using divide-gt1[of  $b$   $c$   $a$ ]
    using ⟨ $c < 0by simp
next$$$$$$ 
```

```

assume c *R a ≥ b c < 0
case False
then have *: c *R a = b
  using ⟨c *R a ≥ b⟩
  by simp
then show ?thesis
  using ⟨c < 0⟩
  by (auto simp add: scaleRat-one scaleRat-scaleRat)
qed

end

class lrv = linordered-rational-vector + one +
assumes zero-neq-one: 0 ≠ 1

subclass (in linordered-rational-vector) ordered-ab-semigroup-add
proof
fix a b c
assume a ≤ b
then show c + a ≤ c + b
  using plus-less[of a b c]
  by (auto simp add: add-ac le-less)
qed

instantiation rat :: rational-vector
begin
definition scaleRat-rat :: rat ⇒ rat ⇒ rat where
[simp]: x *R y = x * y
instance by standard (auto simp add: field-simps)
end

instantiation rat :: ordered-rational-vector
begin
instance ..
end

instantiation rat :: linordered-rational-vector
begin
instance by standard (auto simp add: field-simps)
end

instantiation rat :: lrv
begin
instance by standard (auto simp add: field-simps)
end

lemma uminus-less-lrv[simp]: fixes a b :: 'a :: lrv
  shows - a < - b ↔ b < a
proof -

```

```

have ( $-a < -b$ ) = ( $-1 *R a < -1 *R b$ ) by simp
also have ...  $\longleftrightarrow$  ( $b < a$ )
  using scaleRat-less2[of  $- - - 1$ ] scaleRat-less2[of  $-1 *R a - 1 *R b - 1$ ] by
auto
  finally show ?thesis .
qed

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  $\Rightarrow$  'a::zero where
  [simp]: fun-zero ==  $\lambda v. 0$ 
definition fun-plus :: (var  $\Rightarrow$  'a)  $\Rightarrow$  (var  $\Rightarrow$  'a)  $\Rightarrow$  var  $\Rightarrow$  'a::plus where
  [simp]: fun-plus f1 f2 ==  $\lambda v. f1 v + f2 v$ 
definition fun-scale :: 'a  $\Rightarrow$  (var  $\Rightarrow$  'a)  $\Rightarrow$  (var  $\Rightarrow$  'a::ring) where
  [simp]: fun-scale c f ==  $\lambda v. c*(f v)$ 
definition fun-coeff :: (var  $\Rightarrow$  'a)  $\Rightarrow$  var  $\Rightarrow$  'a where
  [simp]: fun-coeff f var = f var
definition fun-vars :: (var  $\Rightarrow$  'a::zero)  $\Rightarrow$  var set where
  [simp]: fun-vars f = {v. f v  $\neq 0$ }
definition fun-vars-list :: (var  $\Rightarrow$  'a::zero)  $\Rightarrow$  var list where
  [simp]: fun-vars-list f = sorted-list-of-set {v. f v  $\neq 0$ }
definition fun-var :: var  $\Rightarrow$  (var  $\Rightarrow$  'a::{zero,one}) where
  [simp]: fun-var x = ( $\lambda x'. \text{if } x' = x \text{ then } 1 \text{ else } 0$ )
type-synonym 'a valuation = var  $\Rightarrow$  'a
definition fun-validate :: (var  $\Rightarrow$  rat)  $\Rightarrow$  'a valuation  $\Rightarrow$  ('a::rational-vector) where
  [simp]: fun-validate lp val = ( $\sum_{x \in \{v. lp v \neq 0\}} lp x *R val x$ )

Invariant – only finitely many variables

definition inv where
  [simp]: inv c == finite {v. c v  $\neq 0$ }

lemma inv-fun-zero [simp]:
  inv fun-zero by simp

lemma inv-fun-plus [simp]:
  [inv (f1 :: nat  $\Rightarrow$  'a::monoid-add); inv f2]  $\implies$  inv (fun-plus f1 f2)
proof –
  have *: {v. f1 v + f2 v  $\neq (0 :: 'a)$ }  $\subseteq$  {v. f1 v  $\neq (0 :: 'a)$ }  $\cup$  {v. f2 v  $\neq (0 :: 'a)$ }

```

```

    by auto
 $\text{assume } \text{inv } f_1 \text{ inv } f_2$ 
 $\text{then show } ?\text{thesis}$ 
    using *
    by (auto simp add: finite-subset)
qed

lemma inv-fun-scale [simp]:
   $\text{inv } (f :: \text{nat} \Rightarrow 'a::\text{ring}) \implies \text{inv } (\text{fun-scale } r f)$ 
proof-
  have *: {v. r * (f v) ≠ 0} ⊆ {v. f v ≠ 0}
    by auto
  assume inv f
  then show ?thesis
    using *
    by (auto simp add: finite-subset)
qed

```

linear-poly type – rat coeffs

```

typedef linear-poly = {c :: var ⇒ rat. inv c}
  by (rule-tac x=λ v. 0 in exI) auto

```

Linear polynomials are of the form $a_1 \cdot x_1 + \dots + a_n \cdot x_n$. 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 $\sum x \mid p x \neq 0. p x * v x$. 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* by (rule *inv-fun-zero*)

lift-definition *plus-linear-poly* :: *linear-poly* ⇒ *linear-poly* ⇒ *linear-poly* is *fun-plus*
by (rule *inv-fun-plus*)

lift-definition *scaleRat-linear-poly* :: *rat* ⇒ *linear-poly* ⇒ *linear-poly* is *fun-scale*
by (rule *inv-fun-scale*)

definition *uminus-linear-poly* :: *linear-poly* ⇒ *linear-poly* **where**
 $uminus\text{-linear-poly } lp = -1 *R lp$

```

definition minus-linear-poly :: linear-poly  $\Rightarrow$  linear-poly where
  minus-linear-poly lp1 lp2 = lp1 + (- lp2)

instance
proof
  fix a b c :: linear-poly
  show a + b + c = a + (b + c) by (transfer, auto)
  show a + b = b + a by (transfer, auto)
  show 0 + a = a by (transfer, auto)
  show -a + a = 0 unfolding uminus-linear-poly-def by (transfer, auto)
  show a - b = a + (- b) unfolding minus-linear-poly-def ..
next
  fix a :: rat and x y :: linear-poly
  show a *R (x + y) = a *R x + a *R y by (transfer, auto simp: field-simps)
next
  fix a b::rat and x::linear-poly
  show (a + b) *R x = a *R x + b *R x by (transfer, auto simp: field-simps)
  show a *R b *R x = (a * b) *R x by (transfer, auto simp: field-simps)
next
  fix x::linear-poly
  show 1 *R x = x by (transfer, auto)
qed

end

Coefficient

lift-definition coeff :: linear-poly  $\Rightarrow$  var  $\Rightarrow$  rat is fun-coeff .

lemma coeff-plus [simp] : coeff (lp1 + lp2) var = coeff lp1 var + coeff lp2 var
  by transfer auto

lemma coeff-scaleRat [simp]: coeff (k *R lp1) var = k * coeff lp1 var
  by transfer auto

lemma coeff-uminus [simp]: coeff (-lp) var = - coeff lp var
  unfolding uminus-linear-poly-def
  by transfer auto

lemma coeff-minus [simp]: coeff (lp1 - lp2) var = coeff lp1 var - coeff lp2 var
  unfolding minus-linear-poly-def uminus-linear-poly-def
  by transfer auto

Set of variables

lift-definition vars :: linear-poly  $\Rightarrow$  var set is fun-vars .

lemma coeff-zero: coeff p x  $\neq$  0  $\longleftrightarrow$  x  $\in$  vars p
  by transfer auto

```

```

lemma finite-vars: finite (vars p)
  by transfer auto

    List of variables

lift-definition vars-list :: linear-poly  $\Rightarrow$  var list is fun-vars-list .

lemma set-vars-list: set (vars-list lp) = vars lp
  by transfer auto

    Construct single variable polynomial

lift-definition Var :: var  $\Rightarrow$  linear-poly is fun-var by auto

    Value of a polynomial in a given valuation

lift-definition valuate :: linear-poly  $\Rightarrow$  'a valuation  $\Rightarrow$  ('a::rational-vector) is fun-valuate
  .

syntax
  -valuate :: linear-poly  $\Rightarrow$  'a valuation  $\Rightarrow$  'a (‐ {‐} ‐)
syntax-consts
  -valuate == valuate
translations
  p{v} == CONST valuate p v

lemma valuate-zero: (0 {v}) = 0
  by transfer auto

lemma
  valuate-diff: (p {v1}) – (p {v2}) = (p {λ x. v1 x – v2 x})
  by (transfer, simp add: sum-subtractf[THEN sym], auto simp: rational-vector.scale-right-diff-distrib)

lemma valuate-opposite-val:
  shows p {λ x. – v x} = – (p {v})
  using valuate-diff[of p λ x. 0 v]
  by (auto simp add: evaluate-def)

lemma valuate-nonneg:
  fixes v :: 'a::linordered-rational-vector valuation
  assumes  $\forall x \in \text{vars } p.$  (coeff p x > 0  $\longrightarrow$  v x ≥ 0)  $\wedge$  (coeff p x < 0  $\longrightarrow$  v x ≤ 0)
  shows p {v}  $\geq 0$ 
  using assms
proof (transfer, unfold fun-valuate-def, goal-cases)
  case (1 p v)
    from 1 have fin: finite {v. p v ≠ 0} by auto
    then show  $0 \leq (\sum_{x \in \{v. p v \neq 0\}} p x *R v x)$ 
    proof (induct rule: finite-induct)
      case empty show ?case by auto
    next

```

```

case (insert x F)
show ?case unfolding sum.insert[OF insert(1-2)]
proof (rule order.trans[OF - add-mono[OF - insert(3)]]))
  show 0 ≤ p x *R v x using scaleRat-leq1[of 0 v x p x]
    using scaleRat-leq2[of v x 0 p x] 1(2)
    by (cases p x > 0; cases p x < 0; auto)
  qed auto
qed
qed

lemma valuate-nonpos:
  fixes v :: 'a::linordered-rational-vector valuation
  assumes ∀ x ∈ vars p. (coeff p x > 0 → v x ≤ 0) ∧ (coeff p x < 0 → v x ≥ 0)
  shows p {v} ≤ 0
  using assms
  using valuate-opposite-val[of p v]
  using valuate-nonneg[of p λ x. - v x]
  using scaleRat-leq2[of 0:'a - -1]
  using scaleRat-leq2[of - 0:'a - 1]
  by force

lemma valuate-uminus: (‐p) {v} = – (p {v})
  unfolding uminus-linear-poly-def
  by (transfer, auto simp: sum-negf)

lemma valuate-add-lemma:
  fixes v :: 'a ⇒ 'b::rational-vector
  assumes finite {v. f1 v ≠ 0} finite {v. f2 v ≠ 0}
  shows
    (∑ x ∈ {v. f1 v + f2 v ≠ 0}. (f1 x + f2 x) *R v x) =
    (∑ x ∈ {v. f1 v ≠ 0}. f1 x *R v x) + (∑ x ∈ {v. f2 v ≠ 0}. f2 x *R v x)
proof‐
  let ?A = {v. f1 v + f2 v ≠ 0} ∪ {v. f1 v + f2 v = 0 ∧ (f1 v ≠ 0 ∨ f2 v ≠ 0)}
  have ?A = {v. f1 v ≠ 0 ∨ f2 v ≠ 0}
    by auto
  then have
    finite ?A
    using assms
    by (subgoal-tac {v. f1 v ≠ 0 ∨ f2 v ≠ 0} = {v. f1 v ≠ 0} ∪ {v. f2 v ≠ 0})
  auto

  then have (∑ x ∈ {v. f1 v + f2 v ≠ 0}. (f1 x + f2 x) *R v x) =
    (∑ x ∈ {v. f1 v + f2 v ≠ 0} ∪ {v. f1 v + f2 v = 0 ∧ (f1 v ≠ 0 ∨ f2 v ≠ 0)}. (f1 x + f2 x) *R v x)
    by (rule sum.mono-neutral-left) auto
  also have ... = (∑ x ∈ {v. f1 v ≠ 0 ∨ f2 v ≠ 0}. (f1 x + f2 x) *R v x)
    by (rule sum.cong) auto
  also have ... = (∑ x ∈ {v. f1 v ≠ 0 ∨ f2 v ≠ 0}. f1 x *R v x) +

```

```


$$(\sum x \in \{v. f1 v \neq 0 \vee f2 v \neq 0\}. f2 x *R v x)$$

by (simp add: scaleRat-left-distrib sum.distrib)
also have ... =  $(\sum x \in \{v. f1 v \neq 0\}. f1 x *R v x) + (\sum x \in \{v. f2 v \neq 0\}. f2 x *R v x)$ 
proof-
{
  fix f1 f2::'a  $\Rightarrow$  rat
  assume finite {v. f1 v  $\neq$  0} finite {v. f2 v  $\neq$  0}
  then have finite {v. f1 v  $\neq$  0  $\vee$  f2 v  $\neq$  0  $\wedge$  f1 v = 0}
    by (subgoal-tac {v. f1 v  $\neq$  0  $\vee$  f2 v  $\neq$  0} = {v. f1 v  $\neq$  0}  $\cup$  {v. f2 v  $\neq$  0})
  auto
  have  $(\sum x \in \{v. f1 v \neq 0 \vee f2 v \neq 0\}. f1 x *R v x) =$ 
     $(\sum x \in \{v. f1 v \neq 0 \vee (f2 v \neq 0 \wedge f1 v = 0)\}. f1 x *R v x)$ 
    by auto
  also have ... =  $(\sum x \in \{v. f1 v \neq 0\}. f1 x *R v x)$ 
    using <finite {v. f1 v  $\neq$  0  $\vee$  f2 v  $\neq$  0  $\wedge$  f1 v = 0}>
    by (rule sum.mono-neutral-left[THEN sym]) auto
  ultimately have  $(\sum x \in \{v. f1 v \neq 0 \vee f2 v \neq 0\}. f1 x *R v x) =$ 
     $(\sum x \in \{v. f1 v \neq 0\}. f1 x *R v x)$ 
    by simp
}
note * = this
show ?thesis
  using assms
  using*[of f1 f2]
  using*[of f2 f1]
  by (subgoal-tac {v. f2 v  $\neq$  0  $\vee$  f1 v  $\neq$  0} = {v. f1 v  $\neq$  0  $\vee$  f2 v  $\neq$  0}) auto
qed
ultimately
show ?thesis by simp
qed

lemma valuate-add:  $(p1 + p2) \{v\} = (p1 \{v\}) + (p2 \{v\})$ 
by (transfer, simp add: valuate-add-lemma)

lemma valuate-minus:  $(p1 - p2) \{v\} = (p1 \{v\}) - (p2 \{v\})$ 
unfolding minus-linear-poly-def valuate-add
by (simp add: valuate-uminus)

lemma valuate-scaleRat:

$$(c *R lp) \{v\} = c *R (lp \{v\})$$

proof (cases c=0)
  case True
  then show ?thesis
    by (auto simp add: valuate-def zero-linear-poly-def Abs-linear-poly-inverse)
next
  case False
  then have  $\bigwedge v. \text{Rep-linear-poly}(c *R lp) v = c * (\text{Rep-linear-poly} lp v)$ 

```

```

unfolding scaleRat-linear-poly-def
using Abs-linear-poly-inverse[of  $\lambda v. c * Rep\text{-linear-poly} lp v$ ]
using Rep-linear-poly
by auto
then show ?thesis
unfolding valuate-def
using  $c \neq 0$ 
by auto (subst rational-vector.scale-sum-right, auto)
qed

lemma valuate-Var: ( $Var x$ )  $\{v\} = v x$ 
by transfer auto

lemma valuate-sum:  $(\sum_{x \in A. f x} \{v\}) = (\sum_{x \in A. ((f x) \{v\}))$ 
by (induct A rule: infinite-finite-induct, auto simp: valuate-zero valuate-add)

lemma distinct-vars-list:
distinct (vars-list p)
by transfer (use distinct-sorted-list-of-set in auto)

lemma zero-coeff-zero:  $p = 0 \longleftrightarrow (\forall v. coeff p v = 0)$ 
by transfer auto

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 (subst zero-coeff-zero, rule allI)
fix x
show coeff p x = 0
proof (cases x ∈ vars p)
case False
then show ?thesis
using coeff-zero[of p x]
by simp
next
case True
have  $(0::'a::lrv) \neq (1::'a)$ 
using zero-neg-one
by auto

let ?v =  $\lambda x'. if x = x' then 1 else 0::'a$ 
obtain v' where  $\forall x \in vars p. v' x = ?v x p \{v'\} = 0$ 
using assms
by (erule-tac x=?v in alle) auto
then have  $\forall x' \in vars p. v' x' = (if x = x' then 1 else 0) p \{v'\} = 0$ 
by auto

let ?fp = Rep-linear-poly p

```

```

have {x. ?fp x ≠ 0 ∧ v' x ≠ (0 :: 'a)} = {x}
  using ⟨x ∈ vars p⟩ unfolding vars-def
proof (safe, simp-all)
  fix x'
  assume v' x' ≠ 0 Rep-linear-poly p x' ≠ 0
  then show x' = x
    using ⟨∀ x' ∈ vars p. v' x' = (if x = x' then 1 else 0)⟩
    unfolding vars-def
    by (erule-tac x=x' in ballE) (simp-all split: if-splits)
next
  assume v' x = 0 Rep-linear-poly p x ≠ 0
  then show False
    using ⟨∀ x' ∈ vars p. v' x' = (if x = x' then 1 else 0)⟩
    using ⟨0 ≠ 1⟩
    unfolding vars-def
    by simp
qed

have p {v'} = (∑ x ∈ {v. ?fp v ≠ 0}. ?fp x *R v' x)
  unfolding valuate-def
  by auto
also have ... = (∑ x ∈ {v. ?fp v ≠ 0 ∧ v' v ≠ 0}. ?fp x *R v' x)
  apply (rule sum.mono-neutral-left[THEN sym])
  using Rep-linear-poly[of p]
  by auto
also have ... = ?fp x *R v' x
  using ⟨{x. ?fp x ≠ 0 ∧ v' x ≠ (0 :: 'a)} = {x}⟩
  by simp
also have ... = ?fp x *R 1
  using ⟨x ∈ vars p⟩
  using ⟨∀ x' ∈ vars p. v' x' = (if x = x' then 1 else 0)⟩
  by simp
ultimately
have p {v'} = ?fp x *R 1
  by simp
then have coeff p x *R (1::'a) = 0
  using ⟨p {v'} = 0⟩
  unfolding coeff-def
  by simp
then show ?thesis
  using rational-vector.scale-eq-0-iff
  using ⟨0 ≠ 1⟩
  by simp
qed
qed

```

lift-definition *lp-monom* :: *rat* ⇒ *var* ⇒ *linear-poly* **is**
 $\lambda c x y. \text{if } x = y \text{ then } c \text{ else } 0$ **by** *auto*

```

lemma valuate-lp-monom: ((lp-monom c x) {v}) = c * (v x)
proof (transfer, simp, goal-cases)
  case (1 c x v)
    have id: {v. x = v ∧ (x = v → c ≠ 0)} = (if c = 0 then {} else {x}) by auto
    show ?case unfolding id
      by (cases c = 0, auto)
qed

lemma valuate-lp-monom-1[simp]: ((lp-monom 1 x) {v}) = v x
by transfer simp

lemma coeff-lp-monom [simp]:
  shows coeff (lp-monom c v) v' = (if v = v' then c else 0)
  by (transfer, auto)

lemma vars-uminus [simp]: vars (−p) = vars p
  unfolding uminus-linear-poly-def
  by transfer auto

lemma vars-plus [simp]: vars (p1 + p2) ⊆ vars p1 ∪ vars p2
by transfer auto

lemma vars-minus [simp]: vars (p1 − p2) ⊆ vars p1 ∪ vars p2
  unfolding minus-linear-poly-def
  using vars-plus[of p1 − p2] vars-uminus[of p2]
  by simp

lemma vars-lp-monom: vars (lp-monom r x) = (if r = 0 then {} else {x})
by (transfer, auto)

lemma vars-scaleRat1: vars (c *R p) ⊆ vars p
by transfer auto

lemma vars-scaleRat: c ≠ 0 ⇒ vars(c *R p) = vars p
by transfer auto

lemma vars-Var [simp]: vars (Var x) = {x}
by transfer auto

lemma coeff-Var1 [simp]: coeff (Var x) x = 1
by transfer auto

lemma coeff-Var2: x ≠ y ⇒ coeff (Var x) y = 0
by transfer auto

lemma valuate-depend:
  assumes ∀ x ∈ vars p. v x = v' x
  shows (p {v}) = (p {v'})
  using assms

```

by transfer auto

```

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 (cases f x = 0)
    case True
      then have  $\forall y. f y \neq 0 \longrightarrow v1 y = v2 y$ 
      using assms(1) by auto
      then show ?thesis using ⟨f x = 0⟩ by auto
    next
    case False
      let ?A = {v. f v  $\neq 0$ } and ?Ax = {v. v  $\neq x \wedge f v \neq 0$ }
      have ?A = ?Ax  $\cup \{x\}$ 
        using ⟨f x  $\neq 0$ ⟩ by auto
      then have  $(\sum_{x \in ?A} f x *R v1 x) = f x *R v1 x + (\sum_{x \in ?Ax} f x *R v1 x)$ 
       $(\sum_{x \in ?A} f x *R v2 x) = f x *R v2 x + (\sum_{x \in ?Ax} f x *R v2 x)$ 
        using assms(2) by auto
      moreover
      have  $\forall y \in ?Ax. v1 y = v2 y$ 
        using assms by auto
      moreover
      have  $f x *R v1 x + f x *R (v2 x - v1 x) = f x *R v2 x$ 
        by (subst rational-vector.scale-right-diff-distrib) auto
      ultimately
      show ?thesis by simp
  qed

```

```

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})
  using assms
  unfolding valuate-def vars-def coeff-def
  using valuate-update-x-lemma[of Rep-linear-poly lp x v1 v2] Rep-linear-poly
  by auto

```

```

lemma vars-zero: vars 0 = {}
  using zero-coeff-zero coeff-zero by auto

```

```

lemma vars-empty-zero: vars lp = {}  $\longleftrightarrow$  lp = 0
  using zero-coeff-zero coeff-zero by auto

```

definition max-var:: linear-poly \Rightarrow var **where**

```

max-var lp ≡ if lp = 0 then 0 else Max (vars lp)

lemma max-var-max:
  assumes a ∈ vars lp
  shows max-var lp ≥ a
  using assms
  by (auto simp add: finite-vars max-var-def vars-zero)

lemma max-var-code[code]:
  max-var lp = (let vl = vars-list lp
    in if vl = [] then 0 else foldl max (hd vl) (tl vl))
  proof (cases lp = (0::linear-poly))
    case True
    then show ?thesis
      using set-vars-list[of lp]
      by (auto simp add: max-var-def vars-zero)
  next
    case False
    then show ?thesis
      using set-vars-list[of lp, THEN sym]
      using vars-empty-zero[of lp]
      unfolding max-var-def Let-def
      using Max.set-eq-fold[of hd (vars-list lp) tl (vars-list lp)]
      by (cases vars-list lp, auto simp: foldl-conv-fold intro!: fold-cong)
  qed

definition monom-var:: linear-poly ⇒ var where
  monom-var l = max-var l

definition monom-coeff:: linear-poly ⇒ rat where
  monom-coeff l = coeff l (monom-var l)

definition is-monom :: linear-poly ⇒ bool where
  is-monom l ↔ length (vars-list l) = 1

lemma is-monom-vars-not-empty:
  is-monom l ⇒ vars l ≠ {}
  by (auto simp add: is-monom-def vars-list-def) (auto simp add: vars-def)

lemma monom-var-in-vars:
  is-monom l ⇒ monom-var l ∈ vars l
  using vars-zero
  by (auto simp add: monom-var-def max-var-def is-monom-vars-not-empty finite-vars is-monom-def)

lemma zero-is-no-monom[simp]: ¬ is-monom 0
  using is-monom-vars-not-empty vars-zero by blast

lemma is-monom-monom-coeff-not-zero:

```

```

is-monom l ==> monom-coeff l ≠ 0
by (simp add: coeff-zero monom-var-in-vars monom-coeff-def)

```

lemma *list-two-elements*:

```

[y ∈ set l; x ∈ set l; length l = Suc 0; y ≠ x] ==> False
by (induct l) auto

```

lemma *is-monom-vars-monom-var*:

assumes *is-monom l*

shows *vars l = {monom-var l}*

proof-

```

have ∀x. [is-monom l; x ∈ vars l] ==> monom-var l = x

```

proof-

fix x

assume *is-monom l x ∈ vars l*

then have *x ∈ set (vars-list l)*

using *finite-vars*

by (auto simp add: *vars-list-def vars-def*)

show *monom-var l = x*

proof(rule *ccontr*)

assume *monom-var l ≠ x*

then have $\exists y. \text{monom-var } l = y \wedge y \neq x$

by *simp*

then obtain y where *monom-var l = y y ≠ x*

by *auto*

then have *Rep-linear-poly l y ≠ 0*

using *monom-var-in-vars <is-monom l>*

by (auto simp add: *vars-def*)

then have *y ∈ set (vars-list l)*

using *finite-vars*

by (auto simp add: *vars-def vars-list-def*)

then show *False*

using *<x ∈ set (vars-list l)> <is-monom l> <y ≠ x>*

using *list-two-elements*

by (simp add: *is-monom-def*)

qed

qed

then show *vars l = {monom-var l}*

using *assms*

by (auto simp add: *monom-var-in-vars*)

qed

lemma *monom-valuete*:

assumes *is-monom m*

shows $m\{v\} = (\text{monom-coeff } m) *R v (\text{monom-var } m)$

using *assms*

using *is-monom-vars-monom-var*

by (simp add: *vars-def coeff-def monom-coeff-def valuate-def*)

```

lemma coeff-zero-simp [simp]:
  coeff 0 v = 0
  using zero-coeff-zero by blast

lemma poly-eq-iff: p = q  $\longleftrightarrow$  ( $\forall$  v. coeff p v = coeff q v)
  by transfer auto

lemma poly-eqI:
  assumes  $\bigwedge$ v. coeff p v = coeff q v
  shows p = q
  using assms poly-eq-iff by simp

lemma coeff-sum-list:
  assumes distinct xs
  shows coeff ( $\sum$ x $\in$ xs. f x *R lp-monom 1 x) v = (if v  $\in$  set xs then f v else 0)
  using assms by (induction xs) auto

lemma linear-poly-sum:
  p {v} = ( $\sum$ x $\in$ vars p. coeff p x *R v x)
  by transfer simp

lemma all-evaluate-zero: assumes  $\bigwedge$ (v::'a::lrv valuation). p {v} = 0
  shows p = 0
  using all-val assms by blast

lemma linear-poly-eqI: assumes  $\bigwedge$ (v::'a::lrv valuation). (p {v}) = (q {v})
  shows p = q
  using assms
proof -
  have (p - q) {v} = 0 for v::'a::lrv valuation
  using assms by (subst evaluate-minus) auto
  then have p - q = 0
  by (intro all-evaluate-zero) auto
  then show ?thesis
  by simp
qed

lemma monom-poly-assemble:
  assumes is-monom p
  shows monom-coeff p *R lp-monom 1 (monom-var p) = p
  by (simp add: assms linear-poly-eqI monom-evaluate evaluate-scaleRat)

lemma coeff-sum: coeff (sum (f :: -  $\Rightarrow$  linear-poly) is) x = sum ( $\lambda$  i. coeff (f i) x) is
  by (induct is rule: infinite-finite-induct, auto)

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 -
fix fmap
show inv (get-var-coeff fmap) unfolding inv-def
by (rule finite-subset[OF - dom-fmlookup-finite[of fmap]],
auto intro: fmdom'I simp: get-var-coeff-def split: option.splits)
qed

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

context includes fmap.lifting and lifting-syntax
begin

lemma [transfer-rule]: (((=) ===> (=)) ===> pcr-linear-poly ===> (=)) (=)
pcr-linear-poly
by (standard, auto simp: pcr-linear-poly-def cr-linear-poly-def rel-fun-def OO-def)

lemma [transfer-rule]: (pcr-fmap (=) (=) ===> pcr-linear-poly) (λ f x. case f x
of None ⇒ 0 | Some x ⇒ x) LinearPoly
by (standard, transfer, auto simp: get-var-coeff-def fmap.pcr-cr-eq cr-fmap-def)

lift-definition linear-poly-map :: linear-poly ⇒ (var, rat) fmap is
λ lp x. if lp x = 0 then None else Some (lp x) by (auto simp: dom-def)

lemma certificate[code abstype]:
LinearPoly (linear-poly-map lp) = lp
by (transfer, auto)

Zero

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

lemma [code abstract]:
linear-poly-map 0 = zero unfolding zero-def

```

by (*transfer, auto*)

Addition

definition *add-monom* :: *rat* \Rightarrow *var* \Rightarrow (*var, rat*) *fmap* \Rightarrow (*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* \Rightarrow (*var, rat*) *fmap* \Rightarrow (*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 $\neq 0 \Rightarrow
fmlookup (add-monom c v lp) v = *Some (get-var-coeff lp v + c)*
get-var-coeff lp v + c = *0* $\Rightarrow
fmlookup (add-monom c v lp) v = *None*
x $\neq v \Rightarrow *fmlookup (add-monom c v lp) x* = *fmlookup lp x*
unfolding *add-monom-def get-var-coeff-def set-var-coeff-def*
by *auto*$$$

lemma *fmlookup-fold-not-mem*: *x* \notin *set k1* \Rightarrow
fmlookup (foldl (\lambda lp v. add-monom (get-var-coeff P1 v) v lp) P2 k1) x
= *fmlookup P2 x*
by (*induct k1 arbitrary: P2, auto simp: lookup-add-monom*)

lemma [*code abstract*]:

linear-poly-map (p1 + p2) = *add (linear-poly-map p1) (linear-poly-map p2)*

proof (*rule fmap-ext*)

fix *x* :: *nat*

let *?p1* = *fmlookup (linear-poly-map p1) x*

let *?p2* = *fmlookup (linear-poly-map p2) x*

define *P1* **where** *P1* = *linear-poly-map p1*

define *P2* **where** *P2* = *linear-poly-map p2*

define *k1* **where** *k1* = *ordered-keys P1*

have *k1: distinct k1* \wedge *fset (fmdom P1)* = *set k1* **unfolding** *k1-def P1-def*
ordered-keys-def

by *auto*

have *id: fmlookup (linear-poly-map (p1 + p2)) x* = (*case ?p1 of None* \Rightarrow *?p2 |*

Some y1 \Rightarrow

(*case ?p2 of None* \Rightarrow *Some y1 | Some y2* \Rightarrow *if y1 + y2 = 0* *then None* *else Some (y1 + y2))*)

by (*transfer, auto*)

show *fmlookup (linear-poly-map (p1 + p2)) x* = *fmlookup (add (linear-poly-map p1) (linear-poly-map p2)) x*

proof (*cases fmlookup P1 x*)

case *None*

from *fmdom-notI[OF None]* **have** *x* \notin *fset (fmdom P1)* **by** *metis*

with *k1* **have** *x: x* \notin *set k1* **by** *auto*

show *?thesis unfolding id P1-def[symmetric] P2-def[symmetric] None*

unfolding *add-def k1-def[symmetric] fmlookup-fold-not-mem[OF x]* **by** *auto*

```

next
  case (Some  $y_1$ )
    from fmdomI[OF this] have  $x \in fset(fmdom P1)$  by metis
    with  $k_1$  have  $x \in set k_1$  by auto
      from split-list[OF this] obtain bef aft where  $k_1\text{-}id: k_1 = bef @ x \# aft$  by
        auto
        with  $k_1$  have  $x: x \notin set bef x \notin set aft$  by auto
        have  $xy_1: get\text{-}var\text{-}coeff P1 x = y_1$  using Some unfolding get-var-coeff-def by
          auto
        let  $?P = foldl (\lambda lp v. add\text{-}monom (get\text{-}var\text{-}coeff P1 v) v lp) P2 bef$ 
        show ?thesis unfolding id P1\text{-}def[symmetric] P2\text{-}def[symmetric] Some option.simps
          unfolding add\text{-}def k1\text{-}def[symmetric] k1\text{-}id foldl\text{-}append foldl\text{-}Cons
          unfolding fmlookup\text{-}fold\text{-}not\text{-}mem[OF x(2)] xy1
        proof –
          show (case fmlookup P2 x of None  $\Rightarrow$  Some  $y_1 \mid$  Some  $y_2 \Rightarrow if y_1 + y_2 = 0$ 
            then None else Some ( $y_1 + y_2$ ))
             $= fmlookup (add\text{-}monom y_1 x ?P) x$ 
          proof (cases get\text{-}var\text{-}coeff ?P x + y_1 = 0)
            case True
              from Some[unfolded P1\text{-}def] have  $y_1: y_1 \neq 0$ 
              by (transfer, auto split: if-splits)
              then show ?thesis unfolding lookup\text{-}add\text{-}monom(2)[OF True] using True
                unfolding get\text{-}var\text{-}coeff\text{-}def[of - x] fmlookup\text{-}fold\text{-}not\text{-}mem[OF x(1)]
                by (auto split: option.splits)
            next
              case False
              show ?thesis unfolding lookup\text{-}add\text{-}monom(1)[OF False] using False
                unfolding get\text{-}var\text{-}coeff\text{-}def[of - x] fmlookup\text{-}fold\text{-}not\text{-}mem[OF x(1)]
                by (auto split: option.splits)
              qed
            qed
            qed
          qed

```

Scaling

```

definition scale :: rat  $\Rightarrow$  (var, rat) fmap  $\Rightarrow$  (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 (cases r = 0)
  case True
  then have  $*: (r = 0) = True$  by simp
  show ?thesis unfolding scale\text{-}def * if\text{-}True using True
    by (transfer, auto)
next
  case False
  then have  $*: (r = 0) = False$  by simp

```

```

show ?thesis unfolding scale-def * if-False using False
  by (transfer, auto)
qed

```

```

lemma coeff-code[code]:
  coeff lp = get-var-coeff (linear-poly-map lp)
  by (rule ext, unfold get-var-coeff-def, transfer, auto)

```

```

lemma Var-code[code abstract]:
  linear-poly-map (Var x) = set-var-coeff x 1 fmempty
  unfolding set-var-coeff-def
  by (transfer, auto split: if-splits simp: fun-eq-iff map-upd-def)

```

```

lemma vars-code[code]: vars lp = fset (fmdom (linear-poly-map lp))
  by (transfer, auto simp: Transfer.Rel-def rel-fun-def pcr-fset-def cr-fset-def)

```

```

lemma vars-list-code[code]: vars-list lp = ordered-keys (linear-poly-map lp)
  unfolding ordered-keys-def vars-code[symmetric]
  by (transfer, auto)

```

```

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)))
  unfolding Let-def
  proof (subst sum-list-distinct-conv-sum-set)
    show distinct (vars-list lp)
      by (transfer, auto)
  next
    show lp { val } =
      (sum x∈set (vars-list lp). the (fmlookup (linear-poly-map lp) x) *R val x)
    unfolding set-vars-list
    by (transfer, auto)
  qed

```

end

```

lemma lp-monom-code[code]: linear-poly-map (lp-monom c x) = (if c = 0 then
  fmempty else fmupd x c fmempty)
  proof (rule fmap-ext, goal-cases)
    case (1 y)
    include fmap.lifting
    show ?case by (cases c = 0, (transfer, auto)+)
  qed

```

```

instantiation linear-poly :: equal
begin

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

instance
proof (standard, unfold equal-linear-poly-def, standard)
  fix x y
  assume linear-poly-map x = linear-poly-map y
  from arg-cong[OF this, of LinearPoly, unfolded certificate]
  show x = y .
qed auto
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
  by (cases qd) auto

lemma [simp]: ⟦ QDelta.qdsnd x = QDelta.qdsnd y; QDelta.qdfst y = QDelta.qdfst x ⟧ ⇒ x = y
  by (cases x) auto

instantiation QDelta :: rational-vector
begin

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

```

```

definition plus-QDelta :: QDelta  $\Rightarrow$  QDelta  $\Rightarrow$  QDelta
  where
     $qd1 + qd2 = QDelta (qdfst\ qd1 + qdfst\ qd2) (qdsnd\ qd1 + qdsnd\ qd2)$ 

definition minus-QDelta :: QDelta  $\Rightarrow$  QDelta  $\Rightarrow$  QDelta
  where
     $qd1 - qd2 = QDelta (qdfst\ qd1 - qdfst\ qd2) (qdsnd\ qd1 - qdsnd\ qd2)$ 

definition uminus-QDelta :: QDelta  $\Rightarrow$  QDelta
  where
     $- qd = QDelta (- (qdfst\ qd)) (- (qdsnd\ qd))$ 

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

instance
proof
qed (auto simp add: plus-QDelta-def zero-QDelta-def uminus-QDelta-def minus-QDelta-def
scaleRat-QDelta-def field-simps)
end

instantiation QDelta :: linorder
begin
definition less-eq-QDelta :: QDelta  $\Rightarrow$  QDelta  $\Rightarrow$  bool
  where
     $qd1 \leq qd2 \longleftrightarrow (qdfst\ qd1 < qdfst\ qd2) \vee (qdfst\ qd1 = qdfst\ qd2 \wedge qdsnd\ qd1 \leq qdsnd\ qd2)$ 

definition less-QDelta :: QDelta  $\Rightarrow$  QDelta  $\Rightarrow$  bool
  where
     $qd1 < qd2 \longleftrightarrow (qdfst\ qd1 < qdfst\ qd2) \vee (qdfst\ qd1 = qdfst\ qd2 \wedge qdsnd\ qd1 < qdsnd\ qd2)$ 

instance proof qed (auto simp add: less-QDelta-def less-eq-QDelta-def)
end

instantiation QDelta:: linordered-rational-vector
begin
instance proof qed (auto simp add: plus-QDelta-def less-QDelta-def scaleRat-QDelta-def
mult-strict-left-mono mult-strict-left-mono-neg)
end

instantiation QDelta :: lrv
begin
definition one-QDelta where
   $one\text{-}QDelta = QDelta 1\ 0$ 
instance proof qed (auto simp add: one-QDelta-def zero-QDelta-def)
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 ∧ k₁ > k₂) then
        (c₂ - c₁) / (k₁ - k₂)
      else
        1
    )

```

definition val :: QDelta ⇒ rat ⇒ rat
where val qd δ = (qdfst qd) + δ * (qdsnd qd)

lemma val-plus:
 val (qd₁ + qd₂) δ = val qd₁ δ + val qd₂ δ
by (simp add: field-simps val-def plus-QDelta-def)

lemma val-scaleRat:
 val (c *R qd) δ = c * val qd δ
by (simp add: field-simps val-def scaleRat-QDelta-def)

lemma qdfst-setsum:
 finite A ⟹ qdfst (∑ x∈A. f x) = (∑ x∈A. qdfst (f x))
by (induct A rule: finite-induct) (auto simp add: zero-QDelta-def plus-QDelta-def)

lemma qdsnd-setsum:
 finite A ⟹ qdsnd (∑ x∈A. f x) = (∑ x∈A. qdsnd (f x))
by (induct A rule: finite-induct) (auto simp add: zero-QDelta-def plus-QDelta-def)

lemma valuate-validate-rat:
 lp {((λv. (QDelta (vl v) 0))} = QDelta (lp{vl}) 0
using Rep-linear-poly
by (simp add: validate-def scaleRat-QDelta-def qdsnd-setsum qdfst-setsum)

lemma validate-rat-validate:
 lp {((λv. val (vl v) δ)} = val (lp{vl}) δ
unfolding validate-def val-def
using rational-vector.scale-sum-right[of δ λx. Rep-linear-poly lp x * qdsnd (vl x)]
{v :: nat. Rep-linear-poly lp v ≠ (0 :: rat)}
using Rep-linear-poly
by (auto simp add: field-simps sum.distrib qdfst-setsum qdsnd-setsum) (auto simp add: scaleRat-QDelta-def)

lemma delta0:
assumes qd₁ ≤ qd₂
shows ∀ ε. ε > 0 ∧ ε ≤ (δ₀ qd₁ qd₂) ⟹ val qd₁ ε ≤ val qd₂ ε
proof-

```

have  $\wedge e c1 c2 k1 k2 :: rat. [e \geq 0; c1 < c2; k1 \leq k2] \implies c1 + e*k1 \leq c2 + e*k2$ 
proof-
  fix  $e c1 c2 k1 k2 :: rat$ 
  show  $[e \geq 0; c1 < c2; k1 \leq k2] \implies c1 + e*k1 \leq c2 + e*k2$ 
    using mult-left-mono[of  $k1 k2 e$ ]
    using add-less-le-mono[of  $c1 c2 e*k1 e*k2$ ]
    by simp
  qed
  then show ?thesis
    using assms
    by (auto simp add: delta-def val-def less-eq-QDelta-def Let-def field-simps mult-left-mono)
qed

primrec
   $\delta\text{-min} :: (QDelta \times QDelta) list \Rightarrow rat$  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:
   $\delta\text{-min} l > 0$ 
  by (induct l) (auto simp add: Let-def field-simps delta-def)

lemma delta-le-one:
   $\delta\text{-min} l \leq 1$ 
  by (induct l, auto)

lemma delta-min-append:
   $\delta\text{-min} (as @ bs) = min (\delta\text{-min} as) (\delta\text{-min} bs)$ 
  by (induct as, insert delta-le-one[of bs], auto)

lemma delta-min-mono: set as  $\subseteq$  set bs  $\implies \delta\text{-min} bs \leq \delta\text{-min} as$ 
proof (induct as)
  case Nil
  then show ?case using delta-le-one by simp
  next
    case (Cons a as)
    from Cons(2) have a  $\in$  set bs by auto
    from split-list[OF this]
    obtain bs1 bs2 where bs:  $bs = bs1 @ [a] @ bs2$  by auto
    have bs:  $\delta\text{-min} bs = \delta\text{-min} ([a] @ bs)$  unfolding bs delta-min-append by auto
    show ?case unfolding bs using Cons(1-2) by auto
  qed

lemma delta-min:
  assumes  $\forall qd1 qd2. (qd1, qd2) \in set qd \implies qd1 \leq qd2$ 
  shows  $\forall \varepsilon. \varepsilon > 0 \wedge \varepsilon \leq \delta\text{-min} qd \implies (\forall qd1 qd2. (qd1, qd2) \in set qd \implies val qd1 \varepsilon \leq val qd2 \varepsilon)$ 

```

```

using assms
using delta0
by (induct qd, auto)

lemma QDelta-0-0: QDelta 0 0 = 0 by code-simp
lemma qdsnd-0: qdsnd 0 = 0 by code-simp
lemma qdfst-0: qdfst 0 = 0 by code-simp

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-Numerical
begin

```

Linear constraints are of the form $p \bowtie c$, where p is a homogenous 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 $xs \equiv snd`xs$

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

abbreviation (*input*) *flat-list* :: $('i \times 'a) list \Rightarrow 'a list$ **where**
 $flat-list xs \equiv map snd xs$

primrec

satisfies-constraint :: $'a :: lrv valuation \Rightarrow constraint \Rightarrow bool$ (**infixl** $\triangleleft_{c} 100$)

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* $\Rightarrow constraint set \Rightarrow bool$ (**infixl**
 $\triangleleft_{cs} 100$) **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 $\neg (\exists v. v \models_{cs} ds)$
using *assms* **by** *auto*

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

definition *distinct-indices* :: $('i \times 'c) list \Rightarrow bool$ **where**
 $distinct_indices as = (distinct (map fst as))$

lemma *distinct-indicesD*: *distinct-indices as* $\implies (i, x) \in set as \implies (i, y) \in set as$
 $\implies x = y$
unfolding *distinct-indices-def* **by** (*rule eq-key-imp-eq-value*)

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* :: $'i set \Rightarrow 'i i\text{-constraint list} \Rightarrow bool$ **where**
 $minimal_unsat_core I ics = ((I \subseteq fst `set ics) \wedge (\neg (\exists v. (I, v) \models_{ics} set ics)))$
 $\wedge (distinct_indices ics \longrightarrow (\forall J. J \subset I \longrightarrow (\exists v. (J, v) \models_{ics} set ics))))$

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 |=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)
  by (transfer, auto)

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

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

lemma map2fun-def':
  ⟨v⟩ x ≡ case Mapping.lookup v x of None ⇒ 0 | Some y ⇒ y
  by (auto simp add: map2fun-def)
```

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 | 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⟩ |=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
by (unfold-locales, insert simplex-sat0 simplex-unsat0,
  auto simp: solve-def split: sum.splits)

```

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}_δ 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} ≤ r
  | v  $\models_{ns}$  GEQ-ns l r ⟷ l{v} ≥ r

abbreviation satisfies-ns-constraints :: 'a::lrv valuation ⇒ 'a ns-constraint set ⇒
bool (infixl ⟨ $\models_{nss}$ ⟩ 100) where
  v  $\models_{nss}$  cs ≡ ∀ c ∈ 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 ⟷ v  $\models_{nss}$  restrict-to I cs

lemma i-satisfies-ns-constraints-mono:
  (I,v)  $\models_{inss}$  cs ==> J ⊆ I ==> (J,v)  $\models_{inss}$  cs
  by auto

```

```

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) ⊨_inss cs)) ∧
    (distinct-indices-ns cs → (∀ J ⊂ I. ∃ v. (J,v) ⊨_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'⟩) ⊨_inss set (to-ns cs) ⇒ (I,⟨from-ns v' (flat-list (to-ns cs)))⟩ ⊨_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'⟩ ⊨_nss flat (set (to-ns cs)) ⇒ ⟨from-ns v' (flat-list (to-ns cs)))⟩ ⊨_cs flat (set cs)
  using i-to-ns-sat[of UNIV v' cs] by auto
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⟩ ⊨_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  $\Rightarrow$  ('i,'a::lrv) i-ns-constraint list and
  from-ns :: (var, 'a) mapping  $\Rightarrow$  'a ns-constraint list  $\Rightarrow$  (var, rat) mapping and
  solve-exec-ns :: ('i,'a) i-ns-constraint list  $\Rightarrow$  'i list + (var, 'a) mapping
begin

definition solve-exec where
  solve-exec cs  $\equiv$  let cs' = to-ns cs in case solve-exec-ns cs'
    of Sat v  $\Rightarrow$  Sat (from-ns v (flat-list cs'))
    | Unsat is  $\Rightarrow$  Unsat is

end

sublocale SolveExec' < SolveExec solve-exec
  by (unfold-locales, insert simplex-ns-sat simplex-ns-unsat to-ns-sat to-ns-unsat,
    (force simp: solve-exec-def Let-def split: sum.splits)+)

```

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  $\times$  linear-poly
primrec lhs :: eq  $\Rightarrow$  var where lhs (l, r) = l
primrec rhs :: eq  $\Rightarrow$  linear-poly where rhs (l, r) = r
abbreviation rvars-eq :: eq  $\Rightarrow$  var set where
  rvars-eq eq  $\equiv$  vars (rhs eq)

definition satisfies-eq :: 'a::rational-vector valuation  $\Rightarrow$  eq  $\Rightarrow$  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}
  by (simp add: satisfies-eq-def)

```

```
type-synonym tableau = eq list
```

```
definition satisfies-tableau :: 'a::rational-vector valuation ⇒ tableau ⇒ bool (infixl
  ⪯_t 100) where
  v ⪯_t t ≡ ∀ e ∈ set t. v ⪯_e e
```

```
definition lvars :: tableau ⇒ var set where
  lvars t = set (map lhs t)
definition rvars :: tableau ⇒ var set where
  rvars t = ⋃ (set (map rvars-eq t))
abbreviation tvars where tvars t ≡ lvars t ∪ rvars t
```

The condition that the rhs 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 $eq = var \times linear-poly$. Semantics of equations is given by $v \models_e (x, p) \equiv v x = p \setminus v$. 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 Δt) 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 (safe)
  fix x
  assume x ∈ lvars t
  then show ∃ p. (x, p) ∈ set t
    unfolding lvars-def
    by auto
next
  fix x p1 p2
  assume *: (x, p1) ∈ set t (x, p2) ∈ set t
  then show p1 = p2
    using △ t
    unfolding normalized-tableau-def
```

```

by (force simp add: distinct-map inj-on-def)
qed

lemma recalc-tableau-lvars:
assumes  $\Delta t$ 
shows  $\forall v. \exists v'. (\forall x \in rvars t. v x = v' x) \wedge v' \models_t t$ 
proof
fix v
let  $?v' = \lambda x. \text{if } x \in lvars t \text{ then } (\text{THE } p. (x, p) \in \text{set } t) \setminus v \text{ else } v x$ 
show  $\exists v'. (\forall x \in rvars t. v x = v' x) \wedge v' \models_t t$ 
proof (rule-tac x=?v' in exI, rule conjI)
show  $\forall x \in rvars t. v x = ?v' x$ 
using  $\langle \Delta t \rangle$ 
unfolding normalized-tableau-def
by auto
show  $?v' \models_t t$ 
unfolding satisfies-tableau-def satisfies-eq-def
proof
fix e
assume  $e \in \text{set } t$ 
obtain l r where  $e: e = (l, r)$  by force
show  $?v' (\text{lhs } e) = \text{rhs } e \setminus ?v'$ 
proof-
have  $(\text{lhs } e, \text{rhs } e) \in \text{set } t$ 
using  $\langle e \in \text{set } t \rangle e$  by auto
have  $\exists !p. (\text{lhs } e, p) \in \text{set } t$ 
using  $\langle \Delta t \rangle \text{ normalized-tableau-unique-eq-for-lvar}[of t]$ 
using  $\langle e \in \text{set } t \rangle$ 
unfolding lvars-def
by auto

let  $?p = \text{THE } p. (\text{lhs } e, p) \in \text{set } t$ 
have  $(\text{lhs } e, ?p) \in \text{set } t$ 
apply (rule theI')
using  $\langle \exists !p. (\text{lhs } e, p) \in \text{set } t \rangle$ 
by auto
then have  $?p = \text{rhs } e$ 
using  $\langle (\text{lhs } e, \text{rhs } e) \in \text{set } t \rangle$ 
using  $\langle \exists !p. (\text{lhs } e, p) \in \text{set } t \rangle$ 
by auto
moreover
have  $?v' (\text{lhs } e) = ?p \setminus v$ 
using  $\langle e \in \text{set } t \rangle$ 
unfolding lvars-def
by simp
moreover
have  $\text{rhs } e \setminus ?v' = \text{rhs } e \setminus v$ 
apply (rule valuate-depend)
using  $\langle \Delta t \rangle \langle e \in \text{set } t \rangle$ 

```

```

unfolding normalized-tableau-def
  by (auto simp add: lvars-def rvars-def)
ultimately
  show ?thesis
    by auto
qed
qed
qed
qed

lemma tableau-perm:
  assumes lvars t1 = lvars t2 rvars t1 = rvars t2
     $\triangle t1 \triangle t2 \wedge v::'a::lrv\ valuation. v \models_t t1 \longleftrightarrow v \models_t t2$ 
  shows mset t1 = mset t2
proof-
{
  fix t1 t2
  assume lvars t1 = lvars t2 rvars t1 = rvars t2
     $\triangle t1 \wedge v::'a::lrv\ valuation. v \models_t t1 \longleftrightarrow v \models_t t2$ 
  have set t1  $\subseteq$  set t2
  proof (safe)
    fix a b
    assume (a, b)  $\in$  set t1
    then have a  $\in$  lvars t1
      unfolding lvars-def
      by force
    then have a  $\in$  lvars t2
      using ⟨lvars t1 = lvars t2⟩
      by simp
    then obtain b' where (a, b')  $\in$  set t2
      unfolding lvars-def
      by force
    have  $\forall v::'a\ valuation. \exists v'. (\forall x \in vars (b - b'). v' x = v x) \wedge (b - b') \setminus v' = 0$ 
proof
  fix v::'a valuation
  obtain v' where v'  $\models_t t1 \forall x \in rvars t1. v x = v' x$ 
    using recalc-tableau-lvars[of t1] ⟨ $\triangle t1$ ⟩
    by auto
  have v'  $\models_t t2$ 
    using ⟨v'  $\models_t t1$ ⟩  $\wedge$  v'  $\models_t t1 \longleftrightarrow v \models_t t2$ 
    by simp
  have b  $\setminus v' = b' \setminus v'$ 
    using ⟨(a, b)  $\in$  set t1⟩ ⟨v'  $\models_t t1$ ⟩
    using ⟨(a, b')  $\in$  set t2⟩ ⟨v'  $\models_t t2$ ⟩
    unfolding satisfies-tableau-def satisfies-eq-def
    by force
  then have (b - b')  $\setminus v' = 0$ 
    using valuate-minus[of b b' v']

```

```

    by auto
  moreover
  have vars b ⊆ rvars t1 vars b' ⊆ rvars t1
    using ⟨(a, b) ∈ set t1⟩ ⟨(a, b') ∈ set t2⟩ ⟨rvars t1 = rvars t2⟩
    unfolding rvars-def
    by force+
  then have vars (b - b') ⊆ rvars t1
    using vars-minus[of b b']
    by blast
  then have ∀ x∈vars (b - b'). v' x = v x
    using ∀ x ∈ rvars t1. v x = v' x
    by auto
  ultimately
  show ∃ v'. (∀ x∈vars (b - b'). v' x = v x) ∧ (b - b') { v' } = 0
    by auto
qed
then have b = b'
  using all-val[of b - b']
  by simp
then show (a, b) ∈ set t2
  using ⟨(a, b) ∈ set t2⟩
  by simp
qed
}
note * = this
have set t1 = set t2
  using *[of t1 t2] *[of t2 t1]
  using assms
  by auto
moreover
have distinct t1 distinct t2
  using ⟨△ t1⟩ ⟨△ t2⟩
  unfolding normalized-tableau-def
  by (auto simp add: distinct-map)
ultimately
show ?thesis
  by (auto simp add: set-eq-iff-mset-eq-distinct)
qed

```

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  $\Rightarrow$  'a atom  $\Rightarrow$  bool (infixl  $\trianglelefteq_a$  100) where
   $v \models_a \text{Leq } x \ c \longleftrightarrow v \ x \leq c$  |  $v \models_a \text{Geq } x \ c \longleftrightarrow v \ x \geq c$ 

definition satisfies-atom-set :: 'a::linorder valuation  $\Rightarrow$  'a atom set  $\Rightarrow$  bool (infixl  $\trianglelefteq_{as}$  100) where
   $v \models_{as} as \equiv \forall a \in as. v \models_a a$ 

definition satisfies-atom' :: 'a::linorder valuation  $\Rightarrow$  'a atom  $\Rightarrow$  bool (infixl  $\trianglelefteq_{ae}$  100) where
   $v \models_{ae} a \longleftrightarrow v (\text{atom-var } a) = \text{atom-const } a$ 

lemma satisfies-atom'-stronger:  $v \models_{ae} a \implies v \models_a a$  by (cases a, auto simp: satisfies-atom'-def)

abbreviation satisfies-atom-set' :: 'a::linorder valuation  $\Rightarrow$  'a atom set  $\Rightarrow$  bool (infixl  $\trianglelefteq_{aes}$  100) where
   $v \models_{aes} as \equiv \forall a \in as. v \models_{ae} a$ 

lemma satisfies-atom-set'-stronger:  $v \models_{aes} as \implies v \models_{as} as$ 
  using satisfies-atom'-stronger unfolding satisfies-atom-set-def by auto

  There is also the indexed variant of an atom

  type-synonym ('i,'a) i-atom = 'i  $\times$  'a atom

  fun i-satisfies-atom-set :: 'i set  $\times$  'a::linorder valuation  $\Rightarrow$  ('i,'a) i-atom set  $\Rightarrow$  bool (infixl  $\trianglelefteq_{ias}$  100) where
     $(I,v) \models_{ias} as \longleftrightarrow v \models_{as} \text{restrict-to } I \ as$ 

  fun i-satisfies-atom-set' :: 'i set  $\times$  'a::linorder valuation  $\Rightarrow$  ('i,'a) i-atom set  $\Rightarrow$  bool (infixl  $\trianglelefteq_{iaes}$  100) where
     $(I,v) \models_{iaes} as \longleftrightarrow v \models_{aes} \text{restrict-to } I \ as$ 

  lemma i-satisfies-atom-set'-stronger:  $Iv \models_{iaes} as \implies Iv \models_{ias} as$ 
    using satisfies-atom-set'-stronger[of - snd Iv] by (cases Iv, auto)

  lemma satisfies-atom-restrict-to-Cons:  $v \models_{as} \text{restrict-to } I \ (\text{set } as) \implies (i \in I \implies$ 
     $v \models_a a)$ 
     $\implies v \models_{as} \text{restrict-to } I \ (\text{set } ((i,a) \ # \ as))$ 
    unfolding satisfies-atom-set-def by auto

  lemma satisfies-tableau-Cons:  $v \models_t t \implies v \models_e e \implies v \models_t (e \ # \ t)$ 
    unfolding satisfies-tableau-def by auto

  definition distinct-indices-atoms :: ('i,'a) i-atom set  $\Rightarrow$  bool where
     $\text{distinct-indices-atoms } as = (\forall i \ a \ b. (i,a) \in as \longrightarrow (i,b) \in as \longrightarrow \text{atom-var } a = \text{atom-var } b \wedge \text{atom-const } a = \text{atom-const } b)$ 

```

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⟩) ⊨ias set as ==> ⟨v⟩ ⊨t t ==> (I,⟨trans-v v⟩) ⊨inss set cs and

  preprocess-unsat: preprocess cs = (t, as,trans-v,U) ==> (I,v) ⊨inss set cs ==> ∃
  v'. (I,v') ⊨ias set as ∧ v' ⊨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) ⊨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⟩ ⊨as
flat (set as) ==> ⟨v⟩ ⊨t t ==> ⟨trans-v v⟩ ⊨nss flat (set cs)
  using i-preprocess-sat[of cs t as trans-v U UNIV v] by auto

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 ⊨t t ∧ (I,v)
  ⊨ias as)) ∧
  (distinct-indices-atoms as ==> (∀ J ⊂ I. ∃ v. v ⊨t t ∧ (J,v) ⊨iaes as)))

lemma minimal-unsat-core-tabl-atomsD: assumes minimal-unsat-core-tabl-atoms
I t as
shows I ⊆ fst ` as
  ¬ (∃ v. v ⊨t t ∧ (I,v) ⊨ias as)
  distinct-indices-atoms as ==> J ⊂ I ==> ∃ v. v ⊨t t ∧ (J,v) ⊨iaes as
  using assms unfolding minimal-unsat-core-tabl-atoms-def by auto

locale AssertAll =
fixes assert-all :: tableau => ('i,'a::lrv) i-atom list => 'i list + (var, 'a)mapping
assumes assert-all-sat: △ t ==> assert-all t as = Sat v ==> ⟨v⟩ ⊨t t ∧ ⟨v⟩ ⊨as
```

```

flat (set as)
assumes assert-all-unsat:  $\Delta t \Rightarrow \text{assert-all } t \text{ as} = \text{Unsat } I \Rightarrow \text{minimal-unsat-core-tabl-atoms}$ 
(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  $\Rightarrow$  tableau  $\times$  ('i,'a) i-atom list  $\times$  ((var,'a)mapping
 $\Rightarrow$  (var,'a)mapping)  $\times$  'i list and
  assert-all :: tableau  $\Rightarrow$  ('i,'a::lrv) i-atom list  $\Rightarrow$  'i list + (var, 'a) mapping
begin
definition solve-exec-ns where

  solve-exec-ns s  $\equiv$ 
    case preprocess s of (t,as,trans-v,ui)  $\Rightarrow$ 
      (case ui of i # -  $\Rightarrow$  Inl [i] | -  $\Rightarrow$ 
       (case assert-all t as of Inl I  $\Rightarrow$  Inl I | Inr v  $\Rightarrow$  Inr (trans-v v)))
  end

  context Preprocess
  begin

    lemma preprocess-unsat-index: assumes prep: preprocess cs = (t,as,trans-v,ui)
      and i: i  $\in$  set ui
      shows minimal-unsat-core-ns {i} (set cs)
      proof -
        from preprocess-index[OF prep] have set ui  $\subseteq$  fst ' set cs by auto
        with i have i': i  $\in$  fst ' set cs by auto
        from preprocess-unsat-indices[OF prep i]
        show ?thesis unfolding minimal-unsat-core-ns-def using i' by auto
      qed

    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  $\cap$  set ui = {}
      shows minimal-unsat-core-ns I (set cs)
      proof -
        from preprocess-tableau-normalized[OF prep]
        have t:  $\Delta t$  .
        from preprocess-index[OF prep] have index: fst ' set as  $\cup$  set ui  $\subseteq$  fst ' set cs
        by auto
        from minimal-unsat-core-tabl-atomsD(1,2)[OF unsat] preprocess-unsat[OF prep,
        of I]
        have 1: I  $\subseteq$  fst ' set as  $\neg (\exists v. (I, v) \models_{\text{inss}} \text{set cs})$  by force+
        show minimal-unsat-core-ns I (set cs) unfolding minimal-unsat-core-ns-def

```

```

proof (intro conjI impI allI 1(2))
  show I ⊆ fst ` set cs using 1 index by auto
  fix J
  assume distinct-indices-ns (set cs) J ⊂ I
  with preprocess-distinct[OF prep]
  have distinct-indices-atoms (set as) J ⊂ I by auto
  from minimal-unsat-core-tabl-atomsD(3)[OF unsat this]
  obtain v where model: v ⊨t t (J, v) ⊨iases set as by auto
  from i-satisfies-atom-set'-stronger[OF model(2)]
  have model': (J, v) ⊨iass set as .
  define w where w = Mapping.Mapping (λ x. Some (v x))
  have v = ⟨w⟩ unfolding w-def map2fun-def
    by (intro ext, transfer, auto)
  with model model' have ⟨w⟩ ⊨t t (J, ⟨w⟩) ⊨iass set as by auto
  from i-preprocess-sat[OF prep - this(2,1)] ⟨J ⊂ I⟩ inter
  have (J, ⟨trans-v w⟩) ⊨inss set cs by auto
  then show ∃ w. (J, w) ⊨inss set cs by auto
qed
qed
end

sublocale Solve-exec-ns' < Solve-exec-ns solve-exec-ns
proof
  fix cs
  obtain t as trans-v ui where prep: preprocess cs = (t,as,trans-v,ui) by (cases
  preprocess cs)
  from preprocess-tableau-normalized[OF prep]
  have t: △ t .
  from preprocess-index[OF prep] have index: fst ` set as ∪ set ui ⊆ fst ` set cs
  by auto
  note solve = solve-exec-ns-def[of cs, unfolded prep split]
  {
    fix v
    assume solve-exec-ns cs = Sat v
    from this[unfolded solve] t assert-all-sat[OF t] preprocess-sat[OF prep]
    show ⟨v⟩ ⊨nss flat (set cs) by (auto split: sum.splits list.splits)
  }
  {
    fix I
    assume res: solve-exec-ns cs = Unsat I
    show minimal-unsat-core-ns (set I) (set cs)
    proof (cases ui)
      case (Cons i uis)
      hence I: I = [i] using res[unfolded solve] by auto
      from preprocess-unsat-index[OF prep, of i] I Cons index show ?thesis by
      auto
    next
      case Nil
      from res[unfolded solve Nil] have assert: assert-all t as = Unsat I
    qed
  }

```

```

    by (auto split: sum.splits)
  from assert-all-unsat[OF t assert]
  have minimal-unsat-core-tabl-atoms (set I) t (set as) .
  from preprocess-minimal-unsat-core[OF prep this] Nil
  show minimal-unsat-core-ns (set I) (set cs) by simp
qed
}
qed

```

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* \rightarrow '*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 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* \rightarrow '*a*). Upper and lower bounds are represented separately. Infinite bounds map to *None* and this is reflected in the semantics:

$$\begin{aligned} c \geq_{ub} b &\longleftrightarrow \text{case } b \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } b' \Rightarrow c \geq b' \\ c \leq_{ub} b &\longleftrightarrow \text{case } b \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } b' \Rightarrow c \leq b' \end{aligned}$$

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

```

abbreviation (input) le where
  le lt x y ≡ lt x y ∨ x = y
definition geub (⟨≤ub⟩) where
  △ub lt c b ≡ case b of None ⇒ False | Some b' ⇒ le lt b' c
definition gtub (⟨▷ub⟩) where
  ▷ub lt c b ≡ case b of None ⇒ False | Some b' ⇒ lt b' c
definition leub (⟨≤lb⟩) where
  ≤ub lt c b ≡ case b of None ⇒ True | Some b' ⇒ le lt c b'
definition ltub (⟨▷lb⟩) where
  ▷lb lt c b ≡ case b of None ⇒ True | Some b' ⇒ lt c b'
definition lelb (⟨≤lb⟩) where
  ≤lb lt c b ≡ case b of None ⇒ False | Some b' ⇒ le lt c b'
definition ltlb (⟨▷lb⟩) where
  ▷lb lt c b ≡ case b of None ⇒ False | Some b' ⇒ lt c b'
definition gelb (⟨▷lb⟩) where
  ▷lb lt c b ≡ case b of None ⇒ True | Some b' ⇒ le lt b' c
definition gtlb (⟨▷lb⟩) where
  ▷lb lt c b ≡ case b of None ⇒ True | Some b' ⇒ lt b' c

```

definition *ge-ubound* :: '*a*::linorder \Rightarrow '*a* option \Rightarrow bool (**infixl** \geq_{ub} 100) **where**

```

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

```

```

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]:
 $\trianglelefteq_{lb} (>) a b = \sqsupseteq_{ub} (<) a b$ 
 $\trianglelefteq_{ub} (>) a b = \trianglelefteq_{lb} (<) a b$ 
 $\sqsupseteq_{lb} (>) a b = \trianglelefteq_{ub} (<) a b$ 
 $\sqsupseteq_{ub} (>) a b = \trianglelefteq_{lb} (<) a b$ 
 $\trianglelefteq_{lb} (>) a b = \triangleright_{ub} (<) a b$ 
 $\trianglelefteq_{ub} (>) a b = \triangleright_{lb} (<) a b$ 
 $\triangleright_{lb} (>) a b = \triangleleft_{ub} (<) a b$ 
 $\triangleright_{ub} (>) a b = \triangleleft_{lb} (<) a b$ 
by (case-tac[!] b) (auto simp add: bound-compare'-defs)

```

```

lemma [simp]:  $\neg c \geq_{ub} \text{None} \quad \neg c \leq_{lb} \text{None}$ 
by (auto simp add: bound-compare-defs)

```

```

lemma neg-bounds-compare:
 $(\neg (c \geq_{ub} b)) \implies c <_{ub} b \quad (\neg (c \leq_{ub} b)) \implies c >_{ub} b$ 
 $(\neg (c >_{ub} b)) \implies c \leq_{ub} b \quad (\neg (c <_{ub} b)) \implies c \geq_{ub} b$ 
 $(\neg (c \leq_{lb} b)) \implies c >_{lb} b \quad (\neg (c \geq_{lb} b)) \implies c <_{lb} b$ 

```

$(\neg (c <_{lb} b)) \Rightarrow c \geq_{lb} b$ $(\neg (c >_{lb} b)) \Rightarrow c \leq_{lb} b$
by (case-tac[!] b) (auto simp add: bound-compare-defs)

lemma bounds-compare-contradictory [simp]:

$\llbracket c \geq_{ub} b; c <_{ub} b \rrbracket \Rightarrow False$ $\llbracket c \leq_{ub} b; c >_{ub} b \rrbracket \Rightarrow False$
 $\llbracket c >_{ub} b; c \leq_{ub} b \rrbracket \Rightarrow False$ $\llbracket c <_{ub} b; c \geq_{ub} b \rrbracket \Rightarrow False$
 $\llbracket c \leq_{lb} b; c >_{lb} b \rrbracket \Rightarrow False$ $\llbracket c \geq_{lb} b; c <_{lb} b \rrbracket \Rightarrow False$
 $\llbracket c <_{lb} b; c \geq_{lb} b \rrbracket \Rightarrow False$ $\llbracket c >_{lb} b; c \leq_{lb} b \rrbracket \Rightarrow False$
by (case-tac[!] b) (auto simp add: bound-compare-defs)

lemma compare-strict-nonstrict:

$x <_{ub} b \Rightarrow x \leq_{ub} b$
 $x >_{ub} b \Rightarrow x \geq_{ub} b$
 $x <_{lb} b \Rightarrow x \leq_{lb} b$
 $x >_{lb} b \Rightarrow x \geq_{lb} b$
by (case-tac[!] b) (auto simp add: bound-compare-defs)

lemma [simp]:

$\llbracket x \leq c; c <_{ub} b \rrbracket \Rightarrow x <_{ub} b$
 $\llbracket x < c; c \leq_{ub} b \rrbracket \Rightarrow x <_{ub} b$
 $\llbracket x \leq c; c \leq_{ub} b \rrbracket \Rightarrow x \leq_{ub} b$
 $\llbracket x \geq c; c >_{lb} b \rrbracket \Rightarrow x >_{lb} b$
 $\llbracket x > c; c \geq_{lb} b \rrbracket \Rightarrow x >_{lb} b$
 $\llbracket x \geq c; c \geq_{lb} b \rrbracket \Rightarrow x \geq_{lb} b$
by (case-tac[!] b) (auto simp add: bound-compare-defs)

lemma bounds-lg [simp]:

$\llbracket c >_{ub} b; x \leq_{ub} b \rrbracket \Rightarrow x < c$
 $\llbracket c \geq_{ub} b; x <_{ub} b \rrbracket \Rightarrow x < c$
 $\llbracket c \geq_{ub} b; x \leq_{ub} b \rrbracket \Rightarrow x \leq c$
 $\llbracket c <_{lb} b; x \geq_{lb} b \rrbracket \Rightarrow x > c$
 $\llbracket c \leq_{lb} b; x >_{lb} b \rrbracket \Rightarrow x > c$
 $\llbracket c \leq_{lb} b; x \geq_{lb} b \rrbracket \Rightarrow x \geq c$
by (case-tac[!] b) (auto simp add: bound-compare-defs)

lemma bounds-compare-*Some* [simp]:

$x \leq_{ub} \text{Some } c \leftrightarrow x \leq c$ $x \geq_{ub} \text{Some } c \leftrightarrow x \geq c$
 $x <_{ub} \text{Some } c \leftrightarrow x < c$ $x >_{ub} \text{Some } c \leftrightarrow x > c$
 $x \geq_{lb} \text{Some } c \leftrightarrow x \geq c$ $x \leq_{lb} \text{Some } c \leftrightarrow x \leq c$
 $x >_{lb} \text{Some } c \leftrightarrow x > c$ $x <_{lb} \text{Some } c \leftrightarrow x < c$
by (auto simp add: bound-compare-defs)

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 \Rightarrow '*a* bounds \times '*a* bounds \Rightarrow bool
(infixl \triangleleft_b 100) **where**
 $v \triangleleft_b b \leftrightarrow (\forall x. \text{in-bounds } x v b)$
declare *satisfies-bounds.simps* [simp del]

```

lemma satisfies-bounds-iff:
   $v \models_b (lb, ub) \longleftrightarrow (\forall x. v x \geq_{lb} lb x \wedge v x \leq_{ub} ub x)$ 
  by (auto simp add: satisfies-bounds.simps)

lemma not-in-bounds:
   $\neg (in\text{-}bounds x v (lb, ub)) = (v x <_{lb} lb x \vee v x >_{ub} ub x)$ 
  using bounds-compare-contradictory(7)
  using bounds-compare-contradictory(2)
  using neg-bounds-compare(7)[of v x lb x]
  using neg-bounds-compare(2)[of v x ub x]
  by auto

fun atoms-equiv-bounds :: 'a::linorder atom set  $\Rightarrow$  'a bounds  $\times$  'a bounds  $\Rightarrow$  bool
(infixl  $\doteqdot$  100) where
  as  $\doteqdot (lb, ub) \longleftrightarrow (\forall v. v \models_{as} as \longleftrightarrow v \models_b (lb, ub))$ 
declare atoms-equiv-bounds.simps [simp del]

lemma atoms-equiv-bounds-simps:
  as  $\doteqdot (lb, ub) \equiv \forall v. v \models_{as} as \longleftrightarrow v \models_b (lb, ub)$ 
  by (simp add: atoms-equiv-bounds.simps)

```

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

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 know 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 ( $\langle \mathcal{I}_l \rangle$ ) where
 $\mathcal{I}_l s = (\text{fst } o \text{the}) o \text{look } (\mathcal{B}_{il} s)$ 

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

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

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

abbreviation BoundsIndicesMap ( $\langle \mathcal{B}_i \rangle$ ) 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 ( $\langle \mathcal{B} \rangle$ ) 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 ( $\langle \mathcal{I} \rangle$ )
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) ( $\langle \mathcal{BI} \rangle$ )
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  $\text{\textlangle\textgreater}_{ib}$  100) where
 $(I, v) \models_{ib} ((BL, BU), (IL, IU)) \longleftrightarrow$ 
 $(\forall x c. BL x = \text{Some } c \longrightarrow IL x \in I \longrightarrow v x \geq c)$ 
 $\wedge (\forall x c. BU x = \text{Some } c \longrightarrow IU x \in I \longrightarrow v x \leq c)$ 
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  $\text{\textlangle\textgreater}_{ibe}$  100) where
 $(I, v) \models_{ibe} ((BL, BU), (IL, IU)) \longleftrightarrow$ 
 $(\forall x c. BL x = \text{Some } c \longrightarrow IL x \in I \longrightarrow v x = c)$ 
 $\wedge (\forall x c. BU x = \text{Some } c \longrightarrow IU x \in I \longrightarrow v x = c)$ 
declare satisfies-bounds-index'.simp[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  $\text{\textlangle\textgreater}_i$  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:  $as \subseteq as' \implies v \models_{ias} as' \implies v \models_{ias} as$ 
  by (cases  $v$ , auto simp: satisfies-atom-set-def)

lemma atoms-imply-bounds-index-mono:  $as \subseteq as' \implies as \models_i bi \implies as' \models_i bi$ 
  unfolding atoms-imply-bounds-index.simps using i-satisfies-atom-set-mono by blast

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 ( $\langle\models\rangle$ ) 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  $\times$  'a::lrv valuation  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  bool (infixl  $\models_{ise}$ ) 100) where
   $(I,v) \models_{ise} s \longleftrightarrow (v \models_t \mathcal{T} s \wedge (I,v) \models_{ibe} \mathcal{BI} s)$ 
  declare satisfies-state-index'.simp[simp del]

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

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  $\Rightarrow$  bool where
   $distinct-indices-state s = (\forall i x b x' b').$ 
   $((look(\mathcal{B}_{il} s) x = Some(i,b) \vee look(\mathcal{B}_{iu} s) x = Some(i,b)) \longrightarrow$ 
   $(look(\mathcal{B}_{il} s) x' = Some(i,b') \vee look(\mathcal{B}_{iu} s) x' = Some(i,b')) \longrightarrow$ 
   $(x = x' \wedge b = b'))$ 

lemma distinct-indices-stateD: assumes distinct-indices-state  $s$ 
  shows  $look(\mathcal{B}_{il} s) x = Some(i,b) \vee look(\mathcal{B}_{iu} s) x = Some(i,b) \implies look(\mathcal{B}_{il} s) x' = Some(i,b') \vee look(\mathcal{B}_{iu} s) x' = Some(i,b')$ 
   $\implies x = x' \wedge b = b'$ 
  using assms unfolding distinct-indices-state-def by blast+

definition unsat-state-core :: ('i,'a::lrv) state  $\Rightarrow$  bool where
   $unsat-state-core s = (set(the(\mathcal{U}_c s)) \subseteq indices-state s \wedge (\neg(\exists v. (set(the(\mathcal{U}_c s)), v) \models_{is} s)))$ 

definition subsets-sat-core :: ('i,'a::lrv) state  $\Rightarrow$  bool where
   $subsets-sat-core s = ((\forall I. I \subset set(the(\mathcal{U}_c s)) \longrightarrow (\exists v. (I, v) \models_{ise} s)))$ 

```

```

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
  unfolding minimal-unsat-core-tabl-atoms-def
proof (intro conjI impI allI)
  note min = unsat[unfolded minimal-unsat-core-tabl-atoms-def]
  from min have I: I ⊆ fst ` as by blast
  with sub show I ⊆ fst ` bs by blast
  from min have (∀ v. v ⊨t t ∧ (I, v) ⊨i as as) by blast
  with i-satisfies-atom-set-mono[OF sub]
  show (∀ v. v ⊨t t ∧ (I, v) ⊨i as bs) by blast
  fix J
  assume J: J ⊂ I and dist-bs: distinct-indices-atoms bs
  hence dist: distinct-indices-atoms as
    using sub unfolding distinct-indices-atoms-def by blast
  from min dist J obtain v where v: v ⊨t t (J, v) ⊨i as as by blast
  have (J, v) ⊨i as bs
  unfolding i-satisfies-atom-set'.simp
  proof (intro ballI)
    fix a
    assume a ∈ snd ` (bs ∩ J × UNIV)
    then obtain i where ia: (i, a) ∈ bs and i: i ∈ J
      by force
    with J have i ∈ I by auto
    with I obtain b where ib: (i, b) ∈ as by force
    with sub have ib': (i, b) ∈ bs by auto
    from dist-bs[unfolded distinct-indices-atoms-def, rule-format, OF ia ib']
    have id: atom-var a = atom-var b atom-const a = atom-const b by auto
    from v(2)[unfolded i-satisfies-atom-set'.simp] i ib
    have v ⊨ae b by force
    thus v ⊨ae a using id unfolding satisfies-atom'-def by auto
  qed
  with v show ∃ v. v ⊨t t ∧ (J, v) ⊨i as bs by blast
qed

lemma state-satisfies-index: assumes v ⊨s s
  shows (I, v) ⊨i s
  unfolding satisfies-state-index.simps satisfies-bounds-index.simps
proof (intro conjI impI allI)
  fix x c
  from assms[unfolded satisfies-state-def satisfies-bounds.simps, simplified]
  have v ⊨t T s and bnd: v x ≥lb B_l s x v x ≤ub B_u s x by auto
  show v ⊨t T s by fact
  show B_l s x = Some c ==> I_l s x ∈ I ==> c ≤ v x
  using bnd(1) by auto

```

```

show  $\mathcal{B}_u s x = \text{Some } c \implies \mathcal{I}_u s x \in I \implies v x \leq c$ 
  using  $bnd(2)$  by auto
qed

lemma unsat-state-core-unsat:  $\text{unsat-state-core } s \implies \neg (\exists v. v \models_s s)$ 
  unfolding unsat-state-core-def using state-satisfies-index by blast

definition tableau-valuated ( $\langle \nabla \rangle$ ) where
   $\nabla s \equiv \forall x \in \text{tvars } (\mathcal{T} s). \text{Mapping.lookup } (\mathcal{V} s) x \neq \text{None}$ 

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

lemma index-valid-indices-state:  $\text{index-valid as } s \implies \text{indices-state } s \subseteq \text{fst } 'as$ 
  unfolding index-valid-def indices-state-def by force

lemma index-valid-mono:  $as \subseteq bs \implies \text{index-valid as } s \implies \text{index-valid bs } s$ 
  unfolding index-valid-def by blast

lemma index-valid-distinct-indices: assumes index-valid as s
  and distinct-indices-atoms as
  shows distinct-indices-state s
  unfolding distinct-indices-state-def
  proof (intro allI impI, goal-cases)
    case (1 i x b y c)
    note valid = assms(1)[unfolded index-valid-def, rule-format]
    from 1(1) valid[of x i b] have  $(i, \text{Geq } x b) \in as \vee (i, \text{Leq } x b) \in as$  by auto
    then obtain a where  $a: (i,a) \in as$  atom-var  $a = x$  atom-const  $a = b$  by auto
    from 1(2) valid[of y i c] have  $y: (i, \text{Geq } y c) \in as \vee (i, \text{Leq } y c) \in as$  by auto
    then obtain a' where  $a': (i,a') \in as$  atom-var  $a' = y$  atom-const  $a' = c$  by auto
    from assms(2)[unfolded distinct-indices-atoms-def, rule-format, OF a(1) a'(1)]
    show ?case using a a' by auto
qed

```

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 and defined by $\models s \equiv \langle \mathcal{V} s \rangle \models_s s$. ∇s will denote that all variables of $\mathcal{T} s$ are explicitly valuated in $\mathcal{V} s$.

definition *updateBI* **where**

```

[simp]: update $\mathcal{BI}$  field-update  $i\ x\ c\ s = \text{field-update}(\text{upd}\ x\ (i,c))\ s$ 

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

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

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

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

lemma update-simps[simp]:
 $\mathcal{B}_{iu}(\mathcal{B}_{iu}\text{-update up } s) = \text{up } (\mathcal{B}_{iu} s)$ 
 $\mathcal{B}_{il}(\mathcal{B}_{iu}\text{-update up } s) = \mathcal{B}_{il} s$ 
 $\mathcal{T}(\mathcal{B}_{iu}\text{-update up } s) = \mathcal{T} s$ 
 $\mathcal{V}(\mathcal{B}_{iu}\text{-update up } s) = \mathcal{V} s$ 
 $\mathcal{U}(\mathcal{B}_{iu}\text{-update up } s) = \mathcal{U} s$ 
 $\mathcal{U}_c(\mathcal{B}_{iu}\text{-update up } s) = \mathcal{U}_c s$ 
 $\mathcal{B}_{il}(\mathcal{B}_{il}\text{-update up } s) = \text{up } (\mathcal{B}_{il} s)$ 
 $\mathcal{B}_{iu}(\mathcal{B}_{il}\text{-update up } s) = \mathcal{B}_{iu} s$ 
 $\mathcal{T}(\mathcal{B}_{il}\text{-update up } s) = \mathcal{T} s$ 
 $\mathcal{V}(\mathcal{B}_{il}\text{-update up } s) = \mathcal{V} s$ 
 $\mathcal{U}(\mathcal{B}_{il}\text{-update up } s) = \mathcal{U} s$ 
 $\mathcal{U}_c(\mathcal{B}_{il}\text{-update up } s) = \mathcal{U}_c s$ 
 $\mathcal{V}(\mathcal{V}\text{-update } V s) = V$ 
 $\mathcal{B}_{il}(\mathcal{V}\text{-update } V s) = \mathcal{B}_{il} s$ 
 $\mathcal{B}_{iu}(\mathcal{V}\text{-update } V s) = \mathcal{B}_{iu} s$ 
 $\mathcal{T}(\mathcal{V}\text{-update } V s) = \mathcal{T} s$ 
 $\mathcal{U}(\mathcal{V}\text{-update } V s) = \mathcal{U} s$ 
 $\mathcal{U}_c(\mathcal{V}\text{-update } V s) = \mathcal{U}_c s$ 
 $\mathcal{T}(\mathcal{T}\text{-update } T s) = T$ 
 $\mathcal{B}_{il}(\mathcal{T}\text{-update } T s) = \mathcal{B}_{il} s$ 
 $\mathcal{B}_{iu}(\mathcal{T}\text{-update } T s) = \mathcal{B}_{iu} s$ 
 $\mathcal{V}(\mathcal{T}\text{-update } T s) = \mathcal{V} s$ 
 $\mathcal{U}(\mathcal{T}\text{-update } T s) = \mathcal{U} s$ 
 $\mathcal{U}_c(\mathcal{T}\text{-update } T s) = \mathcal{U}_c s$ 
by (atomize(full), cases s, auto)

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

fun set-unsat :: 'i list  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state where
 $\text{set-unsat } I \text{ (State } T \text{ BIL BIU } V \text{ U UC}) = \text{State } T \text{ BIL BIU } V \text{ True } (\text{Some } (\text{remdups } I))$ 

```

```

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  $I$ )
by (atomize(full), cases  $s$ , auto)

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

```

definition Positive **where**

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

definition Negative **where**

[simp]: Negative \equiv Direction ($>$) $\mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_u \mathcal{B}_l \mathcal{I}_u \mathcal{I}_l \mathcal{B}_{il}$ -update Geq 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:

```

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

```

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

```

locale AssertAllState = fixes assert-all-state::tableau  $\Rightarrow$  ('i,'a::lrv) i-atom list  $\Rightarrow$  ('i,'a) state
assumes
  — The final and the initial tableau are equivalent.
  assert-all-state-tableau-equiv:  $\Delta t \Rightarrow$  assert-all-state t as = s'  $\Rightarrow$  (v::'a 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).

assert-all-state-sat: $\Delta t \implies \text{assert-all-state } t \text{ as} = s' \implies \neg \mathcal{U} s' \implies \models s' \text{ and}$

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

— 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).

assert-all-state-unsat: $\Delta t \implies \text{assert-all-state } t \text{ as} = s' \implies \mathcal{U} s' \implies \text{minimal-unsat-state-core } s' \text{ and}$

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

— The set of indices is taken from the constraints

assert-all-state-indices: $\Delta t \implies \text{assert-all-state } t \text{ as} = s \implies \text{indices-state } s \subseteq \text{fst ('set as) and}$

assert-all-index-valid: $\Delta t \implies \text{assert-all-state } t \text{ as} = s \implies \text{index-valid (set as) } s$
begin
definition assert-all where
assert-all t as \equiv *let s = assert-all-state t as in*
if (U s) then Unsat (the (U_c s)) else Sat (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 (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*: $\Delta t \implies \Delta (\mathcal{T} (\text{init } t))$
assumes *init'-tableau-equiv*: $\Delta t \implies (v::'a \text{ valuation}) \models_t t = v \models_t \mathcal{T} (\text{init } t)$
assumes *init'-sat*: $\Delta t \implies \neg \mathcal{U} (\text{init } t) \longrightarrow \models (\text{init } t)$
assumes *init'-unsat*: $\Delta t \implies \mathcal{U} (\text{init } t) \longrightarrow \text{minimal-unsat-state-core } (\text{init } t)$
assumes *init'-atoms-equiv-bounds*: $\Delta t \implies \{\} \doteq \mathcal{B} (\text{init } t)$
assumes *init'-atoms-imply-bounds-index*: $\Delta t \implies \{\} \models_i \mathcal{BI} (\text{init } t)$

Specification for *init* can be obtained from the specification of *assert-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 $\{\} \doteq \mathcal{B} (\text{init } t)$ 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:  $\langle \mathcal{V} (\text{init } t) \rangle \models_t$  t and
  — In an initial state no atoms are yet asserted so the bounds must be empty:
    init-bounds:  $\mathcal{B}_{il} (\text{init } t) = \text{Mapping.empty}$   $\mathcal{B}_{iu} (\text{init } t) = \text{Mapping.empty}$  and
  — All tableau vars are valuated: init-tableau-valuated:  $\nabla (\text{init } t)$ 

begin

lemma init-satisfies-bounds:
 $\langle \mathcal{V} (\text{init } t) \rangle \models_b \mathcal{B} (\text{init } t)$ 
using init-bounds
unfolding satisfies-bounds.simps in-bounds.simps bound-compare-defs
by (auto simp: boundsl-def boundsu-def)

lemma init-satisfies:
 $\models (\text{init } t)$ 
using init-satisfies-tableau init-satisfies-bounds init-tableau-id
unfolding curr-val-satisfies-state-def satisfies-state-def
by simp

lemma init-atoms-equiv-bounds:
 $\{\} \doteq \mathcal{B} (\text{init } t)$ 
using init-bounds
unfolding atoms-equiv-bounds.simps satisfies-bounds.simps in-bounds.simps satisfies-atom-set-def
unfolding bound-compare-defs
by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)

lemma init-atoms-imply-bounds-index:
 $\{\} \models_i \mathcal{BI} (\text{init } t)$ 
using init-bounds
unfolding atoms-imply-bounds-index.simps satisfies-bounds-index.simps in-bounds.simps
i-satisfies-atom-set.simps satisfies-atom-set-def
unfolding bound-compare-defs
by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)

```

```

lemma init-tableau-normalized:
   $\Delta t \implies \Delta (\mathcal{T} (\text{init } t))$ 
  using init-tableau-id
  by simp

lemma init-index-valid: index-valid as (init t)
  using init-bounds unfolding index-valid-def by auto

lemma init-indices: indices-state (init t) = {}
  unfolding indices-state-def init-bounds by auto
end

sublocale Init < Init' init
  using init-tableau-id init-satisfies init-unsat-flag init-atoms-equiv-bounds init-atoms-imply-bounds-index
  by unfold-locales auto

abbreviation vars-list where
  vars-list t ≡ remdups (map lhs t @ (concat (map (Abstract-Linear-Poly.vars-list
  o rhs) t)))

lemma tvars t = set (vars-list t)
  by (auto simp add: set-vars-list lvars-def rvars-def)

```

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 $\neg \mathcal{U} s, \models s, \Delta (\mathcal{T} s)$ and ∇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 ⇒ ('i,'a) state ⇒ ('i,'a) state
  assumes
    — Tableau remains equivalent to the previous one and normalized and valuated.
    assert-tableau:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies \text{let } s' = \text{assert } a \text{ in}$ 
     $((v::'a \text{ valuation}) \models_t \mathcal{T} s \leftrightarrow v \models_t \mathcal{T} s') \wedge \Delta (\mathcal{T} s') \wedge \nabla s' \text{ and}$ 
    — 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:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies \neg \mathcal{U} (\text{assert } a \text{ s}) \implies \models (\text{assert } a \text{ s})$ 

```

and

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

assert-atoms-equiv-bounds: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \text{flat ats} \doteq \mathcal{B} s \implies \text{flat}(\text{ats} \cup \{a\}) \doteq \mathcal{B}(\text{assert } a s)$ **and**

— There is a subset of asserted atoms which remains index-equivalent to the bounds in the state.

assert-atoms-implies-bounds-index: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \text{ats} \models_i \mathcal{BI} s \implies \text{insert } a \text{ ats} \models_i \mathcal{BI}(\text{assert } a s)$ **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: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s; \text{index-valid ats } s \rrbracket \implies \mathcal{U}(\text{assert } a s) \implies \text{minimal-unsat-state-core}(\text{assert } a s)$ **and**

assert-index-valid: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \text{index-valid ats } s \implies \text{index-valid}(\text{insert } a \text{ ats})(\text{assert } a s)$

begin

lemma *assert-tableau-equiv*: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies (\text{valuation } v :: a) \models_t \mathcal{T} s \longleftrightarrow v \models_t \mathcal{T}(\text{assert } a s)$

using *assert-tableau*

by (*simp add: Let-def*)

lemma *assert-tableau-normalized*: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \Delta(\mathcal{T}(\text{assert } a s))$

using *assert-tableau*

by (*simp add: Let-def*)

lemma *assert-tableau-valuated*: $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \nabla(\text{assert } a s)$

using *assert-tableau*

by (*simp add: Let-def*)

end

locale *AssertAllState'* = *Init init + Assert assert for*
 $\text{init} :: \text{tableau} \Rightarrow ('i, 'a :: lrv) \text{ state}$ **and** $\text{assert} :: ('i, 'a) \text{ i-atom} \Rightarrow ('i, 'a) \text{ state} \Rightarrow ('i, 'a) \text{ state}$
begin

definition *assert-loop where*

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

definition *assert-all-state where [simp]*:

assert-all-state t as \equiv *assert-loop as (init t)*

```

lemma AssertAllState'-precond:
   $\triangle t \implies \triangle (\mathcal{T}(\text{assert-all-state } t \text{ as}))$ 
   $\wedge (\nabla(\text{assert-all-state } t \text{ as}))$ 
   $\wedge (\neg \mathcal{U}(\text{assert-all-state } t \text{ as}) \longrightarrow \models(\text{assert-all-state } t \text{ as}))$ 
unfolding assert-all-state-def assert-loop-def
using init-satisfies init-tableau-normalized init-index-valid
using assert-sat assert-tableau-normalized init-tableau-valuated assert-tableau-valuated
by (induct as rule: rev-induct) auto

lemma AssertAllState'Induct:
assumes
   $\triangle t$ 
   $P \{\} (\text{init } t)$ 
   $\wedge \text{as } bs \text{ t. as } \subseteq bs \implies P \text{ as } t \implies P \text{ bs } t$ 
   $\wedge \text{as } s \text{ as. } \llbracket \neg \mathcal{U} s; \models s; \triangle (\mathcal{T} s); \nabla s; P \text{ as } s; \text{index-valid as } s \rrbracket \implies P (\text{insert as as}) (\text{assert as } s)$ 
shows  $P (\text{set as}) (\text{assert-all-state } t \text{ as})$ 
proof -
  have  $P (\text{set as}) (\text{assert-all-state } t \text{ as}) \wedge \text{index-valid (set as) (assert-all-state } t \text{ as)}$ 
  proof (induct as rule: rev-induct)
    case Nil
    then show ?case
      unfolding assert-all-state-def assert-loop-def
      using assms(2) init-index-valid by auto
    next
      case (snoc a as')
        let ?f =  $\lambda s'. a. \text{if } \mathcal{U} s' \text{ then } s' \text{ else assert a } s'$ 
        let ?s = foldl ?f (init t) as'
        show ?case
        proof (cases  $\mathcal{U}$  ?s)
          case True
          from snoc index-valid-mono[of - set (a # as') (assert-all-state t as')]
          have index: index-valid (set (a # as')) (assert-all-state t as')
            by auto
          from snoc assms(3)[of set as' set (a # as')]
          have P:  $P (\text{set (a # as')}) (\text{assert-all-state } t \text{ as'})$  by auto
          show ?thesis
            using True P index
            unfolding assert-all-state-def assert-loop-def
            by simp
        next
          case False
          then show ?thesis
            using snoc
            using assms(1) assms(4)
            using AssertAllState'-precond assert-index-valid
            unfolding assert-all-state-def assert-loop-def
            by auto
    qed

```

```

qed
then show ?thesis ..
qed

lemma AssertAllState-index-valid:  $\Delta t \implies \text{index-valid}(\text{set } as) (\text{assert-all-state } t as)$ 
by (rule AssertAllState'Induct, auto intro: assert-index-valid init-index-valid index-valid-mono)

lemma AssertAllState'-sat-atoms-equiv-bounds:
 $\Delta t \implies \neg \mathcal{U}(\text{assert-all-state } t as) \implies \text{flat}(\text{set } as) \doteq \mathcal{B}(\text{assert-all-state } t as)$ 
using AssertAllState'-precond
using init-atoms-equiv-bounds assert-atoms-equiv-bounds
unfolding assert-all-state-def assert-loop-def
by (induct as rule: rev-induct) auto

lemma AssertAllState'-unsat-atoms-implies-bounds:
assumes  $\Delta t$ 
shows set as  $\models_i \mathcal{BI}$  (assert-all-state t as)
proof (induct as rule: rev-induct)
case Nil
then show ?case
using assms init-atoms-implies-bounds-index
unfolding assert-all-state-def assert-loop-def
by simp
next
case (snoc a as')
let ?s = assert-all-state t as'
show ?case
proof (cases  $\mathcal{U} ?s$ )
case True
then show ?thesis
using snoc atoms-implies-bounds-index-mono[of set as' set (as' @ [a])]
unfolding assert-all-state-def assert-loop-def
by auto
next
case False
then have id: assert-all-state t (as' @ [a]) = assert a ?s
unfolding assert-all-state-def assert-loop-def by simp
from snoc have as': set as'  $\models_i \mathcal{BI}$  ?s by auto
from AssertAllState'-precond[of t as'] assms False
have  $\models ?s \Delta (\mathcal{T} ?s) \nabla ?s$  by auto
from assert-atoms-implies-bounds-index[OF False this as', of a]
show ?thesis unfolding id by auto
qed
qed

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
  fix v::'a valuation and t as s'
  assume*:  $\triangle t$  and id: assert-all-state t as = s'
  note idsym = id[symmetric]

  show v  $\models_t$  t = v  $\models_t$   $\mathcal{T}$  s' unfolding idsym
    using init-tableau-id[of t] assert-tableau-equiv[of - v]
    by (induct rule: AssertAllState'Induct) (auto simp add: *)

  show  $\neg \mathcal{U}$  s'  $\implies$   $\models$  s' unfolding idsym
    using AssertAllState'-precond by (simp add: *)

  show  $\neg \mathcal{U}$  s'  $\implies$  flat (set as)  $\doteq \mathcal{B}$  s'
    unfolding idsym
    using *
    by (rule AssertAllState'-sat-atoms-equiv-bounds)

  show  $\mathcal{U}$  s'  $\implies$  set as  $\models_i \mathcal{BI}$  s'
    using* unfolding idsym
    by (rule AssertAllState'-unsat-atoms-implies-bounds)

  show  $\mathcal{U}$  s'  $\implies$  minimal-unsat-state-core s'
    using init-unsat-flag assert-unsat assert-index-valid unfolding idsym
    by (induct rule: AssertAllState'Induct) (auto simp add: *)

  show indices-state s'  $\subseteq$  fst ` set as unfolding idsym using *
    by (intro index-valid-indices-state, induct rule: AssertAllState'Induct,
         auto simp: init-index-valid index-valid-mono assert-index-valid)

  show index-valid (set as) s' using* AssertAllState-index-valid idsym by blast
qed

```

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-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 ( $\dashv\models_b$  -  $\|$  /  $\rightarrow$ )
syntax-consts
  -satisfies-bounds-set == satisfies-bounds-set
translations
   $v \models_b b \| S == CONST$  satisfies-bounds-set  $v b S$ 
lemma satisfies-bounds-set-iff:
   $v \models_b (lb, ub) \| S \equiv (\forall x \in S. v x \geq_{lb} lb x \wedge v x \leq_{ub} ub x)$ 
  by (simp add: satisfies-bounds-set.simps)

definition curr-val-satisfies-no-lhs ( $\models_{nolhs}$ ) where
   $\models_{nolhs} s \equiv \langle \mathcal{V} s \rangle \models_t (\mathcal{T} s) \wedge (\langle \mathcal{V} s \rangle \models_b (\mathcal{B} s) \| (\dashv\text{lvars} (\mathcal{T} s)))$ 
lemma satisfies-satisfies-no-lhs:
   $\models s \Longrightarrow \models_{nolhs} s$ 
  by (simp add: curr-val-satisfies-state-def satisfies-state-def curr-val-satisfies-no-lhs-def
  satisfies-bounds.simps satisfies-bounds-set.simps)

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  $v \models_b (lb, ub) \| S$ , and define  $v \models_b (lb, ub) \| S \equiv \forall x \in S. v x \geq_{lb} lb x \wedge v x \leq_{ub} ub x$ , and  $\models_{nolhs} s \equiv \langle \mathcal{V} s \rangle \models_t \mathcal{T} s \wedge \langle \mathcal{V} s \rangle \models_b \mathcal{B} s \| \dashv\text{lvars} (\mathcal{T} s)$ . 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:

$(v::'a::linorder valuation) \models_b \mathcal{B} s \longrightarrow \diamond s$

unfolding satisfies-bounds.simps in-bounds.simps bounds-consistent-def bound-compare-defs
by (auto split: option.split) force

```

lemma satisfies-consistent:
   $\models s \longrightarrow \Diamond s$ 
  by (auto simp add: curr-val-satisfies-state-def satisfies-state-def satisfies-bounds-consistent)

lemma bounds-consistent-geq-lb:
   $\llbracket \Diamond s; \mathcal{B}_u s x_i = \text{Some } c \rrbracket$ 
   $\implies c \geq_{lb} \mathcal{B}_l s x_i$ 
  unfolding bounds-consistent-def
  by (cases  $\mathcal{B}_l s x_i$ , auto simp add: bound-compare-defs split: if-splits)
    (erule-tac  $x=x_i$  in allE, auto)

lemma bounds-consistent-leq-ub:
   $\llbracket \Diamond s; \mathcal{B}_l s x_i = \text{Some } c \rrbracket$ 
   $\implies c \leq_{ub} \mathcal{B}_u s x_i$ 
  unfolding bounds-consistent-def
  by (cases  $\mathcal{B}_u s x_i$ , auto simp add: bound-compare-defs split: if-splits)
    (erule-tac  $x=x_i$  in allE, auto)

lemma bounds-consistent-gt-ub:
   $\llbracket \Diamond s; c <_{lb} \mathcal{B}_l s x \rrbracket \implies \neg c >_{ub} \mathcal{B}_u s x$ 
  unfolding bounds-consistent-def
  by (case-tac[!]  $\mathcal{B}_l s x$ , case-tac[!]  $\mathcal{B}_u s x$ )
    (auto simp add: bound-compare-defs, erule-tac  $x=x$  in allE, simp)

lemma bounds-consistent-lt-lb:
   $\llbracket \Diamond s; c >_{ub} \mathcal{B}_u s x \rrbracket \implies \neg c <_{lb} \mathcal{B}_l s x$ 
  unfolding bounds-consistent-def
  by (case-tac[!]  $\mathcal{B}_l s x$ , case-tac[!]  $\mathcal{B}_u s x$ )
    (auto simp add: bound-compare-defs, erule-tac  $x=x$  in allE, simp)

```

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 specification for the *assert-bound* is:

```

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

```

assert-bound-tableau: $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies \text{assert-bound } a s = s' \implies \mathcal{T} s' = \mathcal{T} s \wedge \nabla s'$ **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: $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies \text{assert-bound } a s = s' \implies \neg \mathcal{U} s' \implies \models_{\text{nolhs}} s' \wedge \Diamond s'$ **and**

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

assert-bound-atoms-equiv-bounds: $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies$

```

flat  $ats \doteqdot \mathcal{B} s \implies flat(ats \cup \{a\}) \doteqdot \mathcal{B} (\text{assert-bound } a s)$  and
assert-bound-atoms-imply-bounds-index:  $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies$ 
 $ats \models_i \mathcal{BI} s \implies insert a ats \models_i \mathcal{BI} (\text{assert-bound } a s)$  and
—  $\mathcal{U}$  flag is raised, only if the bounds became inconsistent:
assert-bound-unsat:  $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies index\text{-valid } as s \implies as-$ 
sert-bound  $a s = s' \implies \mathcal{U} s' \implies minimal\text{-unsat-state-core } s'$  and
assert-bound-index-valid:  $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies index\text{-valid } as s \implies$ 
index-valid  $(insert a as) (\text{assert-bound } a s)$ 

begin
lemma assert-bound-tableau-id:  $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \mathcal{T} (\text{assert-bound } a s) = \mathcal{T} s$ 
using assert-bound-tableau by blast

lemma assert-bound-tableau-valuated:  $\llbracket \neg \mathcal{U} s; \models s; \Delta(\mathcal{T} s); \nabla s \rrbracket \implies \nabla (\text{assert-bound } a s)$ 
using assert-bound-tableau by blast

end

locale AssertBoundNoLhs =
fixes assert-bound :: ('i,'a::lrv) i-atom  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state
assumes assert-bound-nolhs-tableau-id:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla s; \Diamond s \rrbracket$ 
 $\implies \mathcal{T} (\text{assert-bound } a s) = \mathcal{T} s$ 
assumes assert-bound-nolhs-sat:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla s; \Diamond s \rrbracket \implies$ 
 $\neg \mathcal{U} (\text{assert-bound } a s) \implies \models_{nolhs} (\text{assert-bound } a s) \wedge \Diamond (\text{assert-bound } a s)$ 
assumes assert-bound-nolhs-atoms-equiv-bounds:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla$ 
 $s; \Diamond s \rrbracket \implies$ 
 $flat ats \doteqdot \mathcal{B} s \implies flat(ats \cup \{a\}) \doteqdot \mathcal{B} (\text{assert-bound } a s)$ 
assumes assert-bound-nolhs-atoms-imply-bounds-index:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla s; \Diamond s \rrbracket \implies$ 
 $ats \models_i \mathcal{BI} s \implies insert a ats \models_i \mathcal{BI} (\text{assert-bound } a s)$ 
assumes assert-bound-nolhs-unsat:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla s; \Diamond s \rrbracket \implies$ 
 $index\text{-valid } as s \implies \mathcal{U} (\text{assert-bound } a s) \implies minimal\text{-unsat-state-core } (\text{assert-bound } a s)$ 
assumes assert-bound-nolhs-tableau-valuated:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla s;$ 
 $\Diamond s \rrbracket \implies$ 
 $\nabla (\text{assert-bound } a s)$ 
assumes assert-bound-nolhs-index-valid:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Delta(\mathcal{T} s); \nabla s; \Diamond s \rrbracket \implies$ 
 $index\text{-valid } as s \implies index\text{-valid } (insert a as) (\text{assert-bound } a s)$ 

sublocale AssertBoundNoLhs < AssertBound
by unfold-locales
 $((metis satisfies-satisfies-no-lhs satisfies-consistent$ 

```

```

assert-bound-nolhs-tableau-id assert-bound-nolhs-sat assert-bound-nolhs-atoms-equiv-bounds
assert-bound-nolhs-index-valid assert-bound-nolhs-atoms-imply-bounds-index
assert-bound-nolhs-unsat assert-bound-nolhs-tableau-valuated)+)

```

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 $\models_{nolhs} s$, $\Diamond s$, $\Delta(\mathcal{T} s)$, and ∇s hold.

```

locale Check = fixes check::('i,'a::lrv) state ⇒ ('i,'a) state
assumes

```

— If *check* is called from an inconsistent state, the state is unchanged.

```

check-unsat-id: U s ⇒ check s = s and

```

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

```

check-tableau: [¬ U s; ⊨_{nolhs} s; □ s; △(T s); ▽ s] ⇒
let s' = check s in ((v::'a valuation) ⊨_t T s ↔ v ⊨_t T s') ∧ △(T s') ∧ ▽ s'
∧ ⊨_{nolhs} s' ∧ □ s' and

```

— The bounds remain unchanged.

```

check-bounds-id: [¬ U s; ⊨_{nolhs} s; □ s; △(T s); ▽ s] ⇒ B_i (check s) = 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: [¬ U s; ⊨_{nolhs} s; □ s; △(T s); ▽ s] ⇒ ¬ U (check s) ⇒ ⊨ (check s) and

```

```

check-unsat: [¬ U s; ⊨_{nolhs} s; □ s; △(T s); ▽ s] ⇒ U (check s) ⇒ minimal-unsat-state-core (check s)

```

begin

```

lemma check-tableau-equiv: [¬ U s; ⊨_{nolhs} s; □ s; △(T s); ▽ s] ⇒
(v::'a valuation) ⊨_t T s ↔ v ⊨_t T (check s)
using check-tableau
by (simp add: Let-def)

```

```

lemma check-tableau-index-valid: assumes ¬ U s ⊨_{nolhs} s □ s △(T s) ▽ s
shows index-valid as (check s) = index-valid as s
unfolding index-valid-def using check-bounds-id[OF assms]
by (auto simp: indexu-def indexl-def boundsl-def boundsu-def)

```

```

lemma check-tableau-normalized:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Diamond s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies \Delta (\mathcal{T} (check s))$ 
  using check-tableau
  by (simp add: Let-def)

lemma check-bounds-consistent: assumes  $\neg \mathcal{U} s \models_{nolhs} s \Diamond s \Delta (\mathcal{T} s) \nabla s$ 
  shows  $\Diamond (check s)$ 
  using check-bounds-id[OF assms] assms(3)
  unfolding Let-def bounds-consistent-def boundsl-def boundsu-def
  by (metis Pair-inject)

lemma check-tableau-valuated:  $\llbracket \neg \mathcal{U} s; \models_{nolhs} s; \Diamond s; \Delta (\mathcal{T} s); \nabla s \rrbracket \implies \nabla (check s)$ 
  using check-tableau
  by (simp add: Let-def)

lemma check-indices-state: assumes  $\neg \mathcal{U} s \implies \models_{nolhs} s \neg \mathcal{U} s \implies \Diamond s \neg \mathcal{U} s$ 
 $\implies \Delta (\mathcal{T} s) \neg \mathcal{U} s \implies \nabla s$ 
  shows indices-state (check s) = indices-state s
  using check-bounds-id[OF - assms] check-unsat-id[of s]
  unfolding indices-state-def by (cases  $\mathcal{U} s$ , auto)

lemma check-distinct-indices-state: assumes  $\neg \mathcal{U} s \implies \models_{nolhs} s \neg \mathcal{U} s \implies \Diamond s$ 
 $\neg \mathcal{U} s \implies \Delta (\mathcal{T} s) \neg \mathcal{U} s \implies \nabla s$ 
  shows distinct-indices-state (check s) = distinct-indices-state s
  using check-bounds-id[OF - assms] check-unsat-id[of s]
  unfolding distinct-indices-state-def by (cases  $\mathcal{U} s$ , auto)

end

locale Assert' = AssertBound assert-bound + Check check for
  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 :: ('i, 'a) i-atom  $\Rightarrow$  ('i, 'a) state  $\Rightarrow$  ('i, 'a) state where
  assert a s  $\equiv$  check (assert-bound a s)

lemma Assert'Precond:
  assumes  $\neg \mathcal{U} s \models s \Delta (\mathcal{T} s) \nabla s$ 
  shows
     $\Delta (\mathcal{T} (\text{assert-bound } a s))$ 
     $\neg \mathcal{U} (\text{assert-bound } a s) \implies \models_{nolhs} (\text{assert-bound } a s) \wedge \Diamond (\text{assert-bound } a s)$ 
     $\nabla (\text{assert-bound } a s)$ 
  using assms
  using assert-bound-tableau-id assert-bound-sat assert-bound-tableau-valuated
  by (auto simp add: satisfies-bounds-consistent Let-def)

```

```

end

sublocale Assert' < Assert assert
proof
fix s::('i,'a) state and v::'a valuation and a::('i,'a) i-atom
assume *:  $\neg \mathcal{U} s \models s \triangle (\mathcal{T} s) \nabla s$ 
have  $\triangle (\mathcal{T} (\text{assert } a s))$ 
using check-tableau-normalized[of assert-bound a s] check-unsat-id[of assert-bound a s] *
using assert-bound-sat[of s a] Assert'Precond[of s a]
by (force simp add: assert-def)
moreover
have  $v \models_t \mathcal{T} s = v \models_t \mathcal{T} (\text{assert } a s)$ 
using check-tableau-equiv[of assert-bound a s v] *
using check-unsat-id[of assert-bound a s]
by (force simp add: assert-def Assert'Precond assert-bound-sat assert-bound-tableau-id)
moreover
have  $\nabla (\text{assert } a s)$ 
using assert-bound-tableau-valuated[of s a] *
using check-tableau-valuated[of assert-bound a s]
using check-unsat-id[of assert-bound a s]
by (cases  $\mathcal{U} (\text{assert-bound } a s)$ ) (auto simp add: Assert'Precond assert-def)
ultimately
show let  $s' = \text{assert } a s$  in  $(v \models_t \mathcal{T} s = v \models_t \mathcal{T} s') \wedge \triangle (\mathcal{T} s') \wedge \nabla s'$ 
by (simp add: Let-def)
next
fix s::('i,'a) state and a::('i,'a) i-atom
assume  $\neg \mathcal{U} s \models s \triangle (\mathcal{T} s) \nabla s$ 
then show  $\neg \mathcal{U} (\text{assert } a s) \implies \models (\text{assert } a s)$ 
unfolding assert-def
using check-unsat-id[of assert-bound a s]
using check-sat[of assert-bound a s]
by (force simp add: Assert'Precond)
next
fix s::('i,'a) state and a::('i,'a) i-atom and ats::('i,'a) i-atom set
assume  $\neg \mathcal{U} s \models s \triangle (\mathcal{T} s) \nabla s$ 
then show flat ats  $\doteq \mathcal{B} s \implies \text{flat } (\text{ats} \cup \{a\}) \doteq \mathcal{B} (\text{assert } a s)$ 
using assert-bound-atoms-equiv-bounds
using check-unsat-id[of assert-bound a s] check-bounds-id
by (cases  $\mathcal{U} (\text{assert-bound } a s)$ ) (auto simp add: Assert'Precond assert-def
simp: indexl-def indexu-def boundsl-def boundsu-def)
next
fix s::('i,'a) state and a::('i,'a) i-atom and ats
assume *:  $\neg \mathcal{U} s \models s \triangle (\mathcal{T} s) \nabla s \mathcal{U} (\text{assert } a s) \text{ index-valid } \text{ats } s$ 
show minimal-unsat-state-core (assert a s)
proof (cases  $\mathcal{U} (\text{assert-bound } a s)$ )
case True
then show ?thesis

```

```

unfolding assert-def
using * assert-bound-unsat check-tableau-equiv[of assert-bound a s] check-bounds-id
  using check-unsat-id[of assert-bound a s]
  by (auto simp add: Assert'Precond satisfies-state-def Let-def)
next
  case False
  then show ?thesis
    unfolding assert-def
    using * assert-bound-sat[of s a] check-unsat Assert'Precond
    by (metis assert-def)
  qed
next
  fix ats
  fix s::('i,'a) state and a::('i,'a) i-atom
  assume *: index-valid ats s
  and **:  $\neg \mathcal{U} s \models s \Delta (\mathcal{T} s) \nabla s$ 
  have *: index-valid (insert a ats) (assert-bound a s)
    using assert-bound-index-valid[OF ** *].
  show index-valid (insert a ats) (assert a s)
  proof (cases  $\mathcal{U}$  (assert-bound a s))
    case True
    show ?thesis unfolding assert-def check-unsat-id[OF True] using * .
  next
    case False
    show ?thesis unfolding assert-def using Assert'Precond[OF **, of a] False *
      by (subst check-tableau-index-valid[OF False], auto)
  qed
next
  fix s ats a
  let ?s = assert-bound a s
  assume *:  $\neg \mathcal{U} s \models s \Delta (\mathcal{T} s) \nabla s$  ats  $\models_i \mathcal{BI} s$ 
  from assert-bound-atoms-imply-bounds-index[OF this, of a]
  have as: insert a ats  $\models_i \mathcal{BI}$  (assert-bound a s) by auto
  show insert a ats  $\models_i \mathcal{BI}$  (assert a s)
  proof (cases  $\mathcal{U}$  ?s)
    case True
    from check-unsat-id[OF True] as show ?thesis unfolding assert-def by auto
  next
    case False
    from assert-bound-tableau-id[OF *(1-4)] *
    have t:  $\Delta (\mathcal{T} ?s)$  by simp
    from assert-bound-tableau-valuated[OF *(1-4)]
    have v:  $\nabla ?s$ .
    from assert-bound-sat[OF *(1-4) refl False]
    have  $\models_{\text{nolhs}} ?s \diamondsuit ?s$  by auto
    from check-bounds-id[OF False this t v] as
    show ?thesis unfolding assert-def
      by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)
  qed

```

qed

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

```
lemma AssertAllState''-precond':
  assumes  $\Delta (\mathcal{T} s) \nabla s \neg \mathcal{U} s \rightarrow \models_{nolhs} s \wedge \Diamond s$ 
  shows let s' = assert-bound-loop ats s in
     $\Delta (\mathcal{T} s') \wedge \nabla s' \wedge (\neg \mathcal{U} s' \rightarrow \models_{nolhs} s' \wedge \Diamond s')$ 
  using assms
  using assert-bound-nolhs-tableau-id assert-bound-nolhs-sat assert-bound-nolhs-tableau-valuated
  by (induct ats rule: rev-induct) (auto simp add: assert-bound-loop-def Let-def)

lemma AssertAllState''-precond':
  assumes  $\Delta t$ 
  shows let s' = assert-bound-loop ats (init t) in
     $\Delta (\mathcal{T} s') \wedge \nabla s' \wedge (\neg \mathcal{U} s' \rightarrow \models_{nolhs} s' \wedge \Diamond s')$ 
  using assms
  using AssertAllState''-precond'[of init t ats]
  by (simp add: Let-def init-tableau-id init-unsat-flag init-satisfies satisfies-consistent
    satisfies-satisfies-no-lhs init-tableau-valuated)

lemma AssertAllState''Induct:
  assumes
```

```

 $\triangle t$ 
 $P \{\} (\text{init } t)$ 
 $\wedge \text{as } bs. \text{as} \subseteq bs \implies P \text{as } t \implies P \text{bs } t$ 
 $\wedge \forall s a \text{ats}. \llbracket \neg \mathcal{U} s; \langle \mathcal{V} s \rangle \models_t \mathcal{T} s; \models_{\text{nolhs}} s; \triangle (\mathcal{T} s); \nabla s; \diamond s; P (\text{set ats}) s; \text{index-valid} (\text{set ats}) s \rrbracket$ 
 $\implies P (\text{insert } a (\text{set ats})) (\text{assert-bound } a s)$ 
shows  $P (\text{set ats}) (\text{assert-bound-loop ats} (\text{init } t))$ 
proof -
have  $P (\text{set ats}) (\text{assert-bound-loop ats} (\text{init } t)) \wedge \text{index-valid} (\text{set ats}) (\text{assert-bound-loop ats} (\text{init } t))$ 
proof (induct ats rule: rev-induct)
case Nil
then show ?case
unfolding assert-bound-loop-def
using assms(2) init-index-valid
by simp
next
case (snoc a as')
let ?s = assert-bound-loop as' (init t)
from snoc index-valid-mono[of - set (a # as') assert-bound-loop as' (init t)]
have index: index-valid (set (a # as')) (assert-bound-loop as' (init t))
by auto
from snoc assms(3)[of set as' set (a # as')]
have P: P (set (a # as')) (assert-bound-loop as' (init t)) by auto
show ?case
proof (cases U ?s)
case True
then show ?thesis
using P index
unfolding assert-bound-loop-def
by simp
next
case False
have id': set (as' @ [a]) = insert a (set as') by simp
have id: assert-bound-loop (as' @ [a]) (init t) =
assert-bound a (assert-bound-loop as' (init t))
using False unfolding assert-bound-loop-def by auto
from snoc have index: index-valid (set as') ?s by simp
show ?thesis unfolding id unfolding id' using False snoc AssertAll-
State"-precond[OF assms(1)]
by (intro conjI assert-bound-nolhs-index-valid index assms(4); (force simp:
Let-def curr-val-satisfies-no-lhs-def) ?)
qed
qed
then show ?thesis ..
qed

lemma AssertAllState"-tableau-id:
 $\triangle t \implies \mathcal{T} (\text{assert-bound-loop ats} (\text{init } t)) = \mathcal{T} (\text{init } t)$ 

```

```

by (rule AssertAllState''Induct) (auto simp add: init-tableau-id assert-bound-nolhs-tableau-id)

lemma AssertAllState''-sat:
 $\Delta t \implies$ 
 $\neg \mathcal{U}(\text{assert-bound-loop ats } (\text{init } t)) \longrightarrow \models_{\text{nolhs}}(\text{assert-bound-loop ats } (\text{init } t))$ 
 $\wedge \Diamond(\text{assert-bound-loop ats } (\text{init } t))$ 
by (rule AssertAllState''Induct) (auto simp add: init-unsat-flag init-satisfies satisfies-consistent satisfies-satisfies-no-lhs assert-bound-nolhs-sat)

lemma AssertAllState''-unsat:
 $\Delta t \implies \mathcal{U}(\text{assert-bound-loop ats } (\text{init } t)) \longrightarrow \text{minimal-unsat-state-core } (\text{assert-bound-loop ats } (\text{init } t))$ 
by (rule AssertAllState''Induct)
    (auto simp add: init-tableau-id assert-bound-nolhs-unsat init-unsat-flag)

lemma AssertAllState''-sat-atoms-equiv-bounds:
 $\Delta t \implies \neg \mathcal{U}(\text{assert-bound-loop ats } (\text{init } t)) \longrightarrow \text{flat } (\text{set ats}) \doteq \mathcal{B}(\text{assert-bound-loop ats } (\text{init } t))$ 
using AssertAllState''-precond
using assert-bound-nolhs-atoms-equiv-bounds init-atoms-equiv-bounds
by (induct ats rule: rev-induct) (auto simp add: Let-def assert-bound-loop-def)

lemma AssertAllState''-atoms-implies-bounds-index:
assumes  $\Delta t$ 
shows set ats  $\models_i \mathcal{BI}$  ( $\text{assert-bound-loop ats } (\text{init } t)$ )
proof (induct ats rule: rev-induct)
case Nil
then show ?case
unfolding assert-bound-loop-def
using init-atoms-implies-bounds-index assms
by simp
next
case (snoc a ats')
let ?s = assert-bound-loop ats' ( $\text{init } t$ )
show ?case
proof (cases  $\mathcal{U} ?s$ )
case True
then show ?thesis
using snoc atoms-implies-bounds-index-mono[of set ats' set (ats' @ [a])]
unfolding assert-bound-loop-def
by auto
next
case False
then have id: assert-bound-loop (ats' @ [a]) ( $\text{init } t$ ) = assert-bound a ?s
unfolding assert-bound-loop-def by auto
from snoc have ats: set ats'  $\models_i \mathcal{BI} ?s$  by auto
from AssertAllState''-precond[of t ats', OF assms, unfolded Let-def] False
have *:  $\models_{\text{nolhs}} ?s \Delta (\mathcal{T} ?s) \nabla ?s \Diamond ?s$  by auto
show ?thesis unfolding id using assert-bound-nolhs-atoms-implies-bounds-index[OF

```

```

False * ats, of a] by auto
qed
qed

lemma AssertAllState"-index-valid:
   $\triangle t \implies \text{index-valid}(\text{set ats}) (\text{assert-bound-loop ats} (\text{init } t))$ 
  by (rule AssertAllState"Induct, auto simp: init-index-valid index-valid-mono assert-bound-nolhs-index-valid)

end

sublocale AssertAllState" < AssertAllState assert-all-state
proof
  fix  $v::'a$  valuation and  $t$   $\text{ats} = s'$ 
  assume  $*: \triangle t$  and assert-all-state  $t$   $\text{ats} = s'$ 
  then have  $s': s' = \text{assert-all-state} t \text{ ats}$  by simp
  let  $?s' = \text{assert-bound-loop ats} (\text{init } t)$ 
  show  $v \models_t t = v \models_t \mathcal{T} s'$ 
    unfolding assert-all-state-def  $s'$ 
    using * check-tableau-equiv[of ?s' v] AssertAllState"-tableau-id[of t ats] init-tableau-id[of t]
    using AssertAllState"-sat[of t ats] check-unsat-id[of ?s']
    using AssertAllState"-precond[of t ats]
    by force

  show  $\neg \mathcal{U} s' \implies \models s'$ 
    unfolding assert-all-state-def  $s'$ 
    using * AssertAllState"-precond[of t ats]
    using check-sat check-unsat-id
    by (force simp add: Let-def)

  show  $\mathcal{U} s' \implies \text{minimal-unsat-state-core } s'$ 
    using * check-unsat check-unsat-id[of ?s'] check-bounds-id
    using AssertAllState"-unsat[of t ats] AssertAllState"-precond[of t ats]  $s'$ 
    by (force simp add: Let-def satisfies-state-def)

  show  $\neg \mathcal{U} s' \implies \text{flat}(\text{set ats}) \doteq \mathcal{B} s'$ 
    unfolding assert-all-state-def  $s'$ 
    using * AssertAllState"-precond[of t ats]
    using check-bounds-id[of ?s'] check-unsat-id[of ?s']
    using AssertAllState"-sat-atoms-equiv-bounds[of t ats]
    by (force simp add: Let-def simp: indexl-def indexu-def boundsl-def boundsu-def)

  show  $\mathcal{U} s' \implies \text{set ats} \models_i \mathcal{BI} s'$ 
    unfolding assert-all-state-def  $s'$ 
    using * AssertAllState"-precond[of t ats]
    unfolding Let-def
    using check-bounds-id[of ?s']
    using AssertAllState"-atoms-imply-bounds-index[of t ats]

```

```

using check-unsat-id[of ?s']
by (cases  $\mathcal{U}$  ?s') (auto simp add: Let-def simp: indexl-def indexu-def boundsl-def
boundsu-def)

show index-valid (set ats) s'
unfolding assert-all-state-def s'
using * AssertAllState''-precond[of t ats] AssertAllState''-index-valid[OF *, of
ats]
unfolding Let-def
using check-tableau-index-valid[of ?s']
using check-unsat-id[of ?s']
by (cases  $\mathcal{U}$  ?s', auto)

show indices-state  $s' \subseteq fst$  ‘ set ats
by (intro index-valid-indices-state, fact)
qed

```

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 update::var  $\Rightarrow$  'a::lrv  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state
assumes
  — Tableau, bounds, and the unsatisfiability flag are preserved.

```

```

update-id:  $\llbracket \Delta(\mathcal{T} s); \nabla s; x \notin lvars(\mathcal{T} s) \rrbracket \implies$ 
  let  $s' = update x c s$  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 valued.

```

update-tableau-valuated:  $\llbracket \Delta(\mathcal{T} s); \nabla s; x \notin lvars(\mathcal{T} s) \rrbracket \implies \nabla (update x v s)$ 
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-validation-nonlhs:  $\llbracket \Delta(\mathcal{T} s); \nabla s; x \notin lvars(\mathcal{T} s) \rrbracket \implies x' \notin lvars(\mathcal{T} s) \rightarrow$ 
  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:  $\llbracket \Delta(\mathcal{T} s); \nabla s; x \notin lvars(\mathcal{T} s) \rrbracket \implies \langle \mathcal{V} s \rangle \models_t \mathcal{T} s \rightarrow$ 
 $\langle \mathcal{V}(update x c s) \rangle \models_t \mathcal{T} s$ 

```

```

begin
lemma update-bounds-id:
  assumes  $\Delta(\mathcal{T}s) \nabla s x \notin lvars(\mathcal{T}s)$ 
  shows  $\mathcal{B}_i(update x c s) = \mathcal{B}_i s$ 
     $\mathcal{BI}(update x c s) = \mathcal{BI} s$ 
     $\mathcal{Bl}(update x c s) = \mathcal{Bl} s$ 
     $\mathcal{Bu}(update x c s) = \mathcal{Bu} s$ 
  using update-id assms
  by (auto simp add: Let-def simp: indexl-def indexu-def boundsl-def boundsu-def)

lemma update-indices-state-id:
  assumes  $\Delta(\mathcal{T}s) \nabla s x \notin lvars(\mathcal{T}s)$ 
  shows indices-state (update x c s) = indices-state s
  using update-bounds-id[OF assms] unfolding indices-state-def by auto

lemma update-tableau-id:  $\llbracket \Delta(\mathcal{T}s); \nabla s; x \notin lvars(\mathcal{T}s) \rrbracket \implies \mathcal{T}(update x c s) = \mathcal{T}s$ 
  using update-id
  by (auto simp add: Let-def)

lemma update-unsat-id:  $\llbracket \Delta(\mathcal{T}s); \nabla s; x \notin lvars(\mathcal{T}s) \rrbracket \implies \mathcal{U}(update x c s) = \mathcal{U}s$ 
  using update-id
  by (auto simp add: Let-def)

lemma update-unsat-core-id:  $\llbracket \Delta(\mathcal{T}s); \nabla s; x \notin lvars(\mathcal{T}s) \rrbracket \implies \mathcal{U}_c(update x c s) = \mathcal{U}_c s$ 
  using update-id
  by (auto simp add: Let-def)

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

fun assert-bound :: ('i,'a::lrv) i-atom  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('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  $\llbracket \geq_{ub} (lt dir) c ((UB dir) s x) \rrbracket \implies P s$ 
  assumes  $\llbracket \neg (\geq_{ub} (lt dir) c ((UB dir) s x)); \leq_{lb} (lt dir) c ((LB dir) s x) \rrbracket \implies$ 

```

```

 $P \text{ (set-unsat } [i, ((LI \text{ dir}) s x)] \text{ (updateBI (UBI-upd dir) } i x c s))$ 
assumes  $\llbracket x \notin \text{lvars}(\mathcal{T} s); (lt \text{ dir}) c (\langle \mathcal{V} s \rangle x); \neg (\sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x));$ 
 $\neg (\triangleleft_{lb} (lt \text{ dir}) c ((LB \text{ dir}) s x)) \rrbracket \implies$ 
 $P \text{ (update } x c \text{ (updateBI (UBI-upd dir) } i x c s))$ 
assumes  $\llbracket \neg (\sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x)); \neg (\triangleleft_{lb} (lt \text{ dir}) c ((LB \text{ dir}) s x)); x$ 
 $\in \text{lvars}(\mathcal{T} s) \rrbracket \implies$ 
 $P \text{ (updateBI (UBI-upd dir) } i x c s)$ 
assumes  $\llbracket \neg (\sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x)); \neg (\triangleleft_{lb} (lt \text{ dir}) c ((LB \text{ dir}) s x)); \neg$ 
 $((lt \text{ dir}) c (\langle \mathcal{V} s \rangle x)) \rrbracket \implies$ 
 $P \text{ (updateBI (UBI-upd dir) } i x c s)$ 
assumes  $dir = Positive \vee dir = Negative$ 
shows  $P \text{ (assert-bound' } dir i x c s)$ 
proof  $(\text{cases } \sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x))$ 
case True
then show ?thesis
using assms(1)
by simp
next
case False
show ?thesis
proof  $(\text{cases } \triangleleft_{lb} (lt \text{ dir}) c ((LB \text{ dir}) s x))$ 
case True
then show ?thesis
using  $\neg \sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x)$ 
using assms(2)
by simp
next
case False
let  $?s = \text{updateBI (UBI-upd dir) } i x c s$ 
show ?thesis
proof  $(\text{cases } x \notin \text{lvars}(\mathcal{T} ?s) \wedge (lt \text{ dir}) c (\langle \mathcal{V} s \rangle x))$ 
case True
then show ?thesis
using  $\neg \sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x) \wedge \neg \triangleleft_{lb} (lt \text{ dir}) c ((LB \text{ dir}) s x)$ 
using assms(3) assms(6)
by auto
next
case False
then have  $x \in \text{lvars}(\mathcal{T} ?s) \vee \neg ((lt \text{ dir}) c (\langle \mathcal{V} s \rangle x))$ 
by simp
then show ?thesis
proof
assume  $x \in \text{lvars}(\mathcal{T} ?s)$ 
then show ?thesis
using  $\neg \sqsupseteq_{ub} (lt \text{ dir}) c ((UB \text{ dir}) s x) \wedge \neg \triangleleft_{lb} (lt \text{ dir}) c ((LB \text{ dir}) s x)$ 
using assms(4) assms(6)
by auto
next
assume  $\neg (lt \text{ dir}) c (\langle \mathcal{V} s \rangle x)$ 

```

then show ?thesis
using $\neg \triangleright_{ub} (lt\ dir) c ((UB\ dir)\ s\ x)$; $\neg \triangleleft_{lb} (lt\ dir) c ((LB\ dir)\ s\ x)$
using assms(5) assms(6)
by simp
qed
qed
qed
qed

lemma assert-bound-cases:
assumes $\bigwedge c\ x\ dir$.
 $\llbracket dir = Positive \vee dir = Negative;$
 $a = LE\ dir\ x\ c;$
 $\triangleright_{ub} (lt\ dir) c ((UB\ dir)\ s\ x)$
 $\rrbracket \implies P' (lt\ dir) (UBI\ dir) (LBI\ dir) (UB\ dir) (LB\ dir) (UBI-upd\ dir) (UI\ dir)$
 $(LI\ dir) (LE\ dir) (GE\ dir)\ s$
assumes $\bigwedge c\ x\ dir$.
 $\llbracket dir = Positive \vee dir = Negative;$
 $a = LE\ dir\ x\ c;$
 $\neg \triangleright_{ub} (lt\ dir) c ((UB\ dir)\ s\ x); \triangleleft_{lb} (lt\ dir) c ((LB\ dir)\ s\ x)$
 $\rrbracket \implies P' (lt\ dir) (UBI\ dir) (LBI\ dir) (UB\ dir) (LB\ dir) (UBI-upd\ dir) (UI\ dir)$
 $(LI\ dir) (LE\ dir) (GE\ dir)$
 $(set-unsat [i, ((LI\ dir)\ s\ x)] (updateBI (UBI-upd\ dir)\ i\ x\ c\ s))$
assumes $\bigwedge c\ x\ dir$.
 $\llbracket dir = Positive \vee dir = Negative;$
 $a = LE\ dir\ x\ c;$
 $x \notin lvars (\mathcal{T}\ s); (lt\ dir) c (\langle V\ s \rangle\ x);$
 $\neg (\triangleright_{ub} (lt\ dir) c ((UB\ dir)\ s\ x)); \neg (\triangleleft_{lb} (lt\ dir) c ((LB\ dir)\ s\ x))$
 $\rrbracket \implies P' (lt\ dir) (UBI\ dir) (LBI\ dir) (UB\ dir) (LB\ dir) (UBI-upd\ dir) (UI\ dir)$
 $(LI\ dir) (LE\ dir) (GE\ dir)$
 $((update\ x\ c\ ((updateBI (UBI-upd\ dir)\ i\ x\ c\ s))))$
assumes $\bigwedge c\ x\ dir$.
 $\llbracket dir = Positive \vee dir = Negative;$
 $a = LE\ dir\ x\ c;$
 $x \in lvars (\mathcal{T}\ s); \neg (\triangleright_{ub} (lt\ dir) c ((UB\ dir)\ s\ x));$
 $\neg (\triangleleft_{lb} (lt\ dir) c ((LB\ dir)\ s\ x))$
 $\rrbracket \implies P' (lt\ dir) (UBI\ dir) (LBI\ dir) (UB\ dir) (LB\ dir) (UBI-upd\ dir) (UI\ dir)$
 $(LI\ dir) (LE\ dir) (GE\ dir)$
 $((updateBI (UBI-upd\ dir)\ i\ x\ c\ s))$
assumes $\bigwedge c\ x\ dir$.
 $\llbracket dir = Positive \vee dir = Negative;$
 $a = LE\ dir\ x\ c;$
 $\neg (\triangleright_{ub} (lt\ dir) c ((UB\ dir)\ s\ x)); \neg (\triangleleft_{lb} (lt\ dir) c ((LB\ dir)\ s\ x));$
 $\neg ((lt\ dir) c (\langle V\ s \rangle\ x))$
 $\rrbracket \implies$

```

 $P' (lt\ dir) (UBI\ dir) (LBI\ dir) (UB\ dir) (LB\ dir) (UBI-upd\ dir) (UI\ dir)$ 
 $(LI\ dir) (LE\ dir) (GE\ dir)$ 
 $((update\mathcal{B}\mathcal{I}\ (UBI-upd\ dir)\ i\ x\ c\ s))$ 

assumes  $\wedge\ s. P\ s = P' (>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u \mathcal{B}_{il}$ -update  $\mathcal{I}_l \mathcal{I}_u Geq Leq s$ 
assumes  $\wedge\ s. P\ s = P' (<) \mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_l \mathcal{B}_{iu}$ -update  $\mathcal{I}_u \mathcal{I}_l Leq Geq s$ 
shows  $P$  (assert-bound  $(i,a)$   $s$ )
proof (cases  $a$ )
  case ( $Leq\ x\ c$ )
    then show ?thesis
      apply (simp del: assert-bound'-def)
      apply (rule assert-bound'-cases, simp-all)
      using assms(1)[of Positive  $x\ c$ ]
      using assms(2)[of Positive  $x\ c$ ]
      using assms(3)[of Positive  $x\ c$ ]
      using assms(4)[of Positive  $x\ c$ ]
      using assms(5)[of Positive  $x\ c$ ]
      using assms(7)
      by auto
  next
    case ( $Geq\ x\ c$ )
    then show ?thesis
      apply (simp del: assert-bound'-def)
      apply (rule assert-bound'-cases)
      using assms(1)[of Negative  $x\ c$ ]
      using assms(2)[of Negative  $x\ c$ ]
      using assms(3)[of Negative  $x\ c$ ]
      using assms(4)[of Negative  $x\ c$ ]
      using assms(5)[of Negative  $x\ c$ ]
      using assms(6)
      by auto
qed
end

```

lemma set-unsat-bounds-id: \mathcal{B} (set-unsat $I\ s$) = $\mathcal{B}\ s$
unfolding bounds-l-def bounds-u-def **by** auto

lemma decrease-ub-satisfied-inverse:
assumes $lt: \triangleleft_{ub} (lt\ dir)\ c\ (UB\ dir\ s\ x)$ **and** $dir: dir = Positive \vee dir = Negative$
assumes $v: v \models_b \mathcal{B}$ (update $\mathcal{B}\mathcal{I}$ (UBI-upd dir) $i\ x\ c\ s$)
shows $v \models_b \mathcal{B}\ s$
unfolding satisfies-bounds.simps
proof
 fix x'
 show in-bounds $x'\ v$ ($\mathcal{B}\ s$)
proof (cases $x = x'$)
 case False
 then show ?thesis

```

using v dir
unfolding satisfies-bounds.simps
by (auto split: if-splits simp: boundsl-def boundsu-def)
next
  case True
  then show ?thesis
    using v dir
    unfolding satisfies-bounds.simps
    using lt
    by (erule-tac x=x' in alle)
      (auto simp add: lt-ubound-def[THEN sym] gt-lbound-def[THEN sym] compare-strict-nonstrict
        boundsl-def boundsu-def)
  qed
qed

lemma atoms-equiv-bounds-extend:
  fixes x c dir
  assumes dir = Positive ∨ dir = Negative ∨ ⊤_ub (lt dir) c (UB dir s x) ats ≡ B s
  shows (ats ∪ {LE dir x c}) ≡ B (updateBIZ (UBI-upd dir) i x c s)
  unfolding atoms-equiv-bounds.simps
proof
  fix v
  let ?s = updateBIZ (UBI-upd dir) i x c s
  show v ⊨_as (ats ∪ {LE dir x c}) = v ⊨_b B ?s
  proof
    assume v ⊨_as (ats ∪ {LE dir x c})
    then have v ⊨_as ats le (lt dir) (v x) c
      using `dir = Positive ∨ dir = Negative`
      unfolding satisfies-atom-set-def
      by auto
    show v ⊨_b B ?s
      unfolding satisfies-bounds.simps
    proof
      fix x'
      show in-bounds x' v (B ?s)
        using `v ⊨_as ats` `le (lt dir) (v x) c` `ats ≡ B s`
        using `dir = Positive ∨ dir = Negative`
        unfolding atoms-equiv-bounds.simps satisfies-bounds.simps
        by (cases x = x') (auto simp: boundsl-def boundsu-def)
    qed
  qed
next
  assume v ⊨_b B ?s
  then have v ⊨_b B s
    using `¬ ⊤_ub (lt dir) c (UB dir s x)`
    using `dir = Positive ∨ dir = Negative`
    using decrease-ub-satisfied-inverse[of dir c s x v]
    using neg-bounds-compare(1)[of c B_u s x]

```

```

using neg-bounds-compare(5)[of c  $\mathcal{B}_l$  s x]
  by (auto simp add: lt-ubound-def[THEN sym] ge-ubound-def[THEN sym]
    le-lbound-def[THEN sym] gt-lbound-def[THEN sym])
  show v  $\models_{as}$  (ats  $\cup \{LE\ dir\ x\ c\}$ )
    unfolding satisfies-atom-set-def
  proof
    fix a
    assume a  $\in$  ats  $\cup \{LE\ dir\ x\ c\}$ 
    then show v  $\models_a$  a
    proof
      assume a  $\in \{LE\ dir\ x\ c\}$ 
      then show ?thesis
        using ‹v  $\models_b$   $\mathcal{B}$  ?s›
        using ‹dir = Positive  $\vee$  dir = Negative›
        unfolding satisfies-bounds.simps
        by (auto split: if-splits simp: boundsl-def boundsu-def)
    next
      assume a  $\in$  ats
      then show ?thesis
        using ‹ats ≡  $\mathcal{B}$  s›
        using ‹v  $\models_b$   $\mathcal{B}$  s›
        unfolding atoms-equiv-bounds.simps satisfies-atom-set-def
        by auto
    qed
  qed
  qed
qed
qed

lemma bounds-updates:  $\mathcal{B}_l (\mathcal{B}_{iu}\text{-update } u\ s) = \mathcal{B}_l\ s$ 
 $\mathcal{B}_u (\mathcal{B}_{il}\text{-update } u\ s) = \mathcal{B}_u\ s$ 
 $\mathcal{B}_u (\mathcal{B}_{iu}\text{-update } (upd\ x\ (i,c))\ s) = (\mathcal{B}_u\ s)\ (x \mapsto c)$ 
 $\mathcal{B}_l (\mathcal{B}_{il}\text{-update } (upd\ x\ (i,c))\ s) = (\mathcal{B}_l\ s)\ (x \mapsto c)$ 
  by (auto simp: boundsl-def boundsu-def)

locale EqForLVar =
  fixes eq-idx-for-lvar :: tableau  $\Rightarrow$  var  $\Rightarrow$  nat
  assumes eq-idx-for-lvar:
     $\llbracket x \in lvars\ t \rrbracket \implies eq\text{-}idx\text{-}for\text{-}lvar\ t\ x < length\ t \wedge lhs\ (t ! eq\text{-}idx\text{-}for\text{-}lvar\ t\ x) = x$ 
begin
  definition eq-for-lvar :: tableau  $\Rightarrow$  var  $\Rightarrow$  eq where
    eq-for-lvar t v  $\equiv$  t ! (eq-idx-for-lvar t v)
  lemma eq-for-lvar:
     $\llbracket x \in lvars\ t \rrbracket \implies eq\text{-}for\text{-}lvar\ t\ x \in set\ t \wedge lhs\ (eq\text{-}for\text{-}lvar\ t\ x) = x$ 
    unfolding eq-for-lvar-def
    using eq-idx-for-lvar
    by auto

abbreviation rvars-of-lvar where
  rvars-of-lvar t x  $\equiv$  rvars-eq (eq-for-lvar t x)

```

```

lemma rvars-of-lvar-rvars:
  assumes  $x \in \text{lvars } t$ 
  shows  $\text{rvars-of-lvar } t \ x \subseteq \text{rvars } t$ 
  using assms eq-for-lvar[of  $x \ t$ ]
  unfolding rvars-def
  by auto

```

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  $\Rightarrow var \Rightarrow 'a \Rightarrow eq \Rightarrow 'a$ 
  — rhs-eq-val computes the value of the rhs of  $e$  in  $\langle v \rangle(x := c)$ .
  assumes rhs-eq-val:  $\langle v \rangle \models_e e \implies \text{rhs-eq-val } v \ x \ c \ e = \text{rhs } e \ \{\langle v \rangle(x := c)\}$ 

```

begin

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

```

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

```

```

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

```

```

lemma update-no-set-none:
  shows look ( $\mathcal{V} \ s$ )  $y \neq \text{None} \implies$ 
    look ( $\text{foldl } (\text{update-eq } (\mathcal{V} \ s) \ x \ v) \ (\mathcal{V} \ s) \ t$ )  $y \neq \text{None}$ 
  by (induct  $t$  rule: rev-induct, auto simp: lookup-update')

```

```

lemma update-no-left:
  assumes  $y \notin \text{lvars } t$ 
  shows look ( $\mathcal{V} \ s$ )  $y = \text{look } (\text{foldl } (\text{update-eq } (\mathcal{V} \ s) \ x \ v) \ (\mathcal{V} \ s) \ t) \ y$ 
  using assms
  by (induct  $t$  rule: rev-induct) (auto simp add: lvars-def lookup-update')

```

```

lemma update-left:
  assumes  $y \in \text{lvars } t$ 
  shows  $\exists \ eq \in \text{set } t. \text{lhs } eq = y \wedge$ 
    look ( $\text{foldl } (\text{update-eq } (\mathcal{V} \ s) \ x \ v) \ (\mathcal{V} \ s) \ t$ )  $y = \text{Some } (\text{rhs-eq-val } (\mathcal{V} \ s) \ x \ v \ eq)$ 
  using assms
  by (induct  $t$  rule: rev-induct) (auto simp add: lvars-def lookup-update')

```

```

lemma update-valuete-rhs:
  assumes  $e \in \text{set } (\mathcal{T} \ s) \triangle (\mathcal{T} \ s)$ 
  shows  $\text{rhs } e \ \{\langle \mathcal{V} \ (\text{update } x \ c \ s)\rangle\} = \text{rhs } e \ \{\langle \mathcal{V} \ s\rangle(x := c)\}$ 
  proof (rule valuate-depend, safe)

```

```

fix y
assume y ∈ rvars-eq e
then have y ∉ lvars (T s)
  using ⟨△ (T s)⟩ ⟨e ∈ set (T s)⟩
  by (auto simp add: normalized-tableau-def rvars-def)
then show ⟨V (update x c s)⟩ y = ((⟨V s⟩(x := c)) y
  using update-no-left[of y T s s x c]
  by (auto simp add: update-def map2fun-def lookup-update')
qed

end

sublocale RhsEqVal < Update update
proof
fix s::('i,'a) state and x c
show let s' = update x c s in T s' = T s ∧ Bi s' = Bi s ∧ U s' = U s ∧ Uc s' = Uc s
  by (simp add: Let-def update-def add: boundsl-def boundsu-def indexl-def indexu-def)
next
fix s::('i,'a) state and x c
assume △ (T s) ∇ s x ∉ lvars (T s)
then show ∇ (update x c s)
  using update-no-set-none[of s]
  by (simp add: Let-def update-def tableau-valuated-def lookup-update')
next
fix s::('i,'a) state and x x' c
assume △ (T s) ∇ s x ∉ lvars (T s)
show x' ∉ lvars (T s) →
  look (V (update x c s)) x' =
  (if x = x' then Some c else look (V s) x')
  using update-no-left[of x' T s s x c]
  unfolding update-def lvars-def Let-def
  by (auto simp: lookup-update')
next
fix s::('i,'a) state and x c
assume △ (T s) ∇ s x ∉ lvars (T s)
have ⟨V s⟩ ⊨t T s ⇒ ∀ e ∈ set (T s). ⟨V (update x c s)⟩ ⊨e e
proof
fix e
assume e ∈ set (T s) ⟨V s⟩ ⊨t T s
then have ⟨V s⟩ ⊨e e
  by (simp add: satisfies-tableau-def)

have x ≠ lhs e
  using ⟨x ∉ lvars (T s)⟩ ⟨e ∈ set (T s)⟩
  by (auto simp add: lvars-def)
then have ⟨V (update x c s)⟩ (lhs e) = rhs-eq-val (V s) x c e

```

```

using update-left[of lhs e  $\mathcal{T}$  s s x c]  $\langle e \in \text{set } (\mathcal{T} s) \rangle \langle \Delta (\mathcal{T} s) \rangle$ 
by (auto simp add: lvars-def lookup-update' update-def Let-def map2fun-def
normalized-tableau-def distinct-map inj-on-def)
then show  $\langle \mathcal{V} (\text{update } x c s) \rangle \models_e e$ 
using  $\langle \langle \mathcal{V} s \rangle \models_e e \rangle \langle e \in \text{set } (\mathcal{T} s) \rangle \langle x \notin \text{lvars } (\mathcal{T} s) \rangle \langle \Delta (\mathcal{T} s) \rangle$ 
using rhs-eq-val
by (simp add: satisfies-eq-def update-valuete-rhs)
qed
then show  $\langle \mathcal{V} s \rangle \models_t \mathcal{T} s \longrightarrow \langle \mathcal{V} (\text{update } x c s) \rangle \models_t \mathcal{T} s$ 
by(simp add: satisfies-tableau-def update-def)
qed

```

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:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \implies$ 
let  $s' = \text{pivot } x_i x_j s$  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:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \implies$ 
let  $s' = \text{pivot } x_i x_j s$  in  $((v::'a \text{ valuation}) \models_t \mathcal{T} s \longleftrightarrow v \models_t \mathcal{T} s') \wedge \Delta (\mathcal{T} s')$  and

```

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

```

pivot-vars':  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \implies$  let  $s' =$ 
pivot  $x_i x_j s$  in
 $\text{rvars}(\mathcal{T} s') = \text{rvars}(\mathcal{T} s) - \{x_j\} \cup \{x_i\}$   $\wedge$   $\text{lvars}(\mathcal{T} s') = \text{lvars}(\mathcal{T} s) - \{x_i\} \cup \{x_j\}$ 

```

begin

lemma pivot-bounds-id: $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \implies$

$\mathcal{B}_i (\text{pivot } x_i x_j s) = \mathcal{B}_i s$

using pivot-id

by (simp add: Let-def)

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

shows $\mathcal{BI}(\text{pivot } x_i \ x_j \ s) = \mathcal{BI} s \ \mathcal{B}(\text{pivot } x_i \ x_j \ s) = \mathcal{B} s \ \mathcal{I}(\text{pivot } x_i \ x_j \ s) = \mathcal{I} s$
using pivot-bounds-id[*OF assms*]
by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)

lemma pivot-valuation-id: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \mathcal{V}(\text{pivot } x_i \ x_j \ s) = \mathcal{V} s$
using pivot-id
by (simp add: Let-def)

lemma pivot-unsat-id: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \mathcal{U}(\text{pivot } x_i \ x_j \ s) = \mathcal{U} s$
using pivot-id
by (simp add: Let-def)

lemma pivot-unsat-core-id: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \mathcal{U}_c(\text{pivot } x_i \ x_j \ s) = \mathcal{U}_c s$
using pivot-id
by (simp add: Let-def)

lemma pivot-tableau-equiv: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies (v::'a \text{ valuation}) \models_t \mathcal{T} s = v \models_t \mathcal{T}(\text{pivot } x_i \ x_j \ s)$
using pivot-tableau
by (simp add: Let-def)

lemma pivot-tableau-normalized: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \Delta(\mathcal{T}(\text{pivot } x_i \ x_j \ s))$
using pivot-tableau
by (simp add: Let-def)

lemma pivot-rvars: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \text{rvars}(\mathcal{T}(\text{pivot } x_i \ x_j \ s)) = \text{rvars}(\mathcal{T} s) - \{x_j\} \cup \{x_i\}$
using pivot-vars'
by (simp add: Let-def)

lemma pivot-lvars: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \text{lvars}(\mathcal{T}(\text{pivot } x_i \ x_j \ s)) = \text{lvars}(\mathcal{T} s) - \{x_i\} \cup \{x_j\}$
using pivot-vars'
by (simp add: Let-def)

lemma pivot-vars:
 $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i \rrbracket \implies \text{tvars}(\mathcal{T}(\text{pivot } x_i \ x_j \ s)) = \text{tvars}(\mathcal{T} s)$
using pivot-lvars[of s x_i x_j] pivot-rvars[of s x_i x_j]
using rvars-of-lvar-rvars[of x_i $\mathcal{T} s$]
by auto

lemma
pivot-tableau-valuated: $\llbracket \Delta(\mathcal{T} s); x_i \in \text{lvars}(\mathcal{T} s); x_j \in \text{rvars-of-lvar}(\mathcal{T} s) \ x_i; \nabla$

```

s] ==>  $\nabla (pivot\ x_i\ x_j\ s)$ 
  using pivot-valuation-id pivot-vars
  by (auto simp add: tableau-valuated-def)

```

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  $\Rightarrow$  var  $\Rightarrow$  'a::lrv  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state
  assumes pivotandupdate-unsat-id:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $\mathcal{U}(\text{pivot-and-update } x_i\ x_j\ c\ s) = \mathcal{U}\ s$ 
  assumes pivotandupdate-unsat-core-id:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $\mathcal{U}_c(\text{pivot-and-update } x_i\ x_j\ c\ s) = \mathcal{U}_c\ s$ 
  assumes pivotandupdate-bounds-id:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $\mathcal{B}_i(\text{pivot-and-update } x_i\ x_j\ c\ s) = \mathcal{B}_i\ s$ 
  assumes pivotandupdate-tableau-normalized:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $\Delta(\mathcal{T}(\text{pivot-and-update } x_i\ x_j\ c\ s))$ 
  assumes pivotandupdate-tableau-equiv:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $(v::'a\ valuation) \models_t \mathcal{T}\ s \longleftrightarrow v \models_t \mathcal{T}(\text{pivot-and-update } x_i\ x_j\ c\ s)$ 
  assumes pivotandupdate-satisfies-tableau:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $\langle \mathcal{V}\ s \rangle \models_t \mathcal{T}\ s \longrightarrow \langle \mathcal{V}(\text{pivot-and-update } x_i\ x_j\ c\ s) \rangle \models_t \mathcal{T}\ s$ 
  assumes pivotandupdate-rvars:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $rvars(\mathcal{T}(\text{pivot-and-update } x_i\ x_j\ c\ s)) = rvars(\mathcal{T}\ s) - \{x_j\} \cup \{x_i\}$ 
  assumes pivotandupdate-lvars:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $lvars(\mathcal{T}(\text{pivot-and-update } x_i\ x_j\ c\ s)) = lvars(\mathcal{T}\ s) - \{x_i\} \cup \{x_j\}$ 
  assumes pivotandupdate-valuation-nonlhs:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $x \notin lvars(\mathcal{T}\ s) - \{x_i\} \cup \{x_j\} \longrightarrow \text{look}(\mathcal{V}(\text{pivot-and-update } x_i\ x_j\ c\ s))\ x = (\text{if } x = x_i \text{ then Some } c \text{ else look}(\mathcal{V}\ s)\ x)$ 
  assumes pivotandupdate-tableau-valuated:  $\llbracket \Delta(\mathcal{T}\ s); \nabla s; x_i \in lvars(\mathcal{T}\ s); x_j \in rvars-of-lvar(\mathcal{T}\ s) x_i \rrbracket \implies$ 
     $\nabla(\text{pivot-and-update } x_i\ x_j\ c\ s)$ 
begin

```

```

lemma pivotandupdate-bounds-id': assumes  $\Delta(\mathcal{T} s) \nabla s x_i \in \text{lvars } (\mathcal{T} s) x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i$ 
shows  $\mathcal{BI}(\text{pivot-and-update } x_i x_j c s) = \mathcal{BI} s$ 
 $\mathcal{B}(\text{pivot-and-update } x_i x_j c s) = \mathcal{B} s$ 
 $\mathcal{I}(\text{pivot-and-update } x_i x_j c s) = \mathcal{I} s$ 
using pivotandupdate-bounds-id[OF assms]
by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)

lemma pivotandupdate-valuation-xi:  $[\Delta(\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i] \implies \text{look } (\mathcal{V}(\text{pivot-and-update } x_i x_j c s)) x_i = \text{Some } c$ 
using pivotandupdate-valuation-nonlhs[of s x_i x_j x_i c]
using rvars-of-lvar-rvars
by (auto simp add: normalized-tableau-def)

lemma pivotandupdate-valuation-other-nolhs:  $[\Delta(\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i; x \notin \text{lvars } (\mathcal{T} s); x \neq x_j] \implies \text{look } (\mathcal{V}(\text{pivot-and-update } x_i x_j c s)) x = \text{look } (\mathcal{V} s) x$ 
using pivotandupdate-valuation-nonlhs[of s x_i x_j x c]
by auto

lemma pivotandupdate-nolhs:
 $[\Delta(\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i; \models_{\text{nolhs}} s; \Diamond s; \mathcal{B}_l s x_i = \text{Some } c \vee \mathcal{B}_u s x_i = \text{Some } c] \implies$ 
 $\models_{\text{nolhs}} (\text{pivot-and-update } x_i x_j c s)$ 
using pivotandupdate-satisfies-tableau[of s x_i x_j c]
using pivotandupdate-tableau-equiv[of s x_i x_j - c]
using pivotandupdate-valuation-xi[of s x_i x_j c]
using pivotandupdate-valuation-other-nolhs[of s x_i x_j - c]
using pivotandupdate-lvars[of s x_i x_j c]
by (auto simp add: curr-val-satisfies-no-lhs-def satisfies-bounds.simps satisfies-bounds-set.simps
      bounds-consistent-geq-lb bounds-consistent-leq-ub map2fun-def pivotandupdate-bounds-id')

lemma pivotandupdate-bounds-consistent:
assumes  $\Delta(\mathcal{T} s) \nabla s x_i \in \text{lvars } (\mathcal{T} s) x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i$ 
shows  $\Diamond(\text{pivot-and-update } x_i x_j c s) = \Diamond s$ 
using assms pivotandupdate-bounds-id'[of s x_i x_j c]
by (simp add: bounds-consistent-def)
end

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

```

```

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

proof-
  from assms have  $x_i \neq x_j$ 
  using rvars-of-lvar-rvars[of  $x_i$   $\mathcal{T} s$ ]
  by (auto simp add: normalized-tableau-def)
  then show  $\Delta(\mathcal{T}(\text{pivot } x_i x_j s)) x_i \notin \text{lvars } (\mathcal{T}(\text{pivot } x_i x_j s))$ 
  using assms
  using pivot-tableau-normalized[of  $s$   $x_i$   $x_j$ ]
  using pivot-lvars[of  $s$   $x_i$   $x_j$ ]
  by auto
qed

end

sublocale PivotUpdate < PivotAndUpdate eq-idx-for-lvar pivot-and-update
  using pivot-update-precond
  using update-unsat-id pivot-unsat-id pivot-unsat-core-id update-bounds-id pivot-bounds-id
  update-tableau-id pivot-tableau-normalized pivot-tableau-equiv update-satisfies-tableau
  pivot-valuation-id pivot-lvars pivot-rvars update-valuation-nonlhs update-valuation-nonlhs
  pivot-tableau-valuated update-tableau-valuated update-unsat-core-id
  by (unfold-locales, auto)

```

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

```

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

by (cases s , auto simp: *indices-state-def*)

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

by (cases s , auto simp: *boundsl-def* *boundsu-def* *indexl-def* *indexu-def*)

lemma satisfies-tableau-cong: **assumes** $\bigwedge x. x \in \text{tvars } t \implies v x = w x$

shows $(v \models_t t) = (w \models_t t)$

unfolding *satisfies-tableau-def* *satisfies-eq-def*

by (intro ball-cong[*OF refl*] arg-cong2[of $\dots (=)$] *valuate-depend*,
insert assms, auto simp: *lvars-def* *rvars-def*)

lemma satisfying-state-valuation-to-atom-tabl: **assumes** $J: J \subseteq \text{indices-state } s$

```

and model:  $(J, v) \models_{ise} s$ 
and invalid: index-valid as  $s$ 
and dist: distinct-indices-atoms as
shows  $(J, v) \models_{iaes} as v \models_t \mathcal{T} s$ 
unfolding i-satisfies-atom-set'.simp
proof (intro ballI)
  from model[unfolded satisfies-state-index'.simp]
  have model:  $v \models_t \mathcal{T} s (J, v) \models_{ibe} \mathcal{BI} s$  by auto
  show  $v \models_t \mathcal{T} s$  by fact
  fix  $a$ 
  assume  $a \in \text{restrict-to } J$  as
  then obtain  $i$  where  $iJ: i \in J$  and mem:  $(i, a) \in as$  by auto
  with  $J$  have  $i \in \text{indices-state } s$  by auto
  from this[unfolded indices-state-def] obtain  $x c$  where
    look: look ( $\mathcal{B}_{il} s$ )  $x = \text{Some } (i, c) \vee$  look ( $\mathcal{B}_{iu} s$ )  $x = \text{Some } (i, c)$  by auto
  with invalid[unfolded index-valid-def]
  obtain  $b$  where  $(i, b) \in as$  atom-var  $b = x$  atom-const  $b = c$  by force
  with dist[unfolded distinct-indices-atoms-def, rule-format, OF this(1) mem]
  have  $a: \text{atom-var } a = x \text{ atom-const } a = c$  by auto
  from model(2)[unfolded satisfies-bounds-index'.simp] look iJ have  $v x = c$ 
    by (auto simp: boundsu-def boundsl-def indexu-def indexl-def)
  thus  $v \models_{ae} a$  unfolding satisfies-atom'-def  $a$  .
qed

```

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

```

sublocale AssertAllState < AssertAll assert-all
proof
  fix  $t$  as  $v I$ 
  assume  $D: \Delta t$ 
  from  $D$  show assert-all  $t as = Sat v \implies \langle v \rangle \models_t t \wedge \langle v \rangle \models_{as} \text{flat (set as)}$ 
  unfolding Let-def assert-all-def
  using assert-all-state-tableau-equiv[OF D refl]
  using assert-all-state-sat[OF D refl]
  using assert-all-state-sat-atoms-equiv-bounds[OF D refl, of as]
  unfolding atoms-equiv-bounds.simps curr-val-satisfies-state-def satisfies-state-def
  satisfies-atom-set-def
  by (auto simp: Let-def split: if-splits)
  let ?s = assert-all-state  $t as$ 
  assume assert-all  $t as = \text{Unsat } I$ 
  then have  $i: I = \text{the } (\mathcal{U}_c ?s)$  and  $U: \mathcal{U} ?s$ 
  unfolding assert-all-def Let-def by (auto split: if-splits)
  from assert-all-index-valid[OF D refl, of as] have invalid: index-valid (set as) ?s

  note unsat = assert-all-state-unsat[OF D refl U, unfolded minimal-unsat-state-core-def
  unsat-state-core-def i[symmetric]]
  from unsat have set  $I \subseteq \text{indices-state } ?s$  by auto
  also have ...  $\subseteq \text{fst } ' \text{set as}$  using assert-all-state-indices[OF D refl] .
  finally have indices: set  $I \subseteq \text{fst } ' \text{set as}$  .

```

```

show minimal-unsat-core-tabl-atoms (set I) t (set as)
  unfolding minimal-unsat-core-tabl-atoms-def
  proof (intro conjI impI allI indices, clarify)
    fix v
    assume model:  $v \models_t t$  (set I,  $v \models_{ias}$  set as)
    from unsat have no-model:  $\neg ((\text{set } I, v) \models_{is} ?s)$  by auto
    from assert-all-state-unsat-atoms-equiv-bounds[OF D refl U]
    have equiv: set as  $\models_i \mathcal{BI} ?s$  by auto
    from assert-all-state-tableau-equiv[OF D refl, of v] model
    have model-t:  $v \models_t \mathcal{T} ?s$  by auto
    have model-as':  $(\text{set } I, v) \models_{ias}$  set as
      using model(2) by (auto simp: satisfies-atom-set-def)
    with equiv model-t have (set I, v)  $\models_{is} ?s$ 
    unfolding satisfies-state-index.simps atoms-imply-bounds-index.simps by simp
    with no-model show False by simp
  next
    fix J
    assume dist: distinct-indices-atoms (set as) and  $J: J \subset \text{set } I$ 
    from J unsat[unfolded subsets-sat-core-def, folded i]
    have J':  $J' \subseteq \text{indices-state} ?s$  by auto
    from index-valid-distinct-indices[OF invalid dist] J unsat[unfolded subsets-sat-core-def,
    folded i]
    obtain v where model:  $(J, v) \models_{ise} ?s$  by blast
    have  $(J, v) \models_{iaes}$  set as  $v \models_t t$ 
      using satisfying-state-validation-to-atom-tabl[OF J' model invalid dist]
      assert-all-state-tableau-equiv[OF D refl] by auto
    then show  $\exists v. v \models_t t \wedge (J, v) \models_{iaes}$  set as by blast
  qed
qed

lemma (in Update) update-to-assert-bound-no-lhs: assumes pivot: Pivot eqvar
(pivot :: var  $\Rightarrow$  var  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state)
  shows AssertBoundNoLhs assert-bound
proof
  fix s::('i,'a) state and a
  assume  $\neg \mathcal{U} s \Delta (\mathcal{T} s) \nabla s$ 
  then show  $\mathcal{T} (\text{assert-bound } a s) = \mathcal{T} s$ 
    by (cases a, cases snd a) (auto simp add: Let-def update-tableau-id tableau-valuated-def)
next
  fix s::('i,'a) state and ia and as
  assume *:  $\neg \mathcal{U} s \Delta (\mathcal{T} s) \nabla s$  and **:  $\mathcal{U} (\text{assert-bound } ia s)$ 
  and index: index-valid as s
  and consistent:  $\models_{nolhs} s \diamondsuit s$ 
  obtain i a where ia:  $ia = (i,a)$  by force
  let ?modelU =  $\lambda lt UB UI s v x c i. UB s x = \text{Some } c \rightarrow UI s x = i \rightarrow i \in$ 
  set (the ( $\mathcal{U}_c$  s))  $\rightarrow (lt (v x) c \vee v x = c)$ 
  let ?modelL =  $\lambda lt LB LI s v x c i. LB s x = \text{Some } c \rightarrow LI s x = i \rightarrow i \in$ 
  set (the ( $\mathcal{U}_c$  s))  $\rightarrow (lt c (v x) \vee c = v x)$ 
  let ?modelIU =  $\lambda I lt UB UI s v x c i. UB s x = \text{Some } c \rightarrow UI s x = i \rightarrow i$ 

```

```

 $\in I \longrightarrow (v x = c)$ 
let ?modelIL =  $\lambda I lt LB LI s v x c i. LB s x = Some c \longrightarrow LI s x = i \longrightarrow i \in I \longrightarrow (v x = c)$ 
let ?P' =  $\lambda lt UBI LBI UB LB UBI-upd UI LI LE GE s.$ 
     $\mathcal{U} s \longrightarrow (set (the (\mathcal{U}_c s)) \subseteq indices-state s \wedge \neg (\exists v. (v \models_t \mathcal{T} s \wedge (\forall x c i. ?modelU lt UB UI s v x c i) \wedge (\forall x c i. ?modelL lt LB LI s v x c i)))) \wedge (distinct-indices-state s \longrightarrow (\forall I. I \subset set (the (\mathcal{U}_c s))) \longrightarrow (\exists v. v \models_t \mathcal{T} s \wedge (\forall x c i. ?modelIU I lt UB UI s v x c i) \wedge (\forall x c i. ?modelIL I lt LB LI s v x c i))))$ 
have  $\mathcal{U}$  (assert-bound ia s)  $\longrightarrow$  (unsat-state-core (assert-bound ia s)  $\wedge$  (distinct-indices-state (assert-bound ia s)  $\longrightarrow$  subsets-sat-core (assert-bound ia s))) (is ?P (assert-bound ia s) unfolding ia
proof (rule assert-bound-cases[of - - ?P'])
    fix  $s' :: ('i,'a) state$ 
    have  $id: ((x :: 'a) < y \vee x = y) \longleftrightarrow x \leq y ((x :: 'a) > y \vee x = y) \longleftrightarrow x \geq y$  for  $x y$  by auto
        have  $id': ?P'(>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u undefined \mathcal{I}_l \mathcal{I}_u Geq Leq s' = ?P'(<) \mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_u \mathcal{B}_l undefined \mathcal{I}_u \mathcal{I}_l Leq Geq s'$ 
            by (intro arg-cong[of - -  $\lambda y. - \longrightarrow y$ ] arg-cong[of - -  $\lambda x. - \wedge x$ ],  

                intro arg-cong2[of - - - - ( $\wedge$ )] arg-cong[of - -  $\lambda y. - \longrightarrow y$ ] arg-cong[of - -  $\lambda y. - \wedge y$ ],  

                 $y. \forall x \subset set (the (\mathcal{U}_c s')). y x] ext arg-cong[of - - Not],  

                unfold id, auto)$ 
        show  $?P s' = ?P'(>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u undefined \mathcal{I}_l \mathcal{I}_u Geq Leg s'$ 
        unfold satisfies-state-def satisfies-bounds-index.simps satisfies-bounds.simps  

in-bounds.simps unsat-state-core-def satisfies-state-index.simps subsets-sat-core-def  

satisfies-state-index'.simps satisfies-bounds-index'.simps
        unfolding bound-compare''-defs id
        by ((intro arg-cong[of - -  $\lambda x. - \longrightarrow x$ ] arg-cong[of - -  $\lambda x. - \wedge x$ ],  

            intro arg-cong2[of - - - - ( $\wedge$ )] refl arg-cong[of - -  $\lambda x. - \longrightarrow x$ ] arg-cong[of - - Not]
             $arg-cong[of - - \lambda y. \forall x \subset set (the (\mathcal{U}_c s')). y x] ext; intro arg-cong[of - - Ex] ext), auto)$ 
        then show  $?P s' = ?P'(<) \mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_u \mathcal{B}_l undefined \mathcal{I}_u \mathcal{I}_l Leq Geq s'$  unfolding  $id'$ .
    next
        fix  $c :: 'a$  and  $x :: nat$  and  $dir$ 
        assume  $\triangleleft_{lb} (lt dir) c (LB dir s x)$  and  $dir: dir = Positive \vee dir = Negative$ 
        then obtain  $d$  where  $some: LB dir s x = Some d$  and  $lt: lt dir c d$ 
            by (auto simp: bound-compare'-defs split: option.splits)
            from index[unfolded index-valid-def, rule-format, of x - d]
             $some dir$  obtain  $j$  where  $ind: LI dir s x = j$  look  $(LBI dir s) x = Some (j,d)$ 
            and  $ge: (j, GE dir x d) \in as$ 
                by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)
            let ?s = set-unsat [i, ((LI dir) s x)] (updateBI (UBI-upd dir) i x c s)
            let ?ss = updateBI (UBI-upd dir) i x c s
            show  $?P' (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd dir) (UI dir) (LI dir) (LE dir) (GE dir) ?s$ 

```

```

proof (intro conjI impI allI, goal-cases)
  case 1
    thus ?case using dir ind ge lt some by (force simp: indices-state-def split:
if-splits)
  next
    case 2
    {
      fix v
      assume vU:  $\forall x c i. \text{?modelU}(\text{lt dir}) (\text{UB dir}) (\text{UI dir}) ?s v x c i$ 
      assume vL:  $\forall x c i. \text{?modelL}(\text{lt dir}) (\text{LB dir}) (\text{LI dir}) ?s v x c i$ 
      from dir have UB dir ?s x = Some c UI dir ?s x = i by (auto simp:
boundsl-def boundsu-def indexl-def indexu-def)
      from vU[rule-format, OF this] have vx-le-c: lt dir (v x) c  $\vee$  v x = c by
auto
      from dir ind some have *: LB dir ?s x = Some d LI dir ?s x = j by (auto
simp: boundsl-def boundsu-def indexl-def indexu-def)
      have d-le-vx: lt dir d (v x)  $\vee$  d = v x by (intro vL[rule-format, OF *], insert
some ind, auto)
      from dir d-le-vx vx-le-c lt
      have False by (auto simp del: Simplex.bounds-lg)
    }
    thus ?case by blast
  next
    case (3 I)
    then obtain j where I:  $I \subseteq \{j\}$  by (auto split: if-splits)
    from 3 have dist: distinct-indices-state ?ss unfolding distinct-indices-state-def
by auto
    have id1: UB dir ?s y = UB dir ?ss y LB dir ?s y = LB dir ?ss y
      UI dir ?s y = UI dir ?ss y LI dir ?s y = LI dir ?ss y
       $\mathcal{T} ?s = \mathcal{T} s$ 
      set (the ( $\mathcal{U}_c ?s$ )) = {i, LI dir s x} for y
      using dir by (auto simp: boundsu-def boundsl-def indexu-def indexl-def)
      from I have id:  $(\forall k. P1 k \rightarrow P2 k \rightarrow k \in I \rightarrow Q k) \leftrightarrow (I = \{\} \vee$ 
 $(P1 j \rightarrow P2 j \rightarrow Q j))$  for P1 P2 Q by auto
      have id2:  $(UB \text{ dir } s \text{ xa } = \text{Some } ca \rightarrow UI \text{ dir } s \text{ xa } = j \rightarrow P) = (\text{look } (UBI \text{ dir } s) \text{ xa } = \text{Some } (j, ca) \rightarrow P)$ 
         $(LB \text{ dir } s \text{ xa } = \text{Some } ca \rightarrow LI \text{ dir } s \text{ xa } = j \rightarrow P) = (\text{look } (LBI \text{ dir } s) \text{ xa } = \text{Some } (j, ca) \rightarrow P)$  for xa ca P s
        using dir by (auto simp: boundsu-def indexu-def boundsl-def indexl-def)
        have  $\exists v. v \models_t \mathcal{T} s \wedge$ 
           $(\forall xa ca ia.$ 
             $UB \text{ dir } ?ss \text{ xa } = \text{Some } ca \rightarrow UI \text{ dir } ?ss \text{ xa } = ia \rightarrow ia \in I \rightarrow v$ 
 $xa = ca) \wedge$ 
           $(\forall xa ca ia.$ 
             $LB \text{ dir } ?ss \text{ xa } = \text{Some } ca \rightarrow LI \text{ dir } ?ss \text{ xa } = ia \rightarrow ia \in I \rightarrow v$ 
 $xa = ca)$ 
          proof (cases  $\exists xa ca. \text{look } (UBI \text{ dir } ?ss) \text{ xa } = \text{Some } (j, ca) \vee \text{look } (LBI \text{ dir }$ 
?ss)  $xa = \text{Some } (j, ca)$ 
          case False

```

```

thus ?thesis unfolding id id2 using consistent unfolding curr-val-satisfies-no-lhs-def
  by (intro exI[of - <V s>], auto)
next
  case True
  from consistent have val: <V s> ⊨t T s unfolding curr-val-satisfies-no-lhs-def
by auto
  define ss where ss: ss = ?ss
  from True obtain y b where look (UBI dir ?ss) y = Some (j,b) ∨ look
(LBI dir ?ss) y = Some (j,b) by force
    then have id3: (look (LBI dir ss) yy = Some (j,bb) ∨ look (UBI dir ss) yy
= Some (j,bb)) ↔ (yy = y ∧ bb = b) for yy bb
      using distinct-indices-stateD(1)[OF dist, of y j b yy bb] using dir
      unfolding ss[symmetric]
      by (auto simp: boundsu-def boundsl-def indexu-def indexl-def)
  have ∃ v. v ⊨t T s ∧ v y = b
  proof (cases y ∈ lvars (T s))
    case False
    let ?v = <V (update y b s)>
    show ?thesis
    proof (intro exI[of - ?v] conjI)
      from update-satisfies-tableau[OF *(2,3) False] val
      show ?v ⊨t T s by simp
      from update-valuation-nonlhs[OF *(2,3) False, of y b] False
      show ?v y = b by (simp add: map2fun-def')
    qed
  next
    case True
    from *(2)[unfolded normalized-tableau-def]
    have zero: 0 ∉ rhs ` set (T s) by auto
    interpret Pivot eqlvar pivot by fact
    interpret PivotUpdate eqlvar pivot update ..
    let ?eq = eq-for-lvar (T s) y
    from eq-for-lvar[OF True] have ?eq ∈ set (T s) lhs ?eq = y by auto
    with zero have rhs: rhs ?eq ≠ 0 by force
    hence rvars-eq ?eq ≠ {} by (simp add: vars-empty-zero)
    then obtain z where z: z ∈ rvars-eq ?eq by auto
    let ?v = V (pivot-and-update y z b s)
    let ?vv = <?v>
    from pivotandupdate-valuation-xi[OF *(2,3) True z]
    have look ?v y = Some b .
    hence vv: ?vv y = b unfolding map2fun-def' by auto
    show ?thesis
    proof (intro exI[of - ?vv] conjI vv)
      show ?vv ⊨t T s using pivotandupdate-satisfies-tableau[OF *(2,3) True
z] val by auto
      qed
    qed
  qed

```

```

thus ?thesis unfolding id id2 ss[symmetric] using id3 by metis
qed
thus ?case unfolding id1 .
qed
next
fix c::'a and x::nat and dir
assume **: dir = Positive ∨ dir = Negative a = LE dir x c x ∉ lvars (T s) lt
dir c ((V s) x)
¬ ⊢ub (lt dir) c (UB dir s x) ⊢lb (lt dir) c (LB dir s x)
let ?s = updateBI (UBI-upd dir) i x c s
show ?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 ?s)
using * **
by (auto simp add: update-unsat-id tableau-valuated-def)
qed (auto simp add: * update-unsat-id tableau-valuated-def)
with ** show minimal-unsat-state-core (assert-bound ia s) by (auto simp: min-
imal-unsat-state-core-def)
next
fix s::('i,'a) state and ia
assume *: ¬ U s ⊨nolhs s ◇ s △ (T s) ▽ s
and **: ¬ U (assert-bound ia s) (is ?lhs)
obtain i a where ia: ia = (i,a) by force
have ⟨V (assert-bound ia s)⟩ ⊨t T (assert-bound ia s)
proof-
let ?P = λ lt UBI LBI UB LB UBI-upd UI LI LE GE s. ⟨V s⟩ ⊨t T s
show ?thesis unfolding ia
proof (rule assert-bound-cases[of - - ?P])
fix c x and dir :: ('i,'a) Direction
let ?s' = updateBI (UBI-upd dir) i x c s
assume x ∉ lvars (T s) (lt dir) c ((V s) x)
dir = Positive ∨ dir = Negative
then show ⟨V (update x c ?s')⟩ ⊨t T (update x c ?s')
using *
using update-satisfies-tableau[of ?s' x c] update-tableau-id
by (auto simp add: curr-val-satisfies-no-lhs-def tableau-valuated-def)
qed (insert *, auto simp add: curr-val-satisfies-no-lhs-def)
qed
moreover
have ¬ U (assert-bound ia s) → ⟨V (assert-bound ia s)⟩ ⊨b B (assert-bound ia
s) || - lvars (T (assert-bound ia s)) (is ?P (assert-bound ia s))
proof-
let ?P' = λ lt UBI LBI UB LB UB-upd UI LI LE GE s.
¬ U s → (∀ x ∈ - lvars (T s). ⊢lb lt ((V s) x) (LB s x) ∧ ⊢ub lt ((V s) x)
(UB s x))
let ?P'' = λ dir. ?P' (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd
dir) (UI dir) (LI dir) (LE dir) (GE dir)

have x: ∏ s'. ?P s' = ?P' (<) B_il B_iu B_u B_l B_iu-update I_u I_l Leq Geq s'

```

and $xx : \bigwedge s'. ?P s' = ?P' (>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u \mathcal{B}_{il}\text{-update } \mathcal{I}_l \mathcal{I}_u \text{ Geq Leq } s'$
unfolding *satisfies-bounds-set.simps in-bounds.simps bound-compare-defs*
by (auto split: option.split)

```

show ?thesis unfolding ia
proof (rule assert-bound-cases[of - - ?P'])
  fix dir :: ('i,'a) Direction
  assume dir = Positive ∨ dir = Negative
  then show ?P'' dir s
    using x[of s] xx[of s] ⊢_nolhs s
    by (auto simp add: curr-val-satisfies-no-lhs-def)
next
  fix x c and dir :: ('i,'a) Direction
  let ?s' = update $\mathcal{BI}$  (UBI-upd dir) i x c s
  assume x ∈ lvars (T s) dir = Positive ∨ dir = Negative
  then have ?P ?s'
    using ⊢_nolhs s
    by (auto simp add: satisfies-bounds-set.simps curr-val-satisfies-no-lhs-def
      boundsl-def boundsu-def indexl-def indexu-def)
  then show ?P'' dir ?s'
    using x[of ?s'] xx[of ?s'] dir = Positive ∨ dir = Negative
    by auto
next
  fix c x and dir :: ('i,'a) Direction
  let ?s' = update $\mathcal{BI}$  (UBI-upd dir) i x c s
  assume ¬ lt dir c (⟨V s⟩ x) dir = Positive ∨ dir = Negative
  then show ?P'' dir ?s'
    using ⊢_nolhs s
    by (auto simp add: satisfies-bounds-set.simps curr-val-satisfies-no-lhs-def
      simp: boundsl-def boundsu-def indexl-def indexu-def)
      (auto simp add: bound-compare-defs)
next
  fix c x and dir :: ('i,'a) Direction
  let ?s' = update x c (update $\mathcal{BI}$  (UBI-upd dir) i x c s)
  assume x ∉ lvars (T s) ¬ ⊲_lb (lt dir) c (LB dir s x)
  dir = Positive ∨ dir = Negative
  show ?P'' dir ?s'
  proof (rule impI, rule ballI)
    fix y
    assume ¬ U ?s' y ∈ - lvars (T ?s')
    show ⊲_lb (lt dir) (⟨V ?s'⟩ y) (LB dir ?s' y) ∧ ⊲_ub (lt dir) (⟨V ?s'⟩ y) (UB
      dir ?s' y)
    proof (cases x = y)
      case True
      then show ?thesis
        using ⟨x ∉ lvars (T s)⟩
        using ⟨y ∈ - lvars (T ?s')⟩
        using ⟨¬ ⊲_lb (lt dir) c (LB dir s x)⟩
        using ⟨dir = Positive ∨ dir = Negative⟩
    qed
  qed
qed

```

```

using neg-bounds-compare(7) neg-bounds-compare(3)
using *
  by (auto simp add: update-valuation-nonlhs update-tableau-id up-
date-bounds-id bound-compare"-defs map2fun-def tableau-valuated-def bounds-updates)
(force simp add: bound-compare'-defs)+

next
  case False
  then show ?thesis
    using <x ∉ lvars (T s)> <y ∈ − lvars (T ?s')>
    using <dir = Positive ∨ dir = Negative> *
      by (auto simp add: update-valuation-nonlhs update-tableau-id up-
date-bounds-id bound-compare"-defs satisfies-bounds-set.simps curr-val-satisfies-no-lhs-def
map2fun-def
      tableau-valuated-def bounds-updates)
    qed
  qed
  qed (auto simp add: x xx)
qed
moreover
have ¬ U (assert-bound ia s) —→ ◊ (assert-bound ia s) (is ?P (assert-bound ia
s))
proof—
  let ?P' = λ lt UBI LBI UB LB UBI-upd UI LI LE GE s.
  ¬ U s —→
  ( ∀ x. if LB s x = None ∨ UB s x = None then True
    else lt (the (LB s x)) (the (UB s x)) ∨ (the (LB s x) = the (UB s x)))
  let ?P'' = λ dir. ?P' (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd
dir) (UI dir) (LI dir) (LE dir) (GE dir)

  have x: ⋀ s'. ?P s' = ?P' (<) B_il B_iu B_u B_l B_iu-update I_u I_l Leq Geq s' and
    xx: ⋀ s'. ?P s' = ?P' (>) B_il B_iu B_l B_u B_il-update I_l I_u Geq Leq s'
  unfolding bounds-consistent-def
  by auto

show ?thesis unfolding ia
proof (rule assert-bound-cases[of - - ?P'])
  fix dir :: ('i,'a) Direction
  assume dir = Positive ∨ dir = Negative
  then show ?P'' dir s
    using <◊ s>
    by (auto simp add: bounds-consistent-def) (erule-tac x=x in allE, auto)+

next
  fix x c and dir :: ('i,'a) Direction
  let ?s' = update x c (updateB I (UBI-upd dir) i x c s)
  assume dir = Positive ∨ dir = Negative x ∉ lvars (T s)
  ¬ ⊇_{ub} (lt dir) c (UB dir s x) ∨ ⊆_{lb} (lt dir) c (LB dir s x)
  then show ?P'' dir ?s'
    using <◊ s> *
    unfolding bounds-consistent-def

```

```

    by (auto simp add: update-bounds-id tableau-valuated-def bounds-updates
split: if-splits)
      (force simp add: bound-compare'-defs, erule-tac x=xa in allE, simp) +
next
  fix x c and dir :: ('i,'a) Direction
  let ?s' = update $\mathcal{BI}$  (UBI-upd dir) i x c s
  assume  $\neg \triangleright_{ub} (lt\ dir)\ c\ (UB\ dir\ s\ x) \neg \triangleleft_{lb} (lt\ dir)\ c\ (LB\ dir\ s\ x)$ 
  dir = Positive  $\vee$  dir = Negative
  then have ?P'' dir ?s'
    using  $\langle \diamond \rangle s$ 
    unfolding bounds-consistent-def
    by (auto split: if-splits simp: bounds-updates)
      (force simp add: bound-compare'-defs, erule-tac x=xa in allE, simp) +
  then show ?P'' dir ?s' ?P'' dir ?s'
    by simp-all
qed (auto simp add: x xx)
qed

```

ultimately

```

show  $\models_{nolhs} (\text{assert-bound ia}\ s) \wedge \diamond (\text{assert-bound ia}\ s)$ 
using  $\langle ?lhs \rangle$ 
unfolding curr-val-satisfies-no-lhs-def
by simp
next
  fix s :: ('i,'a) state and ats and ia :: ('i,'a) i-atom
  assume  $\neg \mathcal{U}\ s \models_{nolhs} s \triangle (\mathcal{T}\ s) \nabla s$ 
  obtain i a where ia: ia = (i,a) by force
  {
    fix ats
    let ?P' =  $\lambda\ lt\ UBI\ LBI\ UB\ LB\ UB\text{-upd}\ UI\ LI\ LE\ GE\ s'.\ ats \doteq \mathcal{B}\ s \longrightarrow (ats$ 
     $\cup \{a\}) \doteq \mathcal{B}\ s'$ 
    let ?P'' =  $\lambda\ dir.\ ?P'\ (lt\ dir)\ (UB\ dir)\ (LB\ dir)\ (UBI\text{-upd}\ dir)\ (UI\ dir)\ (LI$ 
    dir) (LE dir) (GE dir)
    have ats  $\doteq \mathcal{B}\ s \longrightarrow (ats \cup \{a\}) \doteq \mathcal{B}\ (\text{assert-bound ia}\ s)$  (is ?P (assert-bound ia s))
      unfolding ia
    proof (rule assert-bound-cases[of - - ?P'])
      fix x c and dir :: ('i,'a) Direction
      assume dir = Positive  $\vee$  dir = Negative a = LE dir x c  $\triangleright_{ub} (lt\ dir)\ c\ (UB$ 
      dir s x)
      then show ?P s
      unfolding atoms-equiv-bounds.simps satisfies-atom-set-def satisfies-bounds.simps
        by auto (erule-tac x=x in allE, force simp add: bound-compare-defs) +
    next
      fix x c and dir :: ('i,'a) Direction
      let ?s' = set-unsat [i, ((LI dir) s x)] (update $\mathcal{BI}$  (UBI-upd dir) i x c s)
      assume dir = Positive  $\vee$  dir = Negative a = LE dir x c  $\neg (\triangleright_{ub} (lt\ dir)\ c$ 

```

```

(UB dir s x))
  then show ?P ?s' unfolding set-unsat-bounds-id
    using atoms-equiv-bounds-extend[of dir c s x ats i]
    by auto
next
  fix x c and dir :: ('i,'a) Direction
  let ?s' = update $\mathcal{BI}$  (UBI-upd dir) i x c s
  assume dir = Positive ∨ dir = Negative a = LE dir x c ∨ ( $\geq_{ub}$  (lt dir) c
(UB dir s x))
  then have ?P ?s'
    using atoms-equiv-bounds-extend[of dir c s x ats i]
    by auto
  then show ?P ?s' ?P ?s'
    by simp-all
next
  fix x c and dir :: ('i,'a) Direction
  let ?s = update $\mathcal{BI}$  (UBI-upd dir) i x c s
  let ?s' = update x c ?s
  assume *: dir = Positive ∨ dir = Negative a = LE dir x c ∨ ( $\geq_{ub}$  (lt dir) c
(UB dir s x)) x  $\notin$  lvars ( $\mathcal{T}$  s)
  then have  $\Delta$  ( $\mathcal{T}$  ?s)  $\nabla$  ?s x  $\notin$  lvars ( $\mathcal{T}$  ?s)
    using  $\langle \Delta$  ( $\mathcal{T}$  s)  $\rangle \models_{nolhs} s \langle \nabla$  s
    by (auto simp: tableau-valuated-def)
  from update-bounds-id[Of this, of c]
  have  $\mathcal{B}_i$  ?s' =  $\mathcal{B}_i$  ?s by blast
  then have id:  $\mathcal{B}$  ?s' =  $\mathcal{B}$  ?s unfolding bounds-l-def bounds-u-def by auto
  show ?P ?s' unfolding id a = LE dir x c
    by (intro impI atoms-equiv-bounds-extend[rule-format] *(1,3))
  qed simp-all
}
then show flat ats  $\doteqdot$   $\mathcal{B}$  s  $\implies$  flat (ats  $\cup$  {ia})  $\doteqdot$   $\mathcal{B}$  (assert-bound ia s) unfolding
ia by auto
next
  fix s :: ('i,'a) state and ats and ia :: ('i,'a) i-atom
  obtain i a where ia: ia = (i,a) by force
  assume  $\neg$   $\mathcal{U}$  s  $\models_{nolhs} s \Delta$  ( $\mathcal{T}$  s)  $\nabla$  s
  have *:  $\bigwedge$  dir x c s. dir = Positive ∨ dir = Negative  $\implies$ 
     $\nabla$  (update $\mathcal{BI}$  (UBI-upd dir) i x c s) =  $\nabla$  s
     $\bigwedge$  s y I .  $\nabla$  (set-unsat I s) =  $\nabla$  s
    by (auto simp add: tableau-valuated-def)

  show  $\nabla$  (assert-bound ia s) (is ?P (assert-bound ia s))
  proof-
    let ?P' =  $\lambda$  lt UBI LBI UB LB UB-upd UI LI LE GE s'.  $\nabla$  s'
    let ?P'' =  $\lambda$  dir. ?P' (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd
dir) (UI dir) (LI dir) (LE dir) (GE dir)
    show ?thesis unfolding ia
    proof (rule assert-bound-cases[of - - ?P'])
      fix x c and dir :: ('i,'a) Direction

```

```

let ?s' = update $\mathcal{BI}$  (UBI-upd dir) i x c s
assume dir = Positive ∨ dir = Negative
then have  $\nabla$  ?s'
  using *(1)[of dir x c s]  $\langle \nabla s \rangle$ 
  by simp
then show  $\nabla$  (set-unsat [i, ((LI dir) s x)] ?s')
  using *(2) by auto
next
fix x c and dir :: ('i,'a) Direction
assume *: x ∉ lvars ( $\mathcal{T}$  s) dir = Positive ∨ dir = Negative
let ?s = update $\mathcal{BI}$  (UBI-upd dir) i x c s
let ?s' = update x c ?s
from * show  $\nabla$  ?s'
  using  $\langle \Delta (\mathcal{T} s) \rangle \langle \nabla s \rangle$ 
  using update-tableau-valuated[of ?s x c]
  by (auto simp add: tableau-valuated-def)
qed (insert  $\langle \nabla s \rangle$  *(1), auto)
qed
next
fix s :: ('i,'a) state and as and ia :: ('i,'a) i-atom
obtain i a where ia: ia = (i,a) by force
assume *:  $\neg \mathcal{U} s \models_{nolhs} s \Delta (\mathcal{T} s) \nabla s$ 
and valid: index-valid as s
have id:  $\bigwedge$  dir x c s. dir = Positive ∨ dir = Negative  $\implies$ 
 $\nabla$  (update $\mathcal{BI}$  (UBI-upd dir) i x c s) =  $\nabla s$ 
 $\bigwedge$  s y I.  $\nabla$  (set-unsat I s) =  $\nabla s$ 
  by (auto simp add: tableau-valuated-def)
let ?I = insert (i,a) as
define I where I = ?I
from index-valid-mono[OF - valid] have valid: index-valid I s unfolding I-def
by auto
have I: (i,a) ∈ I unfolding I-def by auto
let ?P = λ s. index-valid I s
let ?P' = λ (lt :: 'a ⇒ 'a ⇒ bool)
  (UBI :: ('i,'a) state ⇒ ('i,'a) bounds-index) (LBI :: ('i,'a) state ⇒ ('i,'a) bounds-index)
  (UB :: ('i,'a) state ⇒ 'a bounds) (LB :: ('i,'a) state ⇒ 'a bounds)
  (UBI-upd :: (('i,'a) bounds-index ⇒ ('i,'a) bounds-index) ⇒ ('i,'a) state ⇒ ('i,'a) state)
  (UI :: ('i,'a) state ⇒ 'i bound-index) (LI :: ('i,'a) state ⇒ 'i bound-index)
  LE GE s'.
  ( $\forall$  x c i. look (UBI s') x = Some (i,c)  $\longrightarrow$  (i,LE (x :: var) c) ∈ I)  $\wedge$ 
  ( $\forall$  x c i. look (LBI s') x = Some (i,c)  $\longrightarrow$  (i,GE (x :: nat) c) ∈ I)
define P where P = ?P'
let ?P'' = λ (dir :: ('i,'a) Direction).
  P (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd dir) (UI dir) (LI dir) (LE dir) (GE dir)
have x:  $\bigwedge$  s'. ?P s' = P (<)  $\mathcal{B}_{iu}$   $\mathcal{B}_{il}$   $\mathcal{B}_u$   $\mathcal{B}_l$   $\mathcal{B}_{iu}$ -update  $\mathcal{I}_u$   $\mathcal{I}_l$  Leq Geq s'
  and xx:  $\bigwedge$  s'. ?P s' = P (>)  $\mathcal{B}_{il}$   $\mathcal{B}_{iu}$   $\mathcal{B}_l$   $\mathcal{B}_u$   $\mathcal{B}_{il}$ -update  $\mathcal{I}_l$   $\mathcal{I}_u$  Geq Leq s'

```

```

unfolding satisfies-bounds-set.simps in-bounds.simps bound-compare-defs in-
dex-valid-def P-def
  by (auto split: option.split simp: indexl-def indexu-def boundsl-def boundsu-def)
  from valid have P'': dir = Positive ∨ dir = Negative  $\implies$  ?P'' dir s for dir
    using x[of s] xx[of s] by auto
  have UTrue: dir = Positive ∨ dir = Negative  $\implies$  ?P'' dir s  $\implies$  ?P'' dir
  (set-unsat I s) for dir s I
    unfolding P-def by (auto simp: boundsl-def boundsu-def indexl-def indexu-def)
    have updateIB: a = LE dir x c  $\implies$  dir = Positive ∨ dir = Negative  $\implies$  ?P'' dir
    dir s  $\implies$  ?P'' dir
      (updateBI (UBI-upd dir) i x c s) for dir x c s
    unfolding P-def using I by (auto split: if-splits simp: simp: boundsl-def
    boundsu-def indexl-def indexu-def)
    show index-valid (insert ia as) (assert-bound ia s) unfolding ia I-def[symmetric]
    proof ((rule assert-bound-cases[of - - P]; (intro UTrue x xx updateIB P'')?))
      fix x c and dir :: ('i,'a) Direction
      assume **: dir = Positive ∨ dir = Negative
      a = LE dir x c
      x  $\notin$  lvars (T s)
      let ?s = (updateBI (UBI-upd dir) i x c s)
      define s' where s' = ?s
      have 1:  $\Delta$  (T ?s) using * ** by auto
      have 2:  $\nabla$  ?s using id(1) ** * < $\nabla$  s> by auto
      have 3: x  $\notin$  lvars (T ?s) using id(1) ** * < $\nabla$  s> by auto
      have ?P'' dir ?s using ** by (intro updateIB P'') auto
      with update-id[of ?s x c, OF 1 2 3, unfolded Let-def] **(1)
      show 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 (updateBI (UBI-upd dir) i x c s))
      unfolding P-def s'-def[symmetric] by auto
    qed auto
  next
    fix s and ia :: ('i,'a) i-atom and ats :: ('i,'a) i-atom set
    assume *:  $\neg$  U s  $\models_{nolhs}$  s  $\Delta$  (T s)  $\nabla$  s  $\Diamond$  s and ats: ats  $\models_i$  BI s
    obtain i a where ia: ia = (i,a) by force
    have id:  $\bigwedge$  dir x c s. dir = Positive ∨ dir = Negative  $\implies$ 
       $\nabla$  (updateBI (UBI-upd dir) i x c s) =  $\nabla$  s
       $\bigwedge$  s I.  $\nabla$  (set-unsat I s) =  $\nabla$  s
      by (auto simp add: tableau-valuated-def)
    have idt: (c < (a :: 'a) ∨ c = a) = (c  $\leq$  a)
      (a < c ∨ c = a) = (c  $\geq$  a) for a c by auto
    define A where A = insert (i,a) ats
    let ?P =  $\lambda$  (s :: ('i,'a) state). A  $\models_i$  BI s
    let ?Q =  $\lambda$  bs (lt :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool)
      (UBI :: ('i,'a) state  $\Rightarrow$  ('i,'a) bounds-index) (LBI :: ('i,'a) state  $\Rightarrow$  ('i,'a)
      bounds-index)
      (UB :: ('i,'a) state  $\Rightarrow$  'a bounds) (LB :: ('i,'a) state  $\Rightarrow$  'a bounds)
      (UBI-upd :: (('i,'a) bounds-index  $\Rightarrow$  ('i,'a) bounds-index)  $\Rightarrow$  ('i,'a) state  $\Rightarrow$ 
      ('i,'a) state)

```

$UI\ LI$
 $(LE :: nat \Rightarrow 'a \Rightarrow 'a\ atom) (GE :: nat \Rightarrow 'a \Rightarrow 'a\ atom) s'.$
 $(\forall I v. (I :: 'i\ set, v) \models_{ias} bs \longrightarrow$
 $((\forall x c. LB\ s' x = Some\ c \longrightarrow LI\ s' x \in I \longrightarrow lt\ c (v\ x) \vee c = v\ x)$
 $\wedge (\forall x c. UB\ s' x = Some\ c \longrightarrow UI\ s' x \in I \longrightarrow lt (v\ x)\ c \vee v\ x = c)))$
define Q **where** $Q = ?Q$
let $?P' = Q\ A$
have $equiv:$
 $bs \models_i \mathcal{B}\ s' \longleftrightarrow Q\ bs (<) \mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_u \mathcal{B}_l \mathcal{B}_{iu}\text{-}update \mathcal{I}_u \mathcal{I}_l \text{Leq Geq } s'$
 $bs \models_i \mathcal{B}\ s' \longleftrightarrow Q\ bs (>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u \mathcal{B}_{il}\text{-}update \mathcal{I}_l \mathcal{I}_u \text{Geq Leq } s'$
for $bs\ s'$
unfolding *satisfies-bounds-set.simps in-bounds.simps bound-compare-defs index-valid-def Q-def atoms-imply-bounds-index.simps*
by (*atomize(full)*, (*intro conjI iff-exI allI arg-cong2[of ---- (Λ)] refl iff-allI arg-cong2[of ---- (=)]*; *unfold satisfies-bounds-index.simps idlt*), *auto*)
have $x: \bigwedge s'. ?P\ s' = ?P' (<) \mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_u \mathcal{B}_l \mathcal{B}_{iu}\text{-}update \mathcal{I}_u \mathcal{I}_l \text{Leq Geq } s'$
and $xx: \bigwedge s'. ?P\ s' = ?P' (>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u \mathcal{B}_{il}\text{-}update \mathcal{I}_l \mathcal{I}_u \text{Geq Leq } s'$
using *equiv by blast+*
from *ats equiv[of ats s]*
have $Q\text{-ats}:$
 $Q\ ats (<) \mathcal{B}_{iu} \mathcal{B}_{il} \mathcal{B}_u \mathcal{B}_l \mathcal{B}_{iu}\text{-}update \mathcal{I}_u \mathcal{I}_l \text{Leq Geq } s$
 $Q\ ats (>) \mathcal{B}_{il} \mathcal{B}_{iu} \mathcal{B}_l \mathcal{B}_u \mathcal{B}_{il}\text{-}update \mathcal{I}_l \mathcal{I}_u \text{Geq Leq } s$
by *auto*
let $?P'' = \lambda (dir :: ('i, 'a) Direction). ?P' (lt dir) (UBI dir) (LBI dir) (UB dir)$
 $(LB dir) (UBI-upd dir) (UI dir) (LI dir) (LE dir) (GE dir)$
have $P\text{-upd}: dir = Positive \vee dir = Negative \implies ?P''\ dir (set-unsat I s) = ?P''\ dir\ s$ **for** $s\ I\ dir$
unfolding *Q-def*
by (*intro iff-exI arg-cong2[of ---- (Λ)] refl iff-allI arg-cong2[of ---- (=)] arg-cong2[of ---- (→)]*, *auto simp: boundsl-def boundsu-def indexl-def indexu-def*)
have $P\text{-upd}: dir = Positive \vee dir = Negative \implies ?P''\ dir\ s \implies ?P''\ dir$ *(set-unsat I s)* **for** $s\ I\ dir$
using *P-upd[of dir] by blast*
have *ats-sub: ats ⊆ A* **unfolding** *A-def by auto*
{
fix $x\ c$ **and** $dir :: ('i, 'a) Direction$
assume $dir: dir = Positive \vee dir = Negative$
and $a: a = LE\ dir\ x\ c$
from *Q-ats dir*
have *Q-ats: Q ats (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd dir) (UI dir) (LI dir) (LE dir) (GE dir) s*
by *auto*
have $?P''\ dir (update\mathcal{B}\ s' (UBI-upd dir) i\ x\ c\ s)$
unfolding *Q-def*
proof (*intro allI impI conjI*)
fix $I\ v\ y\ d$
assume $IvA: (I, v) \models_{ias} A$

```

from i-satisfies-atom-set-mono[OF ats-sub this]
have (I, v) ⊨ias ats by auto
from Q-ats[unfolded Q-def, rule-format, OF this]
have s-bnds:
  LB dir s x = Some c ==> LI dir s x ∈ I ==> lt dir c (v x) ∨ c = v x
  UB dir s x = Some c ==> UI dir s x ∈ I ==> lt dir (v x) c ∨ v x = c for x
c by auto
  from IvA[unfolded A-def, unfolded i-satisfies-atom-set.simps satisfies-atom-set-def,
simplified]
    have va: i ∈ I ==> v ⊨a a by auto
    with a dir have vc: i ∈ I ==> lt dir (v x) c ∨ v x = c
      by auto
    let ?s = (updateBI (UBI-upd dir) i x c s)
    show LB dir ?s y = Some d ==> LI dir ?s y ∈ I ==> lt dir d (v y) ∨ d = v y
      UB dir ?s y = Some d ==> UI dir ?s y ∈ I ==> lt dir (v y) d ∨ v y = d
    proof (atomize(full), goal-cases)
      case 1
      consider (main) y = x UI dir ?s x = i |
        (easy1) x ≠ y | (easy2) x = y UI dir ?s y ≠ i
        by blast
      then show ?case
      proof cases
        case easy1
        then show ?thesis using s-bnds[of y d] dir by (fastforce simp: boundsl-def
boundsu-def indexl-def indexu-def)
      next
        case easy2
        then show ?thesis using s-bnds[of y d] dir by (fastforce simp: boundsl-def
boundsu-def indexl-def indexu-def)
      next
        case main
        note s-bnds = s-bnds[of x]
        show ?thesis unfolding main using s-bnds dir vc
          by (auto simp: boundsl-def boundsu-def indexl-def indexu-def)
      qed
    qed
  qed
} note main = this
have Ps: dir = Positive ∨ dir = Negative ==> ?P'' dir s for dir
  using Q-ats unfolding Q-def using i-satisfies-atom-set-mono[OF ats-sub] by
fastforce
  have ?P (assert-bound (i,a) s)
  proof ((rule assert-bound-cases[of - - ?P']; (intro x xx P-upd main Ps ?)))
    fix c x and dir :: ('i,'a) Direction
    assume **: dir = Positive ∨ dir = Negative
    a = LE dir x c
    x ∉ lvars (T s)
  let ?s = updateBI (UBI-upd dir) i x c s
  define s' where s' = ?s

```

```

from main[ $OF **(1-2)$ ] have  $P: ?P'' \text{ dir } s' \text{ unfolding } s'\text{-def}$  .
have 1:  $\Delta(\mathcal{T} ?s)$  using *** by auto
have 2:  $\nabla ?s$  using  $id(1) *** \langle \nabla s \rangle$  by auto
have 3:  $x \notin lvars(\mathcal{T} ?s)$  using  $id(1) *** \langle \nabla s \rangle$  by auto
have  $\Delta(\mathcal{T} s') \nabla s' x \notin lvars(\mathcal{T} s')$  using 1 2 3 unfolding  $s'\text{-def}$  by auto
from update-bounds-id[ $OF this, of c$ ] **(1)
have  $?P'' \text{ dir } (update x c s') = ?P'' \text{ dir } s'$ 
    unfolding  $Q\text{-def}$ 
    by (intro iff-allI arg-cong2[of - - - - (→)] arg-cong2[of - - - - (Λ)] refl, auto)
with  $P$ 
show  $?P'' \text{ dir } (update x c ?s)$  unfolding  $s'\text{-def}$  by blast
qed auto
then show insert ia ats  $\models_i \mathcal{BI}$  (assert-bound ia s) unfolding ia A-def by blast
qed

```

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:
   $\llbracket x_j \in rvars-eq eq; lhs eq \notin rvars-eq eq \rrbracket \implies \text{let } eq' = \text{pivot-eq eq } x_j \text{ in}$ 
   $lhs eq' = x_j \wedge rvars-eq eq' = \{lhs eq\} \cup (rvars-eq eq - \{x_j\})$  and
  — Pivoting keeps the equation equisatisfiable.

equiv-pivot-eq:
 $\llbracket x_j \in rvars-eq eq; lhs eq \notin rvars-eq eq \rrbracket \implies$ 
 $(v::'a::lrv \text{ valuation}) \models_e \text{pivot-eq eq } x_j \longleftrightarrow v \models_e eq$ 

begin

lemma lhs-pivot-eq:
 $\llbracket x_j \in rvars-eq eq; lhs eq \notin rvars-eq eq \rrbracket \implies \text{lhs } (\text{pivot-eq eq } x_j) = x_j$ 
using vars-pivot-eq
by (simp add: Let-def)

lemma rvars-pivot-eq:
 $\llbracket x_j \in rvars-eq eq; lhs eq \notin rvars-eq eq \rrbracket \implies rvars-eq (\text{pivot-eq eq } x_j) = \{lhs eq\}$ 
 $\cup (rvars-eq eq - \{x_j\})$ 
using vars-pivot-eq
by (simp add: Let-def)

end

```

abbreviation doublesub ($\langle - \subseteq s - \subseteq s \rightarrow [50,51,51] \ 50 \rangle$) **where**

doublesub a b c \equiv $a \subseteq b \wedge b \subseteq c$

```

locale SubstVar =
  fixes subst-var::var  $\Rightarrow$  linear-poly  $\Rightarrow$  linear-poly  $\Rightarrow$  linear-poly
  assumes
    — Effect of subst-var  $x_j$   $lp'$   $lp$  on  $lp$  variables.

    vars-subst-var':
    (vars lp − { $x_j$ }) − vars lp'  $\subseteq_s$  vars (subst-var  $x_j$   $lp'$   $lp$ )  $\subseteq_s$  (vars lp − { $x_j$ })  $\cup$ 
    vars lp' and

    subst-no-effect:  $x_j \notin \text{vars } lp \implies \text{subst-var } x_j \text{ } lp' \text{ } lp = lp$  and

    subst-with-effect:  $x_j \in \text{vars } lp \implies x \in \text{vars } lp' - \text{vars } lp \implies x \in \text{vars (subst-var}$ 
     $x_j \text{ } lp' \text{ } lp)$  and

    — Effect of subst-var  $x_j$   $lp'$   $lp$  on  $lp$  value.

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

begin

lemma vars-subst-var:
  vars (subst-var  $x_j$   $lp'$   $lp$ )  $\subseteq$  (vars lp − { $x_j$ })  $\cup$  vars lp'
  using vars-subst-var'
  by simp

lemma vars-subst-var-supset:
  vars (subst-var  $x_j$   $lp'$   $lp$ )  $\supseteq$  (vars lp − { $x_j$ }) − vars lp'
  using vars-subst-var'
  by simp

definition subst-var-eq :: var  $\Rightarrow$  linear-poly  $\Rightarrow$  eq  $\Rightarrow$  eq where
  subst-var-eq v lp' eq  $\equiv$  (lhs eq, subst-var v lp' (rhs eq))

lemma rvars-eq-subst-var-eq:
  shows rvars-eq (subst-var-eq  $x_j$   $lp$  eq)  $\subseteq$  (rvars-eq eq − { $x_j$ })  $\cup$  vars lp
  unfolding subst-var-eq-def
  by (auto simp add: vars-subst-var)

lemma rvars-eq-subst-var-eq-supset:
  rvars-eq (subst-var-eq  $x_j$   $lp$  eq)  $\supseteq$  (rvars-eq eq) − { $x_j$ } − (vars lp)
  unfolding subst-var-eq-def
  by (simp add: vars-subst-var-supset)

lemma equiv-subst-var-eq:
  assumes (v::'a valuation)  $\models_e (x_j, lp')$ 

```

```

shows  $v \models_e eq \longleftrightarrow v \models_e subst\text{-}var\text{-}eq x_j lp' eq$ 
using assms
unfolding subst-var-eq-def
unfolding satisfies-eq-def
using equiv-subst-var[of v x_j lp' rhs eq]
by auto
end

locale Pivot' = EqForLVar + PivotEq + SubstVar
begin
definition pivot-tableau' :: var  $\Rightarrow$  var  $\Rightarrow$  tableau  $\Rightarrow$  tableau where
x_i x_j t  $\equiv$ 
let  $x_i\text{-idx} = eq\text{-idx-for-lvar } t \ x_i$ ;  $eq = t ! x_i\text{-idx}$ ;  $eq' = pivot\text{-eq } eq \ x_j$  in
map ( $\lambda \ idx. \ if \ idx = x_i\text{-idx} \ then \ eq' \ else \ subst\text{-var\text{-}eq } x_j (rhs \ eq') (t ! \ idx)$ 
) [0.. $<length \ t$ ]

definition pivot' :: var  $\Rightarrow$  var  $\Rightarrow$  ('i,'a::lrv) state  $\Rightarrow$  ('i,'a) state where
pivot' x_i x_j s  $\equiv$   $\mathcal{T}\text{-update} (pivot\text{-tableau'} x_i x_j (\mathcal{T} \ s)) \ s$ 

```

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

```

definition pivot-tableau :: var  $\Rightarrow$  var  $\Rightarrow$  tableau  $\Rightarrow$  tableau where
pivot-tableau x_i x_j t  $\equiv$  let  $eq = eq\text{-for-lvar } t \ x_i$ ;  $eq' = pivot\text{-eq } eq \ x_j$  in
map ( $\lambda \ e. \ if \ lhs \ e = lhs \ eq \ then \ eq' \ else \ subst\text{-var\text{-}eq } x_j (rhs \ eq') \ e$ ) t

definition pivot :: var  $\Rightarrow$  var  $\Rightarrow$  ('i,'a::lrv) state  $\Rightarrow$  ('i,'a) state where
pivot x_i x_j s  $\equiv$   $\mathcal{T}\text{-update} (pivot\text{-tableau} x_i x_j (\mathcal{T} \ s)) \ s$ 

lemma pivot-tableau'pivot-tableau:
assumes  $\Delta \ t \ x_i \in lvars \ t$ 
shows pivot-tableau' x_i x_j t = pivot-tableau x_i x_j t
proof-
let ?f =  $\lambda idx. \ if \ idx = eq\text{-idx-for-lvar } t \ x_i \ then \ pivot\text{-eq } (t ! eq\text{-idx-for-lvar } t \ x_i)$ 
x_j
else  $subst\text{-var\text{-}eq } x_j (rhs (pivot\text{-eq } (t ! eq\text{-idx-for-lvar } t \ x_i) \ x_j)) (t ! \ idx)$ 
let ?f' =  $\lambda e. \ if \ lhs \ e = lhs (eq\text{-for-lvar } t \ x_i) \ then \ pivot\text{-eq } (eq\text{-for-lvar } t \ x_i) \ x_j$ 
else  $subst\text{-var\text{-}eq } x_j (rhs (pivot\text{-eq } (eq\text{-for-lvar } t \ x_i) \ x_j)) \ e$ 
have  $\forall \ i < length \ t. \ ?f' (t ! i) = ?f \ i$ 
proof(safe)
fix i
assume i < length t
then have  $t ! i \in set \ t$   $i < length \ t$ 
by auto
moreover
have  $t ! eq\text{-idx-for-lvar } t \ x_i \in set \ t$   $eq\text{-idx-for-lvar } t \ x_i < length \ t$ 
using eq-for-lvar[of x_i t]  $\langle x_i \in lvars \ t \rangle$  eq-idx-for-lvar[of x_i t]
by (auto simp add: eq-for-lvar-def)

```

```

ultimately
have lhs (t ! i) = lhs (t ! eq-idx-for-lvar t xi)  $\implies$  t ! i = t ! (eq-idx-for-lvar t  

 $x_i$ ) distinct t
    using  $\langle \Delta t \rangle$ 
    unfolding normalized-tableau-def
    by (auto simp add: distinct-map inj-on-def)
then have lhs (t ! i) = lhs (t ! eq-idx-for-lvar t xi)  $\implies$  i = eq-idx-for-lvar t xi
    using  $\langle i < length t \rangle \langle eq-idx-for-lvar t xi < length t \rangle$ 
    by (auto simp add: distinct-conv-nth)
then show ?f' (t ! i) = ?f i
    by (auto simp add: eq-for-lvar-def)
qed
then show pivot-tableau' xi xj t = pivot-tableau xi xj t
    unfolding pivot-tableau'-def pivot-tableau-def
    unfolding Let-def
    by (auto simp add: map-reindex)
qed

lemma pivot'pivot: fixes s :: ('i,'a::lrv)state
assumes  $\Delta (\mathcal{T} s) x_i \in lvars (\mathcal{T} s)$ 
shows pivot' xi xj s = pivot xi xj s
using pivot-tableau'pivot-tableau[OF assms]
unfolding pivot-def pivot'-def by auto
end

sublocale Pivot' < Pivot eq-idx-for-lvar pivot
proof
fix s::('i,'a) state and xi xj and v::'a valuation
assume  $\Delta (\mathcal{T} s) x_i \in lvars (\mathcal{T} s)$ 
 $x_j \in rvars-eq (eq-for-lvar (\mathcal{T} s) x_i)$ 
show let s' = pivot xi xj s in V s' = V s ∧ Bi s' = Bi s ∧ U s' = U s ∧ Uc s'  

 $= U_c s$ 
    unfolding pivot-def
    by (auto simp add: Let-def simp: boundsl-def boundsu-def indexl-def indexu-def)
let ?t = T s
let ?idx = eq-idx-for-lvar ?t xi
let ?eq = ?t ! ?idx
let ?eq' = pivot-eq ?eq xj

have ?idx < length ?t lhs (?t ! ?idx) = xi
    using  $\langle x_i \in lvars ?t \rangle$ 
    using eq-idx-for-lvar
    by auto

have distinct (map lhs ?t)
    using  $\langle \Delta ?t \rangle$ 
    unfolding normalized-tableau-def

```

by *simp*

```

have  $x_j \in rvars\text{-eq} ?eq$ 
  using ⟨ $x_j \in rvars\text{-eq}$  (eq-for-lvar ( $\mathcal{T} s$ )  $x_i$ )⟩
  unfolding eq-for-lvar-def
  by simp
then have  $x_j \in rvars ?t$ 
  using ⟨?idx < length ?t⟩
  using in-set-conv-nth[of ?eq ?t]
  by (auto simp add: rvars-def)
then have  $x_j \notin lvars ?t$ 
  using ⟨ $\Delta$  ?t⟩
  unfolding normalized-tableau-def
  by auto

have  $x_i \notin rvars ?t$ 
  using ⟨ $x_i \in lvars ?t$ ⟩ ⟨ $\Delta$  ?t⟩
  unfolding normalized-tableau-def rvars-def
  by auto
then have  $x_i \notin rvars\text{-eq} ?eq$ 
  unfolding rvars-def
  using ⟨?idx < length ?t⟩
  using in-set-conv-nth[of ?eq ?t]
  by auto

have  $x_i \neq x_j$ 
  using ⟨ $x_j \in rvars\text{-eq} ?eq$ ⟩ ⟨ $x_i \notin rvars\text{-eq} ?eq$ ⟩
  by auto

have  $?eq' = (x_j, rhs ?eq')$ 
  using lhs-pivot-eq[of  $x_j$  ?eq]
  using ⟨ $x_j \in rvars\text{-eq}$  (eq-for-lvar ( $\mathcal{T} s$ )  $x_i$ )⟩ ⟨lhs (?t ! ?idx) =  $x_i$ ⟩ ⟨ $x_i \notin rvars\text{-eq}$  ?eq'⟩
  by (auto simp add: eq-for-lvar-def) (cases ?eq', simp)+

let ?I1 = [0..<?idx]
let ?I2 = [|?idx + 1..<length ?t|]
have [0..<length ?t] = ?I1 @ [|?idx|] @ ?I2
  using ⟨?idx < length ?t⟩
  by (rule interval-3split)
then have map-lhs-pivot:
  map lhs ( $\mathcal{T}$  (pivot'  $x_i x_j s$ )) =
    map ( $\lambda idx.$  lhs (?t ! idx)) ?I1 @ [| $x_j$ |] @ map ( $\lambda idx.$  lhs (?t ! idx)) ?I2
  using ⟨ $x_j \in rvars\text{-eq}$  (eq-for-lvar ( $\mathcal{T} s$ )  $x_i$ )⟩ ⟨lhs (?t ! ?idx) =  $x_i$ ⟩ ⟨ $x_i \notin rvars\text{-eq}$  ?eq'⟩
  by (auto simp add: Let-def subst-var-eq-def eq-for-lvar-def lhs-pivot-eq pivot'-def
    pivot-tableau'-def)

have lvars-pivot: lvars ( $\mathcal{T}$  (pivot'  $x_i x_j s$ )) =

```

```

lvars (T s) = {xi} ∪ {xj}

proof-
  have lvars (T (pivot' xi xj s)) =
    {xj} ∪ (λidx. lhs (?t ! idx)) ‘ ({0..<length?t} – {?idx}) 
    using ⟨?idx < length ?t⟩ ⟨?eq' = (xj, rhs ?eq')⟩
    by (cases ?eq', auto simp add: Let-def pivot'-def pivot-tableau'-def lvars-def
  subst-var-eq-def)+

  also have ... = {xj} ∪ (((λidx. lhs (?t ! idx)) ‘ {0..<length?t}) – {lhs (?t !
  ?idx)}))
    using ⟨?idx < length ?t⟩ ⟨distinct (map lhs ?t)⟩
    by (auto simp add: distinct-conv-nth)
  also have ... = {xj} ∪ (set (map lhs ?t) – {xi})
    using ⟨lhs (?t ! ?idx) = xi⟩
    by (auto simp add: in-set-conv-nth rev-image-eqI) (auto simp add: image-def)
  finally show lvars (T (pivot' xi xj s)) =
    lvars (T s) – {xi} ∪ {xj}
    by (simp add: lvars-def)

qed
moreover
have rvars-pivot: rvars (T (pivot' xi xj s)) =
  rvars (T s) – {xj} ∪ {xi}

proof-
  have rvars-eq ?eq' = {xi} ∪ (rvars-eq ?eq – {xj})
    using rvars-pivot-eq[of xj ?eq]
    using ⟨lhs (?t ! ?idx) = xi⟩
    using ⟨xj ∈ rvars-eq ?eq⟩ ⟨xi ∉ rvars-eq ?eq⟩
    by simp

  let ?S1 = rvars-eq ?eq'
  let ?S2 = ⋃ idx ∈ ({0..<length ?t} – {?idx}).
    rvars-eq (subst-var-eq xj (rhs ?eq') (?t ! idx))

  have rvars (T (pivot' xi xj s)) = ?S1 ∪ ?S2
    unfolding pivot'-def pivot-tableau'-def rvars-def
    using ⟨?idx < length ?t⟩
    by (auto simp add: Let-def split: if-splits)
  also have ... = {xi} ∪ (rvars ?t – {xj}) (is ?S1 ∪ ?S2 = ?rhs)
  proof
    show ?S1 ∪ ?S2 ⊆ ?rhs
    proof-
      have ?S1 ⊆ ?rhs
        using ⟨?idx < length ?t⟩
        unfolding rvars-def
        using ⟨rvars-eq ?eq' = {xi} ∪ (rvars-eq ?eq – {xj})⟩
        by (force simp add: in-set-conv-nth)
      moreover
      have ?S2 ⊆ ?rhs
      proof-
        have ?S2 ⊆ (⋃ idx ∈ {0..<length ?t}. (rvars-eq (?t ! idx) – {xj}) ∪ rvars-eq

```

```

?eq')
  apply (rule UN-mono)
  using rvars-eq-subst-var-eq
  by auto
  also have ... ⊆ rvars-eq ?eq' ∪ (∪ idx∈{0..?t}. rvars-eq (?t ! idx)
- {xj})
    by auto
  also have ... = rvars-eq ?eq' ∪ (rvars ?t - {xj})
    unfolding rvars-def
    by (force simp add: in-set-conv-nth)
  finally show ?thesis
    using ⟨rvars-eq ?eq' = {xi} ∪ (rvars-eq ?eq - {xj})⟩
    unfolding rvars-def
    using ⟨?idx < length ?t⟩
    by auto
qed
ultimately
show ?thesis
by simp
qed
next
show ?rhs ⊆ ?S1 ∪ ?S2
proof
fix x
assume x ∈ ?rhs
show x ∈ ?S1 ∪ ?S2
proof (cases x ∈ rvars-eq ?eq')
  case True
  then show ?thesis
  by auto
next
  case False
let ?S2' = ∪ idx∈({0..?t} - {?idx}).  

           (rvars-eq (?t ! idx) - {xj}) - rvars-eq ?eq'
have x ∈ ?S2'
  using False ⟨x ∈ ?rhs⟩
  using ⟨rvars-eq ?eq' = {xi} ∪ (rvars-eq ?eq - {xj})⟩
  unfolding rvars-def
  by (force simp add: in-set-conv-nth)
moreover
have ?S2 ⊇ ?S2'
  apply (rule UN-mono)
  using rvars-eq-subst-var-eq-supset[of - xj rhs ?eq' ]
  by auto
ultimately
show ?thesis
  by auto
qed
qed

```

```

qed
ultimately
show ?thesis
  by simp
qed
ultimately
show let s' = pivot xi xj s in rvars (T s') = rvars (T s) - {xj} ∪ {xi} ∧ lvars
  (T s') = lvars (T s) - {xi} ∪ {xj}
  using pivot'pivot[where ?i = 'i]
  using △ (T s) ⟨xi ∈ lvars (T s)⟩
  by (simp add: Let-def)
have △ (T (pivot' xi xj s))
  unfolding normalized-tableau-def
proof
  have lvars (T (pivot' xi xj s)) ∩ rvars (T (pivot' xi xj s)) = {} (is ?g1)
    using △ (T s)
    unfolding normalized-tableau-def
    using lvars-pivot rvars-pivot
    using xi ≠ xj
    by auto

moreover have 0 ∈ rhs ‘set (T (pivot' xi xj s)) (is ?g2)
proof
  let ?eq = eq-for-lvar (T s) xi
  from eq-for-lvar[OF ⟨xi ∈ lvars (T s)⟩]
  have ?eq ∈ set (T s) and var: lhs ?eq = xi by auto
  have lhs ?eq ∈ rvars-eq ?eq using △ (T s) ⟨?eq ∈ set (T s)⟩
    using ⟨xi ∈ rvars-eq (T s ! eq-idx-for-lvar (T s) xi)⟩ eq-for-lvar-def var by
    auto
  from vars-pivot-eq[OF ⟨xj ∈ rvars-eq ?eq⟩ this]
  have vars-pivot: lhs (pivot-eq ?eq xj) = xj rvars-eq (pivot-eq ?eq xj) = {lhs
    (eq-for-lvar (T s) xi)} ∪ (rvars-eq (eq-for-lvar (T s) xi) - {xj})
    unfolding Let-def by auto
  from vars-pivot(2) have rhs-pivot0: rhs (pivot-eq ?eq xj) ≠ 0 using vars-zero
  by auto
  assume 0 ∈ rhs ‘set (T (pivot' xi xj s))
  from this[unfolded pivot'pivot[OF ⟨△ (T s), xi ∈ lvars (T s)⟩] pivot-def]
  have 0 ∈ rhs ‘set (pivot-tableau xi xj (T s)) by simp
  from this[unfolded pivot-tableau-def Let-def var, unfolded var] rhs-pivot0
  obtain e where e ∈ set (T s) lhs e ≠ xi and rvars-eq: rvars-eq (subst-var-eq
    xj (rhs (pivot-eq ?eq xj)) e) = {}
    by (auto simp: vars-zero)
  from rvars-eq[unfolded subst-var-eq-def]
  have empty: vars (subst-var xj (rhs (pivot-eq ?eq xj)) (rhs e)) = {} by auto
  show False
  proof (cases xj ∈ vars (rhs e))
    case False
    from empty[unfolded subst-no-effect[OF False]]
    have rvars-eq e = {} by auto
  
```

```

hence rhs e = 0 using zero-coeff-zero coeff-zero by auto
with ⟨e ∈ set (T s)⟩ ⟨△ (T s)⟩ show False unfolding normalized-tableau-def
by auto
next
  case True
    from ⟨e ∈ set (T s)⟩ have rvars-eq e ⊆ rvars (T s) unfolding rvars-def
  by auto
    hence xi ∈ vars (rhs (pivot-eq ?eq xj)) = rvars-eq e
      unfolding vars-pivot(2) var
      using ⟨△ (T s)⟩ [unfolded normalized-tableau-def] ⟨xi ∈ lvars (T s)⟩ by
    auto
      from subst-with-effect[OF True this] rvars-eq
      show ?thesis by (simp add: subst-var-eq-def)
    qed
  qed
ultimately show ?g1 ∧ ?g2 ..

show distinct (map lhs (T (pivot' xi xj s)))
  using map-parametrize-idx[of lhs ?t]
  using map-lhs-pivot
  using ⟨distinct (map lhs ?t)⟩
  using interval-3split[of ?idx length ?t] ⟨?idx < length ?t⟩
  using ⟨xj ∉ lvars ?t⟩
  unfolding lvars-def
  by auto
qed
moreover
have v ⊨t ?t = v ⊨t T (pivot' xi xj s)
  unfolding satisfies-tableau-def
proof
  assume ∀ e∈set (?t). v ⊨e e
  show ∀ e∈set (T (pivot' xi xj s)). v ⊨e e
  proof-
    have v ⊨e ?eq'
      using ⟨xi ∉ rvars-eq ?eq⟩
      using ⟨?idx < length ?t⟩ ⟨∀ e∈set (?t). v ⊨e e⟩
      using ⟨xj ∈ rvars-eq ?eq⟩ ⟨xi ∈ lvars ?t⟩
      by (simp add: equiv-pivot-eq eq-idx-for-lvar)
  moreover
  {
    fix idx
    assume idx < length ?t idx ≠ ?idx
    have v ⊨e subst-var-eq xj (rhs ?eq') (?t ! idx)
      using ⟨?eq' = (xj, rhs ?eq')⟩
      using ⟨v ⊨e ?eq'⟩ ⟨idx < length ?t⟩ ⟨∀ e∈set (?t). v ⊨e e⟩
      using equiv-subst-var-eq[of v xj rhs ?eq' ?t ! idx]
    by auto
  }

```

```

}
ultimately
show ?thesis
  by (auto simp add: pivot'-def pivot-tableau'-def Let-def)
qed
next
  assume  $\forall e \in set (\mathcal{T} (pivot' x_i x_j s)). v \models_e e$ 
  then have  $v \models_e ?eq'$ 
     $\wedge idx. [idx < length ?t; idx \neq ?idx] \implies v \models_e subst\text{-}var\text{-}eq x_j (rhs ?eq') (?t ! idx)$ 
    using ‹?idx < length ?t›
    unfolding pivot'-def pivot-tableau'-def
    by (auto simp add: Let-def)

  show  $\forall e \in set (\mathcal{T} s). v \models_e e$ 
proof-
{
  fix idx
  assume  $idx < length ?t$ 
  have  $v \models_e (?t ! idx)$ 
  proof (cases  $idx = ?idx$ )
    case True
    then show ?thesis
      using ‹v \models_e ?eq'›
      using ‹x_j \in rvars\text{-}eq ?eq› ‹x_i \in lvars ?t› ‹x_i \notin rvars\text{-}eq ?eq›
      by (simp add: eq-idx-for-lvar equiv-pivot-eq)
  next
    case False
    then show ?thesis
      using ‹idx < length ?t›
      using ‹[idx < length ?t; idx \neq ?idx] \implies v \models_e subst\text{-}var\text{-}eq x_j (rhs ?eq') (?t ! idx)›
      using ‹v \models_e ?eq'› ‹?eq' = (x_j, rhs ?eq')›
      using equiv-subst-var-eq[of v x_j rhs ?eq' ?t ! idx]
      by auto
  qed
}
then show ?thesis
  by (force simp add: in-set-conv-nth)
qed
qed
ultimately
show let  $s' = pivot x_i x_j s$  in  $v \models_t \mathcal{T} s = v \models_t \mathcal{T} s' \wedge \Delta (\mathcal{T} s')$ 
  using pivot'pivot[where ?'i = 'i]
  using ‹\Delta (\mathcal{T} s)› ‹x_i \in lvars (\mathcal{T} s)›
  by (simp add: Let-def)
qed

```

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 → (forall x∈lvars (T s). in-bounds x ⟨V s⟩ (B s)) and

min-lvar-not-in-bounds-Some': min-lvar-not-in-bounds s = Some xi → xi∈lvars
(T s) ∧ ¬in-bounds xi ⟨V s⟩ (B s)
    ∧ (forall x∈lvars (T s). x < xi → in-bounds x ⟨V s⟩ (B s))

begin
lemma min-lvar-not-in-bounds-None':
min-lvar-not-in-bounds s = None → ((⟨V s⟩ ⊨b B s || lvars (T s))
unfoldng satisfies-bounds-set.simps
by (rule min-lvar-not-in-bounds-None)

lemma min-lvar-not-in-bounds-lvars:min-lvar-not-in-bounds s = Some xi → xi
    ∈ lvars (T s)
using min-lvar-not-in-bounds-Some'
by simp

lemma min-lvar-not-in-bounds-Some: min-lvar-not-in-bounds s = Some xi → ¬
    in-bounds xi ⟨V s⟩ (B s)
using min-lvar-not-in-bounds-Some'
by simp

lemma min-lvar-not-in-bounds-Some-min: min-lvar-not-in-bounds s = Some xi
    → (forall x ∈ lvars (T s). x < xi → in-bounds x ⟨V s⟩ (B s))

```

```

using min-lvar-not-in-bounds-Some'
by simp

end

abbreviation reasable-var where
reasable-var dir x eq s ≡
  (coeff (rhs eq) x > 0 ∧ ▷ub (lt dir) (⟨V s⟩ x) (UB dir s x)) ∨
  (coeff (rhs eq) x < 0 ∧ ▷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 xj ⇒ xj ∈ rvars-eq eq
  assumes min-rvar-incdec-eq-Some-incdec:
    min-rvar-incdec-eq dir s eq = Inr xj ⇒ reasable-var dir xj eq s
  assumes min-rvar-incdec-eq-Some-min:
    min-rvar-incdec-eq dir s eq = Inr xj ⇒
    (∀ x ∈ rvars-eq eq. x < xj → ¬ 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) ⊨ib BI s
  shows le (lt dir) ((rhs eq) {v}) ((rhs eq) {⟨V s⟩})
proof -
  have ∀ x ∈ rvars-eq eq. ¬ reasable-var dir x eq s
  using min
  using min-rvar-incdec-eq-None
  by simp

  have ∀ x ∈ rvars-eq eq. (0 < coeff (rhs eq) x → le (lt dir) 0 (⟨V s⟩ x - v x))
  ∧ (coeff (rhs eq) x < 0 → le (lt dir) (⟨V s⟩ x - v x) 0)
  proof (safe)
    fix x
    assume x: x ∈ rvars-eq eq 0 < coeff (rhs eq) x 0 ≠ ⟨V s⟩ x - v x
    then have ¬ (▷ub (lt dir) (⟨V s⟩ x) (UB dir s x))
    using ∀ x ∈ rvars-eq eq. ¬ reasable-var dir x eq s

```

```

    by auto
  then have  $\sqsubseteq_{ub} (lt\ dir) (\langle \mathcal{V}\ s \rangle\ x) (UB\ dir\ s\ x)$ 
    using *
    by (cases  $UB\ dir\ s\ x$ ) (auto simp add: bound-compare-defs)
  moreover
  from min-rvar-incdec-eq-None[ $OF\ min$ ] x sub have  $UI\ dir\ s\ x \in I$  by auto
  from Iv * this
  have  $\sqsubseteq_{ub} (lt\ dir) (v\ x) (UB\ dir\ s\ x)$ 
    unfolding satisfies-bounds-index.simps
    by (cases  $UB\ dir\ s\ x$ , auto simp: indexl-def indexu-def boundsl-def boundsu-def
        bound-compare'-defs)
       (fastforce) +
  ultimately
  have  $le (lt\ dir) (v\ x) (\langle \mathcal{V}\ s \rangle\ x)$ 
    using *
    by (cases  $UB\ dir\ s\ x$ ) (auto simp add: bound-compare-defs)
  then show  $lt\ dir\ 0 (\langle \mathcal{V}\ s \rangle\ x - v\ x)$ 
    using  $\langle 0 \neq \langle \mathcal{V}\ s \rangle\ x - v\ x \rangle *$ 
    using minus-gt[of  $v\ x\ \langle \mathcal{V}\ s \rangle\ x$ ] minus-lt[of  $\langle \mathcal{V}\ s \rangle\ x\ v\ x$ ]
    by (auto simp del: Simplex.bounds-lg)
next
fix x
assume  $x: x \in rvars\text{-eq eq } 0 > coeff (rhs\ eq)\ x\ \langle \mathcal{V}\ s \rangle\ x - v\ x \neq 0$ 
then have  $\neg (\triangleright_{lb} (lt\ dir) (\langle \mathcal{V}\ s \rangle\ x) (LB\ dir\ s\ x))$ 
  using  $\forall x \in rvars\text{-eq eq. } \neg \text{reasable-var dir } x \text{ eq } s$ 
  by auto
then have  $\sqsubseteq_{lb} (lt\ dir) (\langle \mathcal{V}\ s \rangle\ x) (LB\ dir\ s\ x)$ 
  using *
  by (cases  $LB\ dir\ s\ x$ ) (auto simp add: bound-compare-defs)
moreover
from min-rvar-incdec-eq-None[ $OF\ min$ ] x sub have  $LI\ dir\ s\ x \in I$  by auto
from Iv * this
have  $\sqsupseteq_{lb} (lt\ dir) (v\ x) (LB\ dir\ s\ x)$ 
  unfolding satisfies-bounds-index.simps
  by (cases  $LB\ dir\ s\ x$ , auto simp: indexl-def indexu-def boundsl-def boundsu-def
      bound-compare'-defs)
     (fastforce) +
  ultimately
  have  $le (lt\ dir) (\langle \mathcal{V}\ s \rangle\ x) (v\ x)$ 
    using *
    by (cases  $LB\ dir\ s\ x$ ) (auto simp add: bound-compare-defs)
  then show  $lt\ dir\ (\langle \mathcal{V}\ s \rangle\ x - v\ x) 0$ 
    using  $\langle \langle \mathcal{V}\ s \rangle\ x - v\ x \neq 0 \rangle *$ 
    using minus-lt[of  $\langle \mathcal{V}\ s \rangle\ x\ v\ x$ ] minus-gt[of  $v\ x\ \langle \mathcal{V}\ s \rangle\ x$ ]
    by (auto simp del: Simplex.bounds-lg)
qed
then have  $le (lt\ dir) 0 (rhs\ eq\ \{\lambda x. \langle \mathcal{V}\ s \rangle\ x - v\ x\})$ 
  using *

```

```

apply auto
using valuate-nonneg[of rhs eq  $\lambda x. \langle \mathcal{V} s \rangle x - v x$ ]
  apply (force simp del: Simplex.bounds-lg)
using valuate-nonpos[of rhs eq  $\lambda x. \langle \mathcal{V} s \rangle x - v x$ ]
  apply (force simp del: Simplex.bounds-lg)
done
then show le (lt dir) rhs eq {v} rhs eq {⟨V s⟩}
  using <dir = Positive ∨ dir = Negative>
  using minus-gt[of rhs eq {v} rhs eq {⟨V s⟩}]
  by (auto simp add: valuate-diff[THEN sym] simp del: Simplex.bounds-lg)
qed
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 xi ≡ min-rvar-incdec-eq dir s (eq-for-lvar (T s) xi)
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 xi s ≡
let li = the (LB dir s xi);
xi' = min-rvar-incdec dir s xi
in case xi' of
  Inl I ⇒ set-unsat I s
  | Inr xi ⇒ pivot-and-update xi xi' li s

lemma check'-cases:
assumes ⋀ I. [min-rvar-incdec dir s xi = Inl I; check' dir xi s = set-unsat I s]

```

```

 $\implies P(\text{set-unsat } I \ s)$ 
assumes  $\bigwedge x_j \ l_i. [\text{min-rvar-incdec dir } s \ x_i = \text{Inr } x_j;$ 
 $l_i = \text{the (LB dir } s \ x_i);$ 
 $\text{check'} \ \text{dir } x_i \ s = \text{pivot-and-update } x_i \ x_j \ l_i \ s] \implies$ 
 $P(\text{pivot-and-update } x_i \ x_j \ l_i \ s)$ 
shows  $P(\text{check'} \ \text{dir } x_i \ s)$ 
using assms
unfolding check'-def
by (cases min-rvar-incdec dir s xi, auto)
partial-function (tailrec) check where
check s =
  (if U s then s
   else let xi' = min-lvar-not-in-bounds s
     in case xi' of
       None  $\Rightarrow s$ 
     | Some xi => let dir = if <V s> xi <lb Bl s xi then Positive
        else Negative
        in check (check' dir xi s)))
declare check.simps[code]

inductive check-dom where
  step: [ $\bigwedge x_i. [\neg U s; \text{Some } x_i = \text{min-lvar-not-in-bounds } s; \langle V s \rangle x_i <_{lb} B_l s x_i]$ ]
   $\implies \text{check-dom}(\text{check'} \text{Positive } x_i \ s);$ 
   $\bigwedge x_i. [\neg U s; \text{Some } x_i = \text{min-lvar-not-in-bounds } s; \neg \langle V s \rangle x_i <_{lb} B_l s x_i]$ 
   $\implies \text{check-dom}(\text{check'} \text{Negative } x_i \ s)]$ 
 $\implies \text{check-dom } s$ 

```

The definition of *check* can be given by:

```

check s  $\equiv$  if U s then s
  else let xi' = min-lvar-not-in-bounds s in
    case xi' of None  $\Rightarrow s$ 
    | Some xi => if <V s> xi <lb Bl s xi then check (check-inc xi s)
    else check (check-dec xi s)

check-inc xi s  $\equiv$  let li = the (Bl s xi); xj' = min-rvar-inc s xi in
  case xj' of None  $\Rightarrow s \ \parallel U := \text{True} \ \parallel | \text{Some } x_j \Rightarrow \text{pivot-and-update } x_i \ x_j \ l_i \ s$ 

```

The definition of *check-dec* is analogous. It is shown (mainly by induction) that this definition satisfies the *check* specification. Note that this definition uses general recursion, so its termination is non-trivial. It has been shown that it terminates for all states satisfying the check preconditions. The proof is based on the proof outline given in [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 ⟨V s⟩  $x_i <_{lb} \mathcal{B}_l s x_i$  then Positive else Negative)
min-lvar-not-in-bounds s = Some  $x_i$ 
min-rvar-incdec dir s  $x_i$  = Inr  $x_j$ 
 $l_i$  = the (LB dir s  $x_i$ )
 $\nabla s \triangle (\mathcal{T} s) \models_{nolhs} s \diamondsuit s$ 
shows  $\triangle (\mathcal{T} (\text{pivot-and-update } x_i x_j l_i s)) \wedge \models_{nolhs} (\text{pivot-and-update } x_i x_j l_i s)$ 
 $\wedge \diamondsuit (\text{pivot-and-update } x_i x_j l_i s) \wedge \nabla (\text{pivot-and-update } x_i x_j l_i s)$ 
proof-
have  $\mathcal{B}_l s x_i = \text{Some } l_i \vee \mathcal{B}_u s x_i = \text{Some } l_i$ 
using ⟨ $l_i = \text{the (LB dir s } x_i)$ ⟩ ⟨dir = (if ⟨V s⟩  $x_i <_{lb} \mathcal{B}_l s x_i$  then Positive else Negative)⟩
using ⟨min-lvar-not-in-bounds s = Some  $x_i$ ⟩ min-lvar-not-in-bounds-Some[of s  $x_i$ ]
using ⟨ $\diamondsuit s$ ⟩
by (case-tac[!]  $\mathcal{B}_l s x_i$ , case-tac[!]  $\mathcal{B}_u s x_i$ ) (auto simp add: bounds-consistent-def bound-compare-defs)
then show ?thesis
using assms
using pivotandupdate-tableau-normalized[of s  $x_i x_j l_i$ ]
using pivotandupdate-nolhs[of s  $x_i x_j l_i$ ]
using pivotandupdate-bounds-consistent[of s  $x_i x_j l_i$ ]
using pivotandupdate-tableau-valuated[of s  $x_i x_j l_i$ ]
by (auto simp add: min-lvar-not-in-bounds-lvars min-rvar-incdec-eq-Some-rvars)
qed

```

```

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

definition gt-state :: ('i,'a) state ⇒ ('i,'a) state ⇒ bool (infixl ⟨succ_x⟩ 100) where
ssucc_x s' ≡
∃  $x_i x_j l_i$ .
let dir = if ⟨V s⟩  $x_i <_{lb} \mathcal{B}_l s x_i$  then Positive else Negative in
gt-state' dir s s'  $x_i x_j l_i$ 

abbreviation succ :: ('i,'a) state ⇒ ('i,'a) state ⇒ bool (infixl ⟨succ⟩ 100) where
ssucc s' ≡ △ (T s) ∧  $\diamondsuit s \wedge \models_{nolhs} s \wedge \nabla s \wedge s \succ_x s' \wedge \mathcal{B}_i s' = \mathcal{B}_i s \wedge \mathcal{U}_c s' = \mathcal{U}_c s$ 

abbreviation succ-rel :: ('i,'a) state rel where
succ-rel ≡ {(s, s'). ssucc s'}
```

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

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

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

proof—
from *assms*
obtain $x_i x_j c$
where *:
 $\triangle (\mathcal{T} s) \nabla s$
 $\text{min-lvar-not-in-bounds } s = \text{Some } x_i$
 $\text{min-rvar-incdec Positive } s x_i = \text{Inr } x_j \vee \text{min-rvar-incdec Negative } s x_i = \text{Inr } x_j$
 $s' = \text{pivot-and-update } x_i x_j c s$
unfolding *gt-state-def*
by (*auto split: if-splits*)
then have
 $x_i \in \text{lvars } (\mathcal{T} s)$
 $x_j \in \text{rvars-eq } (\text{eq-for-lvar } (\mathcal{T} s) x_i)$
 $\text{lvars } (\mathcal{T} s') = \text{lvars } (\mathcal{T} s) - \{x_i\} \cup \{x_j\}$
 $\text{rvars } (\mathcal{T} s') = \text{rvars } (\mathcal{T} s) - \{x_j\} \cup \{x_i\}$
using *min-lvar-not-in-bounds-lvars*[of *s* *x_i*]
using *min-rvar-incdec-eq-Some-rvars*[of *Positive s eq-for-lvar* ($\mathcal{T} s$) *x_i x_j*]
using *min-rvar-incdec-eq-Some-rvars*[of *Negative s eq-for-lvar* ($\mathcal{T} s$) *x_i x_j*]
using *pivotandupdate-rvars*[of *s* *x_i x_j*]
using *pivotandupdate-lvars*[of *s* *x_i x_j*]
by auto
moreover
have $x_j \in \text{rvars } (\mathcal{T} s)$
using $\langle x_i \in \text{lvars } (\mathcal{T} s) \rangle$
using $\langle x_j \in \text{rvars-eq } (\text{eq-for-lvar } (\mathcal{T} s) x_i) \rangle$
using *eq-for-lvar*[of *x_i T s*]
unfolding *rvars-def*
by auto
ultimately
have
 $x_i \in \text{lvars } (\mathcal{T} s)$
 $x_j \in \text{rvars-of-lvar } (\mathcal{T} s)$ $x_i x_j \in \text{rvars } (\mathcal{T} s)$

```

lvars ( $\mathcal{T} s'$ ) = lvars ( $\mathcal{T} s$ ) - { $x_i$ }  $\cup$  { $x_j$ }
rvars ( $\mathcal{T} s'$ ) = rvars ( $\mathcal{T} s$ ) - { $x_j$ }  $\cup$  { $x_i$ }
by auto
then show thesis
 $\dots$ 
qed

lemma succ-vars-id:
assumes  $s \succ s'$ 
shows lvars ( $\mathcal{T} s$ )  $\cup$  rvars ( $\mathcal{T} s$ ) =
    lvars ( $\mathcal{T} s'$ )  $\cup$  rvars ( $\mathcal{T} s'$ )
using assms
by (rule succ-vars) auto

lemma succ-inv:
assumes  $s \succ s'$ 
shows  $\Delta(\mathcal{T} s') \nabla s' \diamond s' \mathcal{B}_i s = \mathcal{B}_i s'$ 
    ( $v::'a$  valuation)  $\models_t (\mathcal{T} s) \longleftrightarrow v \models_t (\mathcal{T} s')$ 
proof-
from assms obtain  $x_i x_j c$ 
where *:
 $\Delta(\mathcal{T} s) \nabla s \diamond s$ 
 $\text{min-lvar-not-in-bounds } s = \text{Some } x_i$ 
 $\text{min-rvar-incdec Positive } s x_i = \text{Inr } x_j \vee \text{min-rvar-incdec Negative } s x_i = \text{Inr } x_j$ 
 $s' = \text{pivot-and-update } x_i x_j c s$ 
unfolding gt-state-def
by (auto split: if-splits)
then show  $\Delta(\mathcal{T} s') \nabla s' \diamond s' \mathcal{B}_i s = \mathcal{B}_i s'$ 
    ( $v::'a$  valuation)  $\models_t (\mathcal{T} s) \longleftrightarrow v \models_t (\mathcal{T} s')$ 
using min-lvar-not-in-bounds-lvars[of s x_i]
using min-rvar-incdec-eq-Positive s eq-for-lvar ( $\mathcal{T} s$ )  $x_i x_j$ 
using min-rvar-incdec-eq-Negative s eq-for-lvar ( $\mathcal{T} s$ )  $x_i x_j$ 
using pivotandupdate-tableau-normalized [of s  $x_i x_j c$ ]
using pivotandupdate-bounds-consistent [of s  $x_i x_j c$ ]
using pivotandupdate-bounds-id [of s  $x_i x_j c$ ]
using pivotandupdate-tableau-equiv
using pivotandupdate-tableau-valuated
by auto
qed

lemma succ-rvar-valuation-id:
assumes  $s \succ s' x \in \text{rvars } (\mathcal{T} s) x \in \text{rvars } (\mathcal{T} s')$ 
shows  $\langle \mathcal{V} s \rangle x = \langle \mathcal{V} s' \rangle x$ 
proof-
from assms obtain  $x_i x_j c$ 
where *:
 $\Delta(\mathcal{T} s) \nabla s \diamond s$ 
 $\text{min-lvar-not-in-bounds } s = \text{Some } x_i$ 

```

```

min-rvar-inedec Positive s xi = Inr xj ∨ min-rvar-incdec Negative s xi = Inr
xj
  s' = pivot-and-update xi xj c s
unfolding gt-state-def
by (auto split: if-splits)
then show ?thesis
  using min-lvar-not-in-bounds-lvars[of s xi]
  using min-rvar-incdec-eq-Some-rvars[of Positive s eq-for-lvar (T s) xi xj]
  using min-rvar-incdec-eq-Some-rvars[of Negative s eq-for-lvar (T s) xi xj]
  using ⟨x ∈ rvars (T s)⟩ ⟨x ∈ rvars (T s')⟩
  using pivotandupdate-rvars[of s xi xj c]
  using pivotandupdate-valuation-xi[of s xi xj c]
  using pivotandupdate-valuation-other-nolhs[of s xi xj x c]
  by (force simp add: normalized-tableau-def map2fun-def)
qed

```

```

lemma succ-min-lvar-not-in-bounds:
assumes s ≻ s'
  xr ∈ lvars (T s) xr ∈ rvars (T s')
shows ¬ in-bounds xr ((V s)) (B s)
  ∀ x ∈ lvars (T s). x < xr → in-bounds x ((V s)) (B s)
proof-
  from assms obtain xi xj c
  where *:
    △ (T s) ∇ s ◁ s
    min-lvar-not-in-bounds s = Some xi
    min-rvar-inedec Positive s xi = Inr xj ∨ min-rvar-incdec Negative s xi = Inr
xj
  s' = pivot-and-update xi xj c s
  unfolding gt-state-def
  by (auto split: if-splits)
  then have xi = xr
    using min-lvar-not-in-bounds-lvars[of s xi]
    using min-rvar-incdec-eq-Some-rvars[of Positive s eq-for-lvar (T s) xi xj]
    using min-rvar-incdec-eq-Some-rvars[of Negative s eq-for-lvar (T s) xi xj]
    using ⟨xr ∈ lvars (T s)⟩ ⟨xr ∈ rvars (T s')⟩
    using pivotandupdate-rvars
    by (auto simp add: normalized-tableau-def)
  then show ¬ in-bounds xr ((V s)) (B s)
  ∀ x ∈ lvars (T s). x < xr → in-bounds x ((V s)) (B s)
  using ⟨min-lvar-not-in-bounds s = Some xi⟩
  using min-lvar-not-in-bounds-Some min-lvar-not-in-bounds-Some-min
  by simp-all
qed

```

```

lemma succ-min-rvar:
assumes s ≻ s'
  xs ∈ lvars (T s) xs ∈ rvars (T s')
  xr ∈ rvars (T s) xr ∈ lvars (T s')

```

$eq = eq\text{-for-lvar } (\mathcal{T} s) \ xs \text{ and}$
 $dir: dir = Positive \vee dir = Negative$
shows
 $\neg \sqsupseteq_{lb} (lt \ dir) (\langle \mathcal{V} s \rangle \ xs) (LB \ dir \ s \ xs) \longrightarrow$
 $reasable\text{-var } dir \ xr \ eq \ s \wedge (\forall x \in rvars\text{-eq } eq. \ x < xr \longrightarrow \neg reasable\text{-var}$
 $dir \ x \ eq \ s)$
proof-
from $assms(1)$ **obtain** $x_i \ x_j \ c$
where $\Delta (\mathcal{T} s) \wedge \nabla s \wedge \Diamond s \wedge \models_{nolhs} s$
 $gt\text{-state}' (if \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l \ s \ x_i \ then \ Positive \ else \ Negative) \ s \ s' \ x_i \ x_j \ c$
by (auto simp add: $gt\text{-state-def Let-def}$)
then have
 $\Delta (\mathcal{T} s) \nabla s \Diamond s$
 $min\text{-lvar-not-in-bounds } s = Some \ x_i$
 $s' = pivot\text{-and-update } x_i \ x_j \ c \ s \text{ and}$
 $*: (\langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l \ s \ x_i \wedge min\text{-rvar-incdec Positive } s \ x_i = Inr \ x_j) \vee$
 $(\neg \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l \ s \ x_i \wedge min\text{-rvar-incdec Negative } s \ x_i = Inr \ x_j)$
by (auto split: if-splits)
then have $xr = x_j \ xs = x_i$
using $min\text{-lvar-not-in-bounds-lvars}[of \ s \ x_i]$
using $min\text{-rvar-incdec-eq-Some-rvars}[of \ Positive \ s \ eq\text{-for-lvar } (\mathcal{T} s) \ x_i \ x_j]$
using $min\text{-rvar-incdec-eq-Some-rvars}[of \ Negative \ s \ eq\text{-for-lvar } (\mathcal{T} s) \ x_i \ x_j]$
using $\langle xr \in rvars \ (\mathcal{T} s) \rangle \langle xr \in lvars \ (\mathcal{T} s') \rangle$
using $\langle xs \in lvars \ (\mathcal{T} s) \rangle \langle xs \in rvars \ (\mathcal{T} s') \rangle$
using $pivot\text{andupdate-lvars} \ pivot\text{andupdate-rvars}$
by (auto simp add: normalized-tableau-def)
show $\neg (\sqsubseteq_{lb} (lt \ dir) (\langle \mathcal{V} s \rangle \ xs) (LB \ dir \ s \ xs)) \longrightarrow$
 $reasable\text{-var } dir \ xr \ eq \ s \wedge (\forall x \in rvars\text{-eq } eq. \ x < xr \longrightarrow \neg reasable\text{-var}$
 $dir \ x \ eq \ s)$
proof
assume $\neg \sqsupseteq_{lb} (lt \ dir) (\langle \mathcal{V} s \rangle \ xs) (LB \ dir \ s \ xs)$
then have $\triangleleft_{lb} (lt \ dir) (\langle \mathcal{V} s \rangle \ xs) (LB \ dir \ s \ xs)$
using dir
by (cases $LB \ dir \ s \ xs$) (auto simp add: bound-compare-defs)
moreover
then have $\neg (\triangleright_{ub} (lt \ dir) (\langle \mathcal{V} s \rangle \ xs) (UB \ dir \ s \ xs))$
using $\langle \Diamond s \rangle \ dir$
using $bounds\text{-consistent-gt-ub} \ bounds\text{-consistent-lt-lb}$
by (force simp add: bound-compare"-defs)
ultimately
have $min\text{-rvar-incdec } dir \ s \ xs = Inr \ xr$
using * $\langle xr = x_j \rangle \langle xs = x_i \rangle \ dir$
by (auto simp add: bound-compare"-defs)
then show $reasable\text{-var } dir \ xr \ eq \ s \wedge (\forall x \in rvars\text{-eq } eq. \ x < xr \longrightarrow \neg$
 $reasable\text{-var } dir \ x \ eq \ s)$
using $\langle eq = eq\text{-for-lvar } (\mathcal{T} s) \ xs \rangle$
using $min\text{-rvar-incdec-eq-Some-min}[of \ dir \ s \ eq \ xr]$
using $min\text{-rvar-incdec-eq-Some-incdec}[of \ dir \ s \ eq \ xr]$

by simp

qed

qed

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

$\neg \trianglelefteq_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i) \longrightarrow \langle \mathcal{V} s' \rangle x_i = the(LB\ dir\ s\ x_i)$
 $\langle \mathcal{V} s' \rangle x_i = the(\mathcal{B}_l\ s\ x_i) \vee \langle \mathcal{V} s' \rangle x_i = the(\mathcal{B}_u\ s\ x_i)$

proof –

from assms(1) obtain $x_i' x_j c$

where $\triangle(\mathcal{T} s) \wedge \nabla s \wedge \diamond s \wedge \models_{nolhs} s$

gt-state' (if $\langle \mathcal{V} s \rangle x_i' <_{lb} \mathcal{B}_l\ s\ x_i'$ then Positive else Negative) $s s' x_i' x_j c$

by (auto simp add: gt-state-def Let-def)

then have

$\triangle(\mathcal{T} s) \nabla s \diamond s$

min-lvar-not-in-bounds $s = Some x_i'$

$s' = pivot-and-update x_i' x_j c s$ **and**

*: $(\langle \mathcal{V} s \rangle x_i' <_{lb} \mathcal{B}_l\ s\ x_i' \wedge c = the(\mathcal{B}_l\ s\ x_i')) \wedge min-rvar-incdec Positive\ s\ x_i'$
 $= Inr x_j) \vee$

$(\neg \langle \mathcal{V} s \rangle x_i' <_{lb} \mathcal{B}_l\ s\ x_i' \wedge c = the(\mathcal{B}_u\ s\ x_i')) \wedge min-rvar-incdec Negative\ s$

$x_i' = Inr x_j)$

by (auto split: if-splits)

then have $x_i = x_i' x_i' \in lvars(\mathcal{T} s)$

$x_j \in rvars-eq(eq-for-lvar(\mathcal{T} s) x_i')$

using min-lvar-not-in-bounds-lvars[of s x_i']

using min-rvar-incdec-eq-Some-rvars[of Positive s eq-for-lvar(\mathcal{T} s) x_i' x_j]

using min-rvar-incdec-eq-Some-rvars[of Negative s eq-for-lvar(\mathcal{T} s) x_i' x_j]

using $\langle x_i \in lvars(\mathcal{T} s) \rangle \langle x_i \in rvars(\mathcal{T} s') \rangle$

using pivotandupdate-rvars

by (auto simp add: normalized-tableau-def)

show $\neg \trianglelefteq_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i) \longrightarrow \langle \mathcal{V} s' \rangle x_i = the(LB\ dir\ s\ x_i)$

proof

assume $\neg \trianglelefteq_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i)$

then have $\trianglelefteq_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i)$

using dir

by (cases LB dir s x_i) (auto simp add: bound-compare-defs)

moreover

then have $\neg \triangleright_{ub} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (UB\ dir\ s\ x_i)$

using $\diamond s \triangleright dir$

using bounds-consistent-gt-ub bounds-consistent-lt-lb

by (force simp add: bound-compare''-defs)

ultimately

show $\langle \mathcal{V} s' \rangle x_i = the(LB\ dir\ s\ x_i)$

using * $\langle x_i = x_i' \rangle \langle s' = pivot-and-update x_i' x_j c s \rangle$

using $\langle \triangle(\mathcal{T} s) \rangle \langle \nabla s \rangle \langle x_i' \in lvars(\mathcal{T} s) \rangle$

$\langle x_j \in rvars-eq(eq-for-lvar(\mathcal{T} s) x_i') \rangle$

```

using pivotandupdate-valuation-xi[of s xi xj c] dir
by (case-tac[!]  $\mathcal{B}_l$  s xi', case-tac[!]  $\mathcal{B}_u$  s xi') (auto simp add: bound-compare-defs
map2fun-def)
qed

have  $\neg \langle \mathcal{V} s \rangle x_i' <_{lb} \mathcal{B}_l s x_i' \longrightarrow \langle \mathcal{V} s \rangle x_i' >_{ub} \mathcal{B}_u s x_i'$ 
using ‹min-lvar-not-in-bounds s = Some xi'›
using min-lvar-not-in-bounds-Some[of s xi]
using not-in-bounds[of xi' ⟨V s⟩  $\mathcal{B}_l$  s  $\mathcal{B}_u$  s]
by auto
then show ⟨V s⟩ xi = the ( $\mathcal{B}_l$  s xi)  $\vee$  ⟨V s⟩ xi = the ( $\mathcal{B}_u$  s xi)
using ‹△ (T s)› ‹∇ s› ‹xi' ∈ lvars (T s)›
    ‹xj ∈ rvars-eq (eq-for-lvar (T s) xi')›
using ‹s' = pivot-and-update xi' xj c s› ‹xi = xi'›
using pivotandupdate-valuation-xi[of s xi xj c]
using *
by (case-tac[!]  $\mathcal{B}_l$  s xi', case-tac[!]  $\mathcal{B}_u$  s xi') (auto simp add: map2fun-def
bound-compare-defs)
qed

lemma succ-rvar-valuation:
assumes
    s ≻ s' x ∈ rvars (T s')
shows
    ⟨V s⟩ x = ⟨V s⟩ x  $\vee$  ⟨V s⟩ x = the ( $\mathcal{B}_l$  s x)  $\vee$  ⟨V s⟩ x = the ( $\mathcal{B}_u$  s x)
proof-
from assms
obtain xi xj b where
    △ (T s) ∇ s
    min-lvar-not-in-bounds s = Some xi
    min-rvar-incdec Positive s xi = Inr xj  $\vee$  min-rvar-incdec Negative s xi = Inr
    xj
    b = the ( $\mathcal{B}_l$  s xi)  $\vee$  b = the ( $\mathcal{B}_u$  s xi)
    s' = pivot-and-update xi xj b s
unfolding gt-state-def
by (auto simp add: Let-def split: if-splits)
then have
    xi ∈ lvars (T s) xi ∉ rvars (T s)
    xj ∈ rvars-eq (eq-for-lvar (T s) xi)
    xj ∈ rvars (T s) xj ∉ lvars (T s) xi ≠ xj
using min-lvar-not-in-bounds-lvars[of s xi]
using min-rvar-incdec-eq-Some-rvars[of Positive s eq-for-lvar (T s) xi xj]
using min-rvar-incdec-eq-Some-rvars[of Negative s eq-for-lvar (T s) xi xj]
using rvars-of-lvar-rvars ‹△ (T s)›
by (auto simp add: normalized-tableau-def)
then have
    rvars (T s') = rvars (T s) - {xj} ∪ {xi}
    x ∈ rvars (T s)  $\vee$  x = xi x ≠ xj x ≠ xi  $\longrightarrow$  x ∉ lvars (T s)
using ‹x ∈ rvars (T s')›

```

```

using pivotandupdate-rvars[of s xi xj]
using ⟨△ (T s)⟩ ⟨∇ s⟩ ⟨s' = pivot-and-update xi xj b s⟩
by (auto simp add: normalized-tableau-def)
then show ?thesis
using pivotandupdate-valuation-xi[of s xi xj b]
using pivotandupdate-valuation-other-nolhs[of s xi xj x b]
using ⟨xi ∈ lvars (T s)⟩ ⟨xj ∈ rvars-eq (eq-for-lvar (T s) xi)⟩
using ⟨△ (T s)⟩ ⟨∇ s⟩ ⟨s' = pivot-and-update xi xj b s⟩ ⟨b = the (Bl s xi) ∨ b
= the (Bu s xi)⟩
by (auto simp add: map2fun-def)
qed

lemma succ-no-vars-valuation:
assumes
s ≻ s' x ∉ tvars (T s')
shows look (V s') x = look (V s) x
proof –
from assms
obtain xi xj b where
△ (T s) ∇ s
min-lvar-not-in-bounds s = Some xi
min-rvar-incdec Positive s xi = Inr xj ∨ min-rvar-incdec Negative s xi = Inr
xj
b = the (Bl s xi) ∨ b = the (Bu s xi)
s' = pivot-and-update xi xj b s
unfolding gt-state-def
by (auto simp add: Let-def split: if-splits)
then have
xi ∈ lvars (T s) xi ∉ rvars (T s)
xj ∈ rvars-eq (eq-for-lvar (T s) xi)
xj ∈ rvars (T s) xj ∉ lvars (T s) xi ≠ xj
using min-lvar-not-in-bounds-lvars[of s xi]
using min-rvar-incdec-eq-Some-rvars[of Positive s eq-for-lvar (T s) xi xj]
using min-rvar-incdec-eq-Some-rvars[of Negative s eq-for-lvar (T s) xi xj]
using rvars-of-lvar-rvars ⟨△ (T s)⟩
by (auto simp add: normalized-tableau-def)
then show ?thesis
using pivotandupdate-valuation-other-nolhs[of s xi xj x b]
using ⟨△ (T s)⟩ ⟨∇ s⟩ ⟨s' = pivot-and-update xi xj b s⟩
using ⟨x ∉ tvars (T s')⟩
using pivotandupdate-rvars[of s xi xj]
using pivotandupdate-lvars[of s xi xj]
by (auto simp add: map2fun-def)
qed

lemma succ-valuation-satisfies:
assumes s ≻ s' ⟨V s⟩ ⊨t T s
shows ⟨V s'⟩ ⊨t T s'
proof –

```

```

from ⟨ $s \succ s'$ ⟩
obtain  $x_i \ x_j \ b$  where
   $\Delta (\mathcal{T} s) \nabla s$ 
  min-lvar-not-in-bounds  $s = \text{Some } x_i$ 
  min-rvar-incdec Positive  $s \ x_i = \text{Inr } x_j \vee$  min-rvar-incdec Negative  $s \ x_i = \text{Inr }$ 
 $x_j$ 
   $b = \text{the } (\mathcal{B}_l \ s \ x_i) \vee b = \text{the } (\mathcal{B}_u \ s \ x_i)$ 
   $s' = \text{pivot-and-update } x_i \ x_j \ b \ s$ 
unfolding gt-state-def
by (auto simp add: Let-def split: if-splits)
then have
   $x_i \in \text{lvars } (\mathcal{T} s)$ 
   $x_j \in \text{rvars-of-lvar } (\mathcal{T} s) \ x_i$ 
using min-lvar-not-in-bounds-lvars[of  $s \ x_i$ ]
using min-rvar-incdec-eq-Some-rvars[of Positive  $s$  eq-for-lvar  $(\mathcal{T} s) \ x_i \ x_j$ ]
using min-rvar-incdec-eq-Some-rvars[of Negative  $s$  eq-for-lvar  $(\mathcal{T} s) \ x_i \ x_j$ ] ⟨ $\Delta$ 
 $(\mathcal{T} s)$ ⟩
by (auto simp add: normalized-tableau-def)
then show ?thesis
using pivotandupdate-satisfies-tableau[of  $s \ x_i \ x_j \ b$ ]
using pivotandupdate-tableau-equiv[of  $s \ x_i \ x_j$ ]
using ⟨ $\Delta (\mathcal{T} s)$ ⟩ ⟨ $\nabla s$ ⟩ ⟨ $\mathcal{V} s$ ⟩  $\models_t \mathcal{T} s \ s' = \text{pivot-and-update } x_i \ x_j \ b \ s$ 
by auto
qed

lemma succ-tableau-valuated:
assumes  $s \succ s' \nabla s$ 
shows  $\nabla s'$ 
using succ-inv(2) assms by blast

```

```

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

lemma succ-chain-induct:
assumes *: succ-chain  $l \ i \leq j \ j < \text{length } l$ 
assumes base:  $\bigwedge i. P i i$ 
assumes step:  $\bigwedge i. l ! i \succ (l ! (i + 1)) \implies P i (i + 1)$ 
assumes trans:  $\bigwedge i j k. [P i j; P j k; i < j; j \leq k] \implies P i k$ 
shows  $P i j$ 
using *
proof (induct  $j - i$  arbitrary:  $i$ )
  case 0
  then show ?case
    by (simp add: base)
  next
    case ( $Suc k$ )
    have  $P (i + 1) j$ 
    using Suc(1)[of  $i + 1$ ] Suc(2) Suc(3) Suc(4) Suc(5)

```

```

by auto
moreover
have P i (i + 1)
proof (rule step)
  show l ! i ⊳ (l ! (i + 1))
    using Suc(2) Suc(3) Suc(5)
    unfolding rel-chain-def
      by auto
qed
ultimately
show ?case
  using trans[of i i + 1 j] Suc(2)
  by simp
qed

lemma succ-chain-bounds-id:
assumes succ-chain l i ≤ j j < length l
shows Bi (l ! i) = Bi (l ! j)
using assms
proof (rule succ-chain-induct)
  fix i
  assume l ! i ⊳ (l ! (i + 1))
  then show Bi (l ! i) = Bi (l ! (i + 1))
    by (rule succ-inv(4))
qed simp-all

lemma succ-chain-vars-id':
assumes succ-chain l i ≤ j j < length l
shows lvars (T (l ! i)) ∪ rvars (T (l ! i)) =
      lvars (T (l ! j)) ∪ rvars (T (l ! j))
using assms
proof (rule succ-chain-induct)
  fix i
  assume l ! i ⊳ (l ! (i + 1))
  then show tvars (T (l ! i)) = tvars (T (l ! (i + 1)))
    by (rule succ-vars-id)
qed simp-all

lemma succ-chain-vars-id:
assumes succ-chain l i < length l j < length l
shows lvars (T (l ! i)) ∪ rvars (T (l ! i)) =
      lvars (T (l ! j)) ∪ rvars (T (l ! j))
proof (cases i ≤ j)
  case True
  then show ?thesis
    using assms succ-chain-vars-id'[of l i j]
    by simp
next
  case False

```

```

then have  $j \leq i$ 
  by simp
then show ?thesis
  using assms succ-chain-vars-id'[of l j i]
  by simp
qed

lemma succ-chain-tableau-equiv':
  assumes succ-chain  $l \ i \leq j \ j < \text{length } l$ 
  shows ( $v::'a \text{ valuation}$ )  $\models_t \mathcal{T}(l ! i) \longleftrightarrow v \models_t \mathcal{T}(l ! j)$ 
  using assms
proof (rule succ-chain-induct)
  fix  $i$ 
  assume  $l ! i \succ (l ! (i + 1))$ 
  then show  $v \models_t \mathcal{T}(l ! i) = v \models_t \mathcal{T}(l ! (i + 1))$ 
  by (rule succ-inv(5))
qed simp-all

lemma succ-chain-tableau-equiv:
  assumes succ-chain  $l \ i < \text{length } l \ j < \text{length } l$ 
  shows ( $v::'a \text{ valuation}$ )  $\models_t \mathcal{T}(l ! i) \longleftrightarrow v \models_t \mathcal{T}(l ! j)$ 
proof (cases  $i \leq j$ )
  case True
  then show ?thesis
  using assms succ-chain-tableau-equiv'[of l i j v]
  by simp
next
  case False
  then have  $j \leq i$ 
  by auto
  then show ?thesis
  using assms succ-chain-tableau-equiv'[of l j i v]
  by simp
qed

lemma succ-chain-no-vars-valuation:
  assumes succ-chain  $l \ i \leq j \ j < \text{length } l$ 
  shows  $\forall x. x \notin \text{tvars}(\mathcal{T}(l ! i)) \longrightarrow \text{look}(\mathcal{V}(l ! i)) x = \text{look}(\mathcal{V}(l ! j)) x$  (is ?P i j)
  using assms
proof (induct  $j - i$  arbitrary:  $i$ )
  case 0
  then show ?case
  by simp
next
  case (Suc  $k$ )
  have ?P  $(i + 1) j$ 
  using Suc(1)[of i + 1] Suc(2) Suc(3) Suc(4) Suc(5)
  by auto

```

```

moreover
have ?P (i + 1) i
proof (rule+, rule succ-no-vars-valuation)
  show l ! i ⊳ (l ! (i + 1))
    using Suc(2) Suc(3) Suc(5)
    unfolding rel-chain-def
    by auto
  qed
moreover
have tvars (T (l ! i)) = tvars (T (l ! (i + 1)))
proof (rule succ-vars-id)
  show l ! i ⊳ (l ! (i + 1))
    using Suc(2) Suc(3) Suc(5)
    unfolding rel-chain-def
    by simp
  qed
ultimately
show ?case
  by simp
qed

lemma succ-chain-rvar-valuation:
assumes succ-chain l i ≤ j j < length l
shows ∀ x ∈ rvars (T (l ! j)).
  ⟨V (l ! j)⟩ x = ⟨V (l ! i)⟩ x ∨
  ⟨V (l ! j)⟩ x = the (Bl (l ! i) x) ∨
  ⟨V (l ! j)⟩ x = the (Bu (l ! i) x) (is ?P i j)
using assms
proof (induct j - i arbitrary: j)
  case 0
  then show ?case
    by simp
next
  case (Suc k)
  have k = j - 1 - i succ-chain l i ≤ j - 1 j - 1 < length l j > 0
    using Suc(2) Suc(3) Suc(4) Suc(5)
    by auto
  then have ji: ?P i (j - 1)
    using Suc(1)
    by simp

  have l ! (j - 1) ⊳ (l ! j)
    using Suc(3) ⟨j < length l, j > 0⟩
    unfolding rel-chain-def
    by (erule-tac x=j - 1 in alle) simp

then have
  jj: ?P (j - 1) j
using succ-rvar-valuation

```

by auto

obtain $x_i \ x_j$ **where**

vars: $x_i \in lvars(\mathcal{T}(l ! (j - 1)))$ $x_j \in rvars(\mathcal{T}(l ! (j - 1)))$

$rvars(\mathcal{T}(l ! j)) = rvars(\mathcal{T}(l ! (j - 1))) - \{x_j\} \cup \{x_i\}$

using $\langle l ! (j - 1) \succ (l ! j)$

by (*rule succ-vars*) **simp**

then have bounds:

$\mathcal{B}_l(l ! (j - 1)) = \mathcal{B}_l(l ! i) \ \mathcal{B}_l(l ! j) = \mathcal{B}_l(l ! i)$

$\mathcal{B}_u(l ! (j - 1)) = \mathcal{B}_u(l ! i) \ \mathcal{B}_u(l ! j) = \mathcal{B}_u(l ! i)$

using $\langle succ-chain \ l \rangle$

using *succ-chain-bounds-id*[*of l i j - 1, THEN sym*] $\langle j - 1 < length l \rangle \ \langle i \leq j - 1 \rangle$

using *succ-chain-bounds-id*[*of l j - 1 j, THEN sym*] $\langle j < length l \rangle$

by (*auto simp: indexl-def indexu-def boundsl-def boundsu-def*)

show ?case

proof

fix x

assume $x \in rvars(\mathcal{T}(l ! j))$

then have $x \neq x_j \wedge x \in rvars(\mathcal{T}(l ! (j - 1))) \vee x = x_i$

using vars

by auto

then show $\langle \mathcal{V}(l ! j) \rangle x = \langle \mathcal{V}(l ! i) \rangle x \vee$

$\langle \mathcal{V}(l ! j) \rangle x = \text{the}(\mathcal{B}_l(l ! i) x) \vee$

$\langle \mathcal{V}(l ! j) \rangle x = \text{the}(\mathcal{B}_u(l ! i) x)$

proof

assume $x \neq x_j \wedge x \in rvars(\mathcal{T}(l ! (j - 1)))$

then show ?thesis

using *jj* $\langle x \in rvars(\mathcal{T}(l ! j)) \rangle ji$

using *bounds*

by force

next

assume $x = x_i$

then show ?thesis

using *succ-set-on-bound(2)*[*of l ! (j - 1) l ! j xi*] $\langle l ! (j - 1) \succ (l ! j) \rangle$

using vars bounds

by auto

qed

qed

qed

lemma *succ-chain-valuation-satisfies*:

assumes *succ-chain l i ≤ j j < length l*

shows $\langle \mathcal{V}(l ! i) \rangle \models_t \mathcal{T}(l ! i) \longrightarrow \langle \mathcal{V}(l ! j) \rangle \models_t \mathcal{T}(l ! j)$

using *assms*

proof (*rule succ-chain-induct*)

fix i

assume $l ! i \succ (l ! (i + 1))$

```

then show  $\langle \mathcal{V} (l ! i) \rangle \models_t \mathcal{T} (l ! i) \longrightarrow \langle \mathcal{V} (l ! (i + 1)) \rangle \models_t \mathcal{T} (l ! (i + 1))$ 
  using succ-valuation-satisfies
  by auto
qed simp-all

```

```

lemma succ-chain-tableau-valuated:
  assumes succ-chain  $l$   $i \leq j$   $j < \text{length } l$ 
  shows  $\nabla (l ! i) \longrightarrow \nabla (l ! j)$ 
  using assms
proof(rule succ-chain-induct)
  fix  $i$ 
  assume  $l ! i \succ (l ! (i + 1))$ 
  then show  $\nabla (l ! i) \longrightarrow \nabla (l ! (i + 1))$ 
    using succ-tableau-valuated
    by auto
qed simp-all

```

```

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

```

```

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

```

```

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

```

```

lemma succ-chain-always-r-valuation-id:
  assumes succ-chain  $l$   $i \leq j$   $j < \text{length } l$ 
  shows always-r  $l$   $i$   $j$   $x \longrightarrow \langle \mathcal{V} (l ! i) \rangle x = \langle \mathcal{V} (l ! j) \rangle x$  (is ?P  $i$   $j$ )
  using assms
proof (rule succ-chain-induct)
  fix  $i$ 
  assume  $l ! i \succ (l ! (i + 1))$ 
  then show ?P  $i$   $(i + 1)$ 
    using succ-rvar-valuation-id
    by simp
qed simp-all

```

```

lemma succ-chain-swap-rl-exists:
  assumes succ-chain  $l$   $i < j$   $j < \text{length } l$ 
     $x \in \text{rvars} (\mathcal{T} (l ! i))$   $x \in \text{lvars} (\mathcal{T} (l ! j))$ 
  shows  $\exists k. i \leq k \wedge k < j \wedge \text{swap-rl } l k x$ 
  using assms
proof (induct  $j - i$  arbitrary:  $i$ )
  case 0
  then show ?case
    by simp

```

```

next
  case (Suc k)
    have l ! i  $\succ$  (l ! (i + 1))
      using Suc(3) Suc(4) Suc(5)
      unfolding rel-chain-def
      by auto
    then have  $\triangle (\mathcal{T} (l ! (i + 1)))$ 
      by (rule succ-inv)
    show ?case
    proof (cases x ∈ rvars (T (l ! (i + 1))))
      case True
        then have j  $\neq i + 1$ 
        using Suc(7) △ (T (l ! (i + 1)))
        by (auto simp add: normalized-tableau-def)
        have k = j - Suc i
        using Suc(2)
        by simp
        then obtain k where k  $\geq i + 1$  k  $< j$  swap-rl l k x
        using ⟨x ∈ rvars (T (l ! (i + 1)))⟩ ⟨j  $\neq i + 1$ ⟩
        using Suc(1)[of i + 1] Suc(2) Suc(3) Suc(4) Suc(5) Suc(6) Suc(7)
        by auto
        then show ?thesis
        by (rule-tac x=k in exI) simp
    next
      case False
      then have x ∈ lvars (T (l ! (i + 1)))
      using Suc(6)
      using ⟨l ! i  $\succ$  (l ! (i + 1))⟩ succ-vars-id
      by auto
      then show ?thesis
      using Suc(4) Suc(5) Suc(6)
      by force
    qed
  qed

lemma succ-chain-swap-lr-exists:
  assumes succ-chain l i < j j < length l
  x ∈ lvars (T (l ! i)) x ∈ rvars (T (l ! j))
  shows  $\exists k. i \leq k \wedge k < j \wedge \text{swap-lr } l k x$ 
  using assms
  proof (induct j - i arbitrary: i)
    case 0
    then show ?case
      by simp
  next
    case (Suc k)
    have l ! i  $\succ$  (l ! (i + 1))
    using Suc(3) Suc(4) Suc(5)

```

```

unfolding rel-chain-def
by auto
then have  $\Delta (\mathcal{T} (l ! (i + 1)))$ 
by (rule succ-inv)

show ?case
proof (cases x ∈ lvars ( $\mathcal{T} (l ! (i + 1))$ ))
  case True
    then have  $j \neq i + 1$ 
    using Suc(7)  $\langle \Delta (\mathcal{T} (l ! (i + 1))) \rangle$ 
    by (auto simp add: normalized-tableau-def)
    have  $k = j - Suc i$ 
    using Suc(2)
    by simp
    then obtain k where  $k \geq i + 1$   $k < j$  swap-lr l k x
    using ⟨x ∈ lvars ( $\mathcal{T} (l ! (i + 1))$ )⟩ ⟨ $j \neq i + 1$ ⟩
    using Suc(1)[of i + 1] Suc(2) Suc(3) Suc(4) Suc(5) Suc(6) Suc(7)
    by auto
    then show ?thesis
    by (rule-tac x=k in exI) simp
  next
    case False
    then have x ∈ rvars ( $\mathcal{T} (l ! (i + 1))$ )
    using Suc(6)
    using ⟨l ! i ∼ (l ! (i + 1))⟩ succ-vars-id
    by auto
    then show ?thesis
    using Suc(4) Suc(5) Suc(6)
    by force
  qed
qed

```

```

lemma finite-tableaus-aux:
  shows finite {t. lvars t = L ∧ rvars t = V - L ∧  $\Delta t \wedge (\forall v::'a$  valuation.  $v \models_t t = v \models_t t0)$ } (is finite (?Al L))
  proof (cases ?Al L = {})
    case True
      show ?thesis
      by (subst True) simp
    next
      case False
      then have  $\exists t. t \in ?Al L$ 
      by auto
      let ?t = SOME t. t ∈ ?Al L
      have ?t ∈ ?Al L
      using ⟨ $\exists t. t \in ?Al L$ ⟩
      by (rule someI-ex)

```

```

have ?Al L ⊆ {t. mset t = mset ?t}
proof
  fix x
  assume x ∈ ?Al L
  have mset x = mset ?t
    apply (rule tableau-perm)
    using ‹?t ∈ ?Al L› ‹x ∈ ?Al L›
    by auto
  then show x ∈ {t. mset t = mset ?t}
    by simp
qed

moreover
have finite {t. mset t = mset ?t}
  by (fact mset-eq-finite)
ultimately
show ?thesis
  by (rule finite-subset)
qed

lemma finite-tableaus:
assumes finite V
shows finite {t. tvars t = V ∧ △ t ∧ (∀ v::'a valuation. v |=t t = v |=t t0)} (is finite ?A)
proof-
  let ?Al = λ L. {t. lvars t = L ∧ rvars t = V - L ∧ △ t ∧ (∀ v::'a valuation. v |=t t = v |=t t0)}
  have ?A = ⋃ (?Al ` {L. L ⊆ V})
    by (auto simp add: normalized-tableau-def)
  then show ?thesis
    using ‹finite V›
    using finite-tableaus-aux
    by auto
qed

lemma finite-accessible-tableaus:
shows finite (T ` {s'. s ≻* s'})
proof-
  have {s'. s ≻* s'} = {s'. s ≻+ s'} ∪ {s}
    by (auto simp add: rtrancel-eq-or-trancel)
  moreover
  have finite (T ` {s'. s ≻+ s'}) (is finite ?A)
  proof-
    let ?T = {t. tvars t = tvars (T s) ∧ △ t ∧ (∀ v::'a valuation. v |=t t = v |=t (T s))}
    have ?A ⊆ ?T
    proof
      fix t
      assume t ∈ ?A
      then obtain s' where s ≻+ s' t = T s'
    qed
  qed

```

```

by auto
then obtain l where *:  $l \neq []$   $1 < \text{length } l$   $\text{hd } l = s$   $\text{last } l = s'$   $\text{succ-chain } l$ 
  using trancl-rel-chain[of s s' succ-rel]
  by auto
show  $t \in ?T$ 
proof-
  have tvars ( $\mathcal{T} s'$ ) = tvars ( $\mathcal{T} s$ )
    using succ-chain-vars-id[of l 0 length l - 1]
    using * hd-conv-nth[of l] last-conv-nth[of l]
    by simp
  moreover
  have  $\Delta (\mathcal{T} s')$ 
    using  $\langle s \succ^+ s' \rangle$ 
    using succ-inv(1)[of - s']
    by (auto dest: tranclD2)
  moreover
  have  $\forall v::'a \text{ valuation. } v \models_t \mathcal{T} s' = v \models_t \mathcal{T} s$ 
    using succ-chain-tableau-equiv[of l 0 length l - 1]
    using * hd-conv-nth[of l] last-conv-nth[of l]
    by auto
  ultimately
  show ?thesis
    using  $\langle t = \mathcal{T} s' \rangle$ 
    by simp
qed
qed
moreover
have finite (tvars ( $\mathcal{T} s$ ))
  by (auto simp add: lvars-def rvars-def finite-vars)
ultimately
show ?thesis
  using finite-tableaus[of tvars ( $\mathcal{T} s$ )  $\mathcal{T} s$ ]
  by (auto simp add: finite-subset)
qed
ultimately
show ?thesis
  by simp
qed

```

abbreviation check-valuation **where**

```

check-valuation (v::'a valuation) v0 bl0 bu0 t0 V ≡
   $\exists t. \text{tvars } t = V \wedge \Delta t \wedge (\forall v::'a \text{ valuation. } v \models_t t = v \models_t t0) \wedge v \models_t t \wedge$ 
   $(\forall x \in \text{rvars } t. v x = v0 x \vee v x = bl0 x \vee v x = bu0 x) \wedge$ 
   $(\forall x. x \notin V \longrightarrow v x = v0 x)$ 

```

lemma finite-valuations:

assumes finite V

shows finite {v::'a valuation. check-valuation v v0 bl0 bu0 t0 V} (is finite ?A)

proof-

```

let ?Al = λ L. {t. lvars t = L ∧ rvars t = V - L ∧ △ t ∧ (∀ v::'a valuation. v
\models_t t = v |t t0)}
let ?Vt = λ t. {v::'a valuation. v |t t ∧ (∀ x ∈ rvars t. v x = v0 x ∨ v x = bl0
x ∨ v x = bu0 x) ∧ (∀ x. x ∉ V → v x = v0 x)}

have finite {L. L ⊆ V}
using ⟨finite V⟩
by auto
have ∀ L. L ⊆ V → finite (?Al L)
using finite-tableaus-aux
by auto
have ∀ L t. L ⊆ V ∧ t ∈ ?Al L → finite (?Vt t)
proof (safe)
fix L t
assume lvars t ⊆ V rvars t = V - lvars t △ t ∀ v. v |t t = v |t t0
then have rvars t ∪ lvars t = V
by auto

let ?f = λ v x. if x ∈ rvars t then v x else 0

have inj-on ?f (?Vt t)
unfolding inj-on-def
proof (safe, rule ext)
fix v1 v2 x
assume (λx. if x ∈ rvars t then v1 x else (0 :: 'a)) =
(λx. if x ∈ rvars t then v2 x else (0 :: 'a)) (is ?f1 = ?f2)
have ∀ x ∈ rvars t. v1 x = v2 x
proof
fix x
assume x ∈ rvars t
then show v1 x = v2 x
using ⟨?f1 = ?f2⟩ fun-cong[of ?f1 ?f2 x]
by auto
qed
assume *: v1 |t t v2 |t t
∀ x. x ∉ V → v1 x = v0 x ∀ x. x ∉ V → v2 x = v0 x
show v1 x = v2 x
proof (cases x ∈ lvars t)
case False
then show ?thesis
using * ⟨∀ x ∈ rvars t. v1 x = v2 x⟩ ⟨rvars t ∪ lvars t = V⟩
by auto
next
case True
let ?eq = eq-for-lvar t x
have ?eq ∈ set t ∧ lhs ?eq = x
using eq-for-lvar ⟨x ∈ lvars t⟩
by simp
then have v1 x = rhs ?eq {v1} v2 x = rhs ?eq {v2}

```

```

using <v1 ⊨t t> <v2 ⊨t t>
unfolding satisfies-tableau-def satisfies-eq-def
by auto
moreover
have rhs ?eq { v1 } = rhs ?eq { v2 }
apply (rule value-depend)
using <∀ x∈rvars t. v1 x = v2 x> <?eq ∈ set t ∧ lhs ?eq = x>
unfolding rvars-def
by auto
ultimately
show ?thesis
by simp
qed
qed

let ?R = {v. ∀ x. if x ∈ rvars t then v x = v0 x ∨ v x = bl0 x ∨ v x = bu0 x
else v x = 0 }
have ?f '(?Vt t) ⊆ ?R
by auto
moreover
have finite ?R
proof -
have finite (rvars t)
using <finite V> <rvars t ∪ lvars t = V>
using finite-subset[of rvars t V]
by auto
moreover
let ?R' = {v. ∀ x. if x ∈ rvars t then v x ∈ {v0 x, bl0 x, bu0 x} else v x = 0 }
have ?R = ?R'
by auto
ultimately
show ?thesis
using finite-fun-args[of rvars t λ x. {v0 x, bl0 x, bu0 x} λ x. 0]
by auto
qed
ultimately
have finite (?f '(?Vt t))
by (simp add: finite-subset)
then show finite (?Vt t)
using inj-on ?f (?Vt t)
by (auto dest: finite-imageD)
qed

have ?A = ⋃ (⋃ (((?Vt) ' (?Al ' {L. L ⊆ V}))) (is ?A = ?A')
by (auto simp add: normalized-tableau-def cong del: image-cong-simp)
moreover
have finite ?A'
proof (rule finite-Union)
show finite (⋃ (((?Vt) ' (?Al ' {L. L ⊆ V}))))

```

```

using ‹finite {L. L ⊆ V}› ‹∀ L. L ⊆ V —→ finite (?Al L)›
by auto
next
fix M
assume M ∈ ⋃ (((?) ?Vt) ‘ (?Al ‘ {L. L ⊆ V}))
then obtain L t where L ⊆ V t ∈ ?Al L M = ?Vt t
by blast
then show finite M
using ‹∀ L t. L ⊆ V ∧ t ∈ ?Al L —→ finite (?Vt t)›
by blast
qed
ultimately
show ?thesis
by simp
qed

```

lemma *finite-accessible-valuations*:

shows $\text{finite } (\mathcal{V} \setminus \{s'. s \succ^* s'\})$

proof–

have $\{s'. s \succ^* s'\} = \{s'. s \succ^+ s'\} \cup \{s\}$
by (auto simp add: rtrancl-eq-or-trancl)

moreover

have $\text{finite } (\mathcal{V} \setminus \{s'. s \succ^+ s'\})$ (**is finite** ?A)

proof–

let $?P = \lambda v. \text{check-valuation } v ((\langle \mathcal{V} s \rangle) (\lambda x. \text{the } (\mathcal{B}_l s x)) (\lambda x. \text{the } (\mathcal{B}_u s x)))$
 $(\mathcal{T} s) (\text{tvars } (\mathcal{T} s))$
let $?P' = \lambda v::(var, 'a) \text{ mapping.}$
 $\exists t. \text{tvars } t = \text{tvars } (\mathcal{T} s) \wedge \Delta t \wedge (\forall v::'a \text{ valuation. } v \models_t t = v \models_t \mathcal{T} s)$
 $\wedge \langle v \rangle \models_t t \wedge$
 $(\forall x \in \text{rvars } t. \langle v \rangle x = \langle \mathcal{V} s \rangle x \vee$
 $\langle v \rangle x = \text{the } (\mathcal{B}_l s x) \vee$
 $\langle v \rangle x = \text{the } (\mathcal{B}_u s x)) \wedge$
 $(\forall x. x \notin \text{tvars } (\mathcal{T} s) \longrightarrow \text{look } v x = \text{look } (\mathcal{V} s) x) \wedge$
 $(\forall x. x \in \text{tvars } (\mathcal{T} s) \longrightarrow \text{look } v x \neq \text{None})$

have $\text{finite } (\text{tvars } (\mathcal{T} s))$
by (auto simp add: lvars-def rvars-def finite-vars)

then have $\text{finite } \{v. ?P v\}$
using $\text{finite-valuations}[\text{of tvars } (\mathcal{T} s) \mathcal{T} s \langle \mathcal{V} s \rangle \lambda x. \text{the } (\mathcal{B}_l s x) \lambda x. \text{the } (\mathcal{B}_u s x)]$
by auto

moreover

have $\text{map2fun } \{v. ?P' v\} \subseteq \{v. ?P v\}$
by (auto simp add: map2fun-def)

ultimately

have $\text{finite } (\text{map2fun } \{v. ?P' v\})$
by (auto simp add: finite-subset)

moreover

```

have inj-on map2fun {v. ?P' v}
  unfolding inj-on-def
proof (safe)
  fix x y
  assume ⟨x⟩ = ⟨y⟩ and *:
    ∀ x. x ∉ Simplex.tvars (T s) → look y x = look (V s) x
    ∀ xa. xa ∉ Simplex.tvars (T s) → look x xa = look (V s) xa
    ∀ x. x ∈ Simplex.tvars (T s) → look y x ≠ None
    ∀ xa. xa ∈ Simplex.tvars (T s) → look x xa ≠ None
  show x = y
  proof (rule mapping-eqI)
    fix k
    have ⟨x⟩ k = ⟨y⟩ k
      using ⟨⟨x⟩ = ⟨y⟩⟩
      by simp
    then show look x k = look y k
      using *
      by (cases k ∈ tvars (T s)) (auto simp add: map2fun-def split: option.split)
  qed
qed
ultimately
have finite {v. ?P' v}
  by (rule finite-imageD)
moreover
have ?A ⊆ {v. ?P' v}
proof (safe)
  fix s'
  assume s ≻+ s'
  then obtain l where *: l ≠ [] 1 < length l hd l = s last l = s' succ-chain l
    using trancl-rel-chain[of s s' succ-rel]
    by auto
  show ?P' (V s')
proof-
  have ∇ s △ (T s) ⟨V s⟩ ⊨t T s
    using ⟨s ≻+ s'⟩
    using tranclD[of s s' succ-rel]
    by (auto simp add: curr-val-satisfies-no-lhs-def)
  have tvars (T s') = tvars (T s)
    using succ-chain-vars-id[of l 0 length l - 1]
    using * hd-conv-nth[of l] last-conv-nth[of l]
    by simp
  moreover
  have △(T s')
    using ⟨s ≻+ s'⟩
    using succ-inv(1)[of - s']
    by (auto dest: tranclD2)
  moreover
  have ∀ v::'a valuation. v ⊨t T s' = v ⊨t T s
    using succ-chain-tableau-equiv[of l 0 length l - 1]

```

```

using * hd-conv-nth[of l] last-conv-nth[of l]
by auto
moreover
have ⟨V s⟩ ⊨t T s'
  using succ-chain-validation-satisfies[of l 0 length l - 1]
  using * hd-conv-nth[of l] last-conv-nth[of l] ⟨⟨V s⟩ ⊨t T s⟩
  by simp
moreover
have ∀ x ∈ rvars (T s'). ⟨V s⟩ x = ⟨V s⟩ x ∨ ⟨V s⟩ x = the (Bl s x) ∨ ⟨V
s'⟩ x = the (Bu s x)
  using succ-chain-rvar-validation[of l 0 length l - 1]
  using * hd-conv-nth[of l] last-conv-nth[of l]
  by auto
moreover
have ∀ x. x ∉ tvars (T s) → look (V s') x = look (V s) x
  using succ-chain-no-vars-validation[of l 0 length l - 1]
  using * hd-conv-nth[of l] last-conv-nth[of l]
  by auto
moreover
have ∀ x. x ∈ Simplex.tvars (T s') → look (V s') x ≠ None
  using succ-chain-tableau-valuated[of l 0 length l - 1]
  using * hd-conv-nth[of l] last-conv-nth[of l]
  using ⟨tvars (T s') = tvars (T s)⟩ ⟨∇ s⟩
  by (auto simp add: tableau-valuated-def)
ultimately
show ?thesis
  by (rule-tac x=“T s’” in exI) auto
qed
qed
ultimately
show ?thesis
  by (auto simp add: finite-subset)
qed
ultimately
show ?thesis
  by simp
qed

lemma accessible-bounds:
  shows Bi ‘{s'. s ⊳* s'} = {Bi s}
proof –
  have s ⊳* s' ⟹ Bi s' = Bi s for s'
    by (induct s s' rule: rtrancl.induct, auto)
  then show ?thesis by blast
qed

lemma accessible-unsat-core:
  shows Uc ‘{s'. s ⊳* s'} = {Uc s}
proof –

```

```

have  $s \succ^* s' \Rightarrow \mathcal{U}_c s' = \mathcal{U}_c s$  for  $s'$ 
  by (induct  $s s'$  rule: rtrancl.induct, auto)
  then show ?thesis by blast
qed

lemma state-eqI:
 $\mathcal{B}_{il} s = \mathcal{B}_{il} s' \Rightarrow \mathcal{B}_{iu} s = \mathcal{B}_{iu} s' \Rightarrow$ 
 $\mathcal{T} s = \mathcal{T} s' \Rightarrow \mathcal{V} s = \mathcal{V} s' \Rightarrow$ 
 $\mathcal{U} s = \mathcal{U} s' \Rightarrow \mathcal{U}_c s = \mathcal{U}_c s' \Rightarrow$ 
 $s = s'$ 
by (cases  $s$ , cases  $s'$ , auto)

lemma finite-accessible-states:
shows finite { $s'. s \succ^* s'$ } (is finite ?A)
proof-
  let ?V =  $\mathcal{V} ` ?A$ 
  let ?T =  $\mathcal{T} ` ?A$ 
  let ?P = ?V × ?T × { $\mathcal{B}_i s$ } × {True, False} × { $\mathcal{U}_c s$ }
  have finite ?P
    using finite-accessible-valuations finite-accessible-tableaus
    by auto
  moreover
  let ?f =  $\lambda s. (\mathcal{V} s, \mathcal{T} s, \mathcal{B}_i s, \mathcal{U} s, \mathcal{U}_c s)$ 
  have ?f ` ?A ⊆ ?P
    using accessible-bounds[of  $s$ ] accessible-unsat-core[of  $s$ ]
    by auto
  moreover
  have inj-on ?f ?A
    unfolding inj-on-def by (auto intro: state-eqI)
  ultimately
  show ?thesis
    using finite-imageD [of ?f ?A]
    using finite-subset
    by auto
qed

lemma acyclic-suc-rel: acyclic succ-rel
proof (rule acyclicI, rule allI)
  fix s
  show  $(s, s) \notin \text{succ-rel}^+$ 
  proof
    assume  $s \succ^+ s$ 
    then obtain l where
       $l \neq []$  length  $l > 1$  hd  $l = s$  last  $l = s$  succ-chain  $l$ 
    using trancl-rel-chain[of  $s$  s succ-rel]
    by auto

  have  $l @ 0 = s$ 

```

```

using ⟨l ≠ []⟩ ⟨hd l = s⟩
by (simp add: hd-conv-nth)
then have s ∙ (l ! 1)
  using ⟨succ-chain l⟩
  unfolding rel-chain-def
  using ⟨length l > 1⟩
  by auto
then have △ (T s)
  by simp

let ?enter-rvars =
  {x. ∃ sl. swap-lr l sl x}

have finite ?enter-rvars
proof-
  let ?all-vars = ∪ (set (map (λ t. lvars t ∪ rvars t) (map T l)))
  have finite ?all-vars
    by (auto simp add: lvars-def rvars-def finite-vars)
  moreover
  have ?enter-rvars ⊆ ?all-vars
    by force
  ultimately
  show ?thesis
    by (simp add: finite-subset)
qed

let ?xr = Max ?enter-rvars
have ?xr ∈ ?enter-rvars
proof (rule Max-in)
  show ?enter-rvars ≠ {}
  proof-
    from ⟨s ∙ (l ! 1)⟩
    obtain xi xj :: var where
      xi ∈ lvars (T s) xi ∈ rvars (T (l ! 1))
      by (rule succ-vars) auto
    then have xi ∈ ?enter-rvars
      using ⟨hd l = s⟩ ⟨l ≠ []⟩ ⟨length l > 1⟩
      by (auto simp add: hd-conv-nth)
    then show ?thesis
      by auto
  qed
next
show finite ?enter-rvars
  using ⟨finite ?enter-rvars⟩
.
qed
then obtain xr sl where
  xr = ?xr swap-lr l sl xr
  by auto

```

```

then have  $sl + 1 < \text{length } l$ 
  by simp

have  $(l ! sl) \succ (l ! (sl + 1))$ 
  using  $\langle sl + 1 < \text{length } l \rangle, \langle \text{succ-chain } l \rangle$ 
  unfolding rel-chain-def
  by auto

have  $\text{length } l > 2$ 
proof (rule ccontr)
  assume  $\neg ?\text{thesis}$ 
  with  $\langle \text{length } l > 1 \rangle$ 
  have  $\text{length } l = 2$ 
    by auto
  then have  $\text{last } l = l ! 1$ 
    by (cases  $l$ ) (auto simp add: last-conv-nth nth-Cons split: nat.split)
  then have  $xr \in \text{lvards } (\mathcal{T} s) \quad xr \in \text{rvards } (\mathcal{T} s)$ 
    using  $\langle \text{length } l = 2 \rangle$ 
    using  $\langle \text{swap-lr } l \ sl \ xr \rangle$ 
    using  $\langle \text{hd } l = s \rangle, \langle \text{last } l = s \rangle, \langle l \neq [] \rangle$ 
    by (auto simp add: hd-conv-nth)
  then show False
    using  $\langle \triangle (\mathcal{T} s) \rangle$ 
    unfolding normalized-tableau-def
    by auto
qed

obtain  $l'$  where
   $\text{hd } l' = l ! (sl + 1)$ 
   $\text{last } l' = l ! sl$ 
   $\text{length } l' = \text{length } l - 1$ 
   $\text{succ-chain } l' \text{ and}$ 
   $\forall i. i + 1 < \text{length } l' \longrightarrow$ 
   $(\exists j. j + 1 < \text{length } l \wedge l' ! i = l ! j \wedge l' ! (i + 1) = l ! (j + 1))$ 
  using  $\langle \text{length } l > 2 \rangle, \langle sl + 1 < \text{length } l \rangle, \langle \text{hd } l = s \rangle, \langle \text{last } l = s \rangle, \langle \text{succ-chain } l \rangle$ 
  using reorder-cyclic-list[of l s sl]
  by blast

then have  $xr \in \text{rvards } (\mathcal{T} (\text{hd } l')) \quad xr \in \text{lvards } (\mathcal{T} (\text{last } l'))$ 
   $\text{length } l' > 1 \quad l' \neq []$ 
  using  $\langle \text{swap-lr } l \ sl \ xr \rangle, \langle \text{length } l > 2 \rangle$ 
  by auto

then have  $\exists sp. \text{swap-rl } l' sp \ xr$ 
  using  $\langle \text{succ-chain } l' \rangle$ 
  using succ-chain-swap-rl-exists[of l' 0 length l' - 1 xr]
  by (auto simp add: hd-conv-nth last-conv-nth)
then have  $\exists sp. \text{swap-rl } l' sp \ xr \wedge (\forall sp'. sp' < sp \longrightarrow \neg \text{swap-rl } l' sp' xr)$ 
  by (rule min-element)
then obtain  $sp$  where
   $\text{swap-rl } l' sp \ xr \quad \forall sp'. sp' < sp \longrightarrow \neg \text{swap-rl } l' sp' xr$ 
  by blast

```

```

then have sp + 1 < length l'
  by simp

have ⟨V (l' ! 0)⟩ xr = ⟨V (l' ! sp)⟩ xr
proof -
  have always-r l' 0 sp xr
    using ⟨xr ∈ rvars (T (hd l'))⟩ ⟨sp + 1 < length l'⟩
      ⟨∀ sp'. sp' < sp → swap-rl l' sp' xr⟩
  proof (induct sp)
    case 0
      then have l' ≠ []
        by auto
      then show ?case
        using 0(1)
        by (auto simp add: hd-conv-nth)
  next
    case (Suc sp')
    show ?case
    proof (safe)
      fix k
      assume k ≤ Suc sp'
      show xr ∈ rvars (T (l' ! k))
      proof (cases k = sp' + 1)
        case False
          then show ?thesis
            using Suc ⟨k ≤ Suc sp'⟩
            by auto
        next
          case True
          then have xr ∈ rvars (T (l' ! (k - 1)))
            using Suc
            by auto
        moreover
          then have xr ∉ lvars (T (l' ! k))
            using True Suc(3) Suc(4)
            by auto
        moreover
          have (l' ! (k - 1)) ≻ (l' ! k)
            using ⟨succ-chain l'⟩
            using Suc(3) True
            by (simp add: rel-chain-def)
        ultimately
          show ?thesis
            using succ-vars-id[of l' ! (k - 1) l' ! k]
            by auto
        qed
      qed
    qed
    then show ?thesis

```

```

using ⟨sp + 1 < length l'⟩
using ⟨succ-chain l'⟩
using succ-chain-always-r-valuation-id
by simp
qed

have (l' ! sp) ⊢ (l' ! (sp+1))
using ⟨sp + 1 < length l'⟩ ⟨succ-chain l'⟩
unfolding rel-chain-def
by simp
then obtain xs xr' :: var where
  xs ∈ lvars (T (l' ! sp))
  xr ∈ rvars (T (l' ! sp))
  swap-lr l' sp xs
  apply (rule succ-vars)
  using ⟨swap-rl l' sp xr⟩ ⟨sp + 1 < length l'⟩
  by auto
then have xs ≠ xr
  using ⟨(l' ! sp) ⊢ (l' ! (sp+1))⟩
  by (auto simp add: normalized-tableau-def)

obtain sp' where
  l' ! sp = l ! sp' l' ! (sp + 1) = l ! (sp' + 1)
  sp' + 1 < length l
  using ⟨sp + 1 < length l'⟩ l'-l
  by auto

have xs ∈ ?enter-rvars
using ⟨swap-lr l' sp xs⟩ l'-l
by force

have xs < xr
proof -
  have xs ≤ ?xr
  using ⟨finite ?enter-rvars⟩ ⟨xs ∈ ?enter-rvars⟩
  by (rule Max-ge)
  then show ?thesis
  using ⟨xr = ?xr⟩ ⟨xs ≠ xr⟩
  by simp
qed

let ?sl = l ! sl
let ?sp = l' ! sp
let ?eq = eq-for-lvar (T ?sp) xs
let ?bl = V ?sl
let ?bp = V ?sp

have ⊨_nolhs ?sl ⊨_nolhs ?sp
using ⟨l ! sl ⊢ (l ! (sl + 1))⟩

```

using $\langle l' ! sp \succ \succ (l' ! (sp + 1))$
by simp-all

have $\mathcal{B}_i ?sp = \mathcal{B}_i ?sl$

proof –

have $\mathcal{B}_i (l' ! sp) = \mathcal{B}_i (l' ! (\text{length } l' - 1))$
using $\langle sp + 1 < \text{length } l' \rangle \langle \text{succ-chain } l' \rangle$
using succ-chain-bounds-id
by auto
then have $\mathcal{B}_i (\text{last } l') = \mathcal{B}_i (l' ! sp)$
using $\langle l' \neq [] \rangle$
by (simp add: last-conv-nth)
then show ?thesis
using $\langle \text{last } l' = l ! sl \rangle$
by simp
qed

have diff-satisfied: $\langle ?bl \rangle xs - \langle ?bp \rangle xs = ((rhs ?eq) \setminus \langle ?bl \rangle \setminus) - ((rhs ?eq) \setminus \langle ?bp \rangle \setminus)$

proof –

have $\langle ?bp \rangle \models_e ?eq$
using $\langle \models_{\text{no-lhs}} ?sp \rangle$
using eq-for-lvar[of xs $\mathcal{T} ?sp$]
using $\langle xs \in \text{lvars} (\mathcal{T} (l' ! sp)) \rangle$
unfolding curr-val-satisfies-no-lhs-def satisfies-tableau-def
by auto

moreover

have $\langle ?bl \rangle \models_e ?eq$

proof –

have $\langle \mathcal{V} (l ! sl) \rangle \models_t \mathcal{T} (l' ! sp)$
using $\langle l' ! sp = l ! sp' \rangle \langle sp' + 1 < \text{length } l \rangle \langle sl + 1 < \text{length } l \rangle$
using succ-chain l
using succ-chain-tableau-equiv[of l sl sp]
using $\langle \models_{\text{no-lhs}} ?sl \rangle$
unfolding curr-val-satisfies-no-lhs-def
by simp

then show ?thesis

unfolding satisfies-tableau-def

using eq-for-lvar

using $\langle xs \in \text{lvars} (\mathcal{T} (l' ! sp)) \rangle$

by simp

qed

moreover

have $\text{lhs} ?eq = xs$

using $\langle xs \in \text{lvars} (\mathcal{T} (l' ! sp)) \rangle$

using eq-for-lvar

by simp

ultimately

show ?thesis

```

unfolding satisfies-eq-def
  by auto
qed

have  $\neg \text{in-bounds } xr \langle ?bl \rangle (\mathcal{B} ?sl)$ 
  using  $\langle l ! sl \succ (l ! (sl + 1)) \rangle \langle \text{swap-lr } l sl xr \rangle$ 
  using succ-min-lvar-not-in-bounds(1)[of ?sl l ! (sl + 1) xr]
  by simp

have  $\forall x. x < xr \longrightarrow \text{in-bounds } x \langle ?bl \rangle (\mathcal{B} ?sl)$ 
proof (safe)
  fix x
  assume  $x < xr$ 
  show  $\text{in-bounds } x \langle ?bl \rangle (\mathcal{B} ?sl)$ 
  proof (cases x ∈ lvars (T ?sl))
    case True
      then show ?thesis
        using succ-min-lvar-not-in-bounds(2)[of ?sl l ! (sl + 1) xr]
        using  $\langle l ! sl \succ (l ! (sl + 1)) \rangle \langle \text{swap-lr } l sl xr \rangle \langle x < xr \rangle$ 
        by simp
    next
      case False
      then show ?thesis
        using  $\models_{\text{nolhs}} ?sl$ 
        unfolding curr-val-satisfies-no-lhs-def
        by (simp add: satisfies-bounds-set.simps)
  qed
qed

then have  $\text{in-bounds } xs \langle ?bl \rangle (\mathcal{B} ?sl)$ 
  using  $\langle xs < xr \rangle$ 
  by simp

have  $\neg \text{in-bounds } xs \langle ?bp \rangle (\mathcal{B} ?sp)$ 
  using  $\langle l' ! sp \succ (l' ! (sp + 1)) \rangle \langle \text{swap-lr } l' sp xs \rangle$ 
  using succ-min-lvar-not-in-bounds(1)[of ?sp l' ! (sp + 1) xs]
  by simp

have  $\forall x \in \text{rvars-eq } ?eq. x > xr \longrightarrow \langle ?bp \rangle x = \langle ?bl \rangle x$ 
proof (safe)
  fix x
  assume  $x \in \text{rvars-eq } ?eq$ 
  then have always-r l' 0 (length l' - 1) x
  proof (safe)
    fix k
    assume  $x \in \text{rvars-eq } ?eq$ 
     $x > xr$ 
     $0 \leq k$ 
     $k \leq \text{length } l' - 1$ 
    obtain k' where  $l ! k' = l' ! k$ 
     $k' < \text{length } l$ 
    using l'-l ⟨k ≤ length l' - 1⟩ ⟨length l' > 1⟩
    apply (cases k > 0)

```

```

apply (erule-tac x=k - 1 in allE)
apply (drule mp)
by auto

let ?eq' = eq-for-lvar (T (l ! sp')) xs

have ∀ x ∈ rvars-eq ?eq'. x > xr → always-r l 0 (length l - 1) x
proof (safe)
fix x k
assume x ∈ rvars-eq ?eq' xr < x 0 ≤ k k ≤ length l - 1
then have x ∈ rvars (T (l ! sp'))
using eq-for-lvar[of xs T (l ! sp')]
using ⟨swap-lr l' sp xs⟩ ⟨l' ! sp = l ! sp'⟩
by (auto simp add: rvars-def)
have *: ∀ i. i < sp' → x ∈ rvars (T (l ! i))
proof (safe, rule ccontr)
fix i
assume i < sp' x ∉ rvars (T (l ! i))
then have x ∈ lvars (T (l ! i))
using ⟨x ∈ rvars (T (l ! sp'))⟩
using ⟨sp' + 1 < length l⟩
using ⟨succ-chain l⟩
using succ-chain-vars-id[of l i sp']
by auto
obtain i' where swap-lr l i' x
using ⟨x ∈ lvars (T (l ! i))⟩
using ⟨x ∈ rvars (T (l ! sp'))⟩
using ⟨i < sp'⟩ ⟨sp' + 1 < length l⟩
using ⟨succ-chain l⟩
using succ-chain-swap-lr-exists[of l i sp' x]
by auto
then have x ∈ ?enter-rvars
by auto
then have x ≤ ?xr
using ⟨finite ?enter-rvars⟩
using Max-ge[of ?enter-rvars x]
by simp
then show False
using ⟨x > xr⟩
using ⟨xr = ?xr⟩
by simp
qed

then have x ∈ rvars (T (last l))
using ⟨hd l = s⟩ ⟨last l = s⟩ ⟨l ≠ []⟩
using ⟨x ∈ rvars (T (l ! sp'))⟩
by (auto simp add: hd-conv-nth)

show x ∈ rvars (T (l ! k))

```

```

proof (cases  $k = \text{length } l - 1$ )
  case True
    then show ?thesis
      using  $\langle x \in \text{rvars}(\mathcal{T}(\text{last } l)) \rangle$ 
      using  $\langle l \neq [] \rangle$ 
      by (simp add: last-conv-nth)
  next
    case False
    then have  $k < \text{length } l - 1$ 
      using  $\langle k \leq \text{length } l - 1 \rangle$ 
      by simp
    then have  $k < \text{length } l$ 
      using  $\langle \text{length } l > 1 \rangle$ 
      by auto
    show ?thesis
    proof (rule ccontr)
      assume  $\neg$  ?thesis
      then have  $x \in \text{lvars}(\mathcal{T}(l ! k))$ 
      using  $\langle x \in \text{rvars}(\mathcal{T}(l ! sp')) \rangle$ 
      using  $\langle sp' + 1 < \text{length } l \rangle \langle k < \text{length } l \rangle$ 
      using succ-chain-vars-id[of  $l k sp'$ ]
      using  $\langle \text{succ-chain } l \rangle \langle l \neq [] \rangle$ 
      by auto
      obtain  $i'$  where swap-lr  $l i' x$ 
      using  $\langle \text{succ-chain } l \rangle$ 
      using  $\langle x \in \text{lvars}(\mathcal{T}(l ! k)) \rangle$ 
      using  $\langle x \in \text{rvars}(\mathcal{T}(\text{last } l)) \rangle$ 
      using  $\langle k < \text{length } l - 1 \rangle \langle l \neq [] \rangle$ 
      using succ-chain-swap-lr-exists[of  $l k \text{length } l - 1 x$ ]
      by (auto simp add: last-conv-nth)
      then have  $x \in \text{?enter-rvars}$ 
      by auto
      then have  $x \leq \text{?xr}$ 
      using finite ?enter-rvars
      using Max-ge[of ?enter-rvars  $x$ ]
      by simp
      then show False
      using  $\langle x > \text{?xr} \rangle$ 
      using  $\langle \text{?xr} = \text{?xr} \rangle$ 
      by simp
    qed
    qed
    qed
    then have  $x \in \text{rvars}(\mathcal{T}(l ! k'))$ 
    using  $\langle x \in \text{rvars-eq } ?eq \rangle \langle x > \text{?xr} \rangle \langle k' < \text{length } l \rangle$ 
    using  $\langle l' ! sp = l ! sp' \rangle$ 
    by simp
  then show  $x \in \text{rvars}(\mathcal{T}(l' ! k))$ 

```

```

    using  $\langle l ! k' = l' ! k \rangle$ 
    by simp
qed
then have  $\langle ?bp \rangle x = \langle \mathcal{V}(l' ! (length l' - 1)) \rangle x$ 
    using  $\langle succ-chain l' \rangle \langle sp + 1 < length l' \rangle$ 
    by (auto intro!: succ-chain-always-r-valuation-id[rule-format])
then have  $\langle ?bp \rangle x = \langle \mathcal{V}(\text{last } l') \rangle x$ 
    using  $\langle l' \neq [] \rangle$ 
    by (simp add: last-conv-nth)
then show  $\langle ?bp \rangle x = \langle ?bl \rangle x$ 
    using  $\langle \text{last } l' = l ! sl \rangle$ 
    by simp
qed

have  $\langle ?bp \rangle xr = \langle \mathcal{V}(l ! (sl + 1)) \rangle xr$ 
    using  $\langle \mathcal{V}(l' ! 0) \rangle xr = \langle \mathcal{V}(l' ! sp) \rangle xr$ 
    using  $\langle hd l' = l ! (sl + 1) \rangle \langle l' \neq [] \rangle$ 
    by (simp add: hd-conv-nth)

{

fix dir1 dir2 :: ('i,'a) Direction
assume dir1:  $dir1 = (\text{if } \langle ?bl \rangle xr <_{lb} \mathcal{B}_l ?sl xr \text{ then Positive else Negative})$ 
then have  $\triangleleft_{lb} (lt dir1) (\langle ?bl \rangle xr) (LB dir1 ?sl xr)$ 
    using  $\neg \text{in-bounds } xr \langle ?bl \rangle (\mathcal{B} ?sl)$ 
    using neg-bounds-compare(7) neg-bounds-compare(3)
    by (auto simp add: bound-compare''-defs)
then have  $\neg \triangleright_{lb} (lt dir1) (\langle ?bl \rangle xr) (LB dir1 ?sl xr)$ 
    using bounds-compare-contradictory(7) bounds-compare-contradictory(3)
neg-bounds-compare(6) dir1
    unfolding bound-compare''-defs
    by auto force
have LB dir1 ?sl xr ≠ None
    using  $\triangleleft_{lb} (lt dir1) (\langle ?bl \rangle xr) (LB dir1 ?sl xr)$ 
    by (cases LB dir1 ?sl xr) (auto simp add: bound-compare-defs)

assume dir2:  $dir2 = (\text{if } \langle ?bp \rangle xs <_{lb} \mathcal{B}_l ?sp xs \text{ then Positive else Negative})$ 
then have  $\triangleleft_{lb} (lt dir2) (\langle ?bp \rangle xs) (LB dir2 ?sp xs)$ 
    using  $\neg \text{in-bounds } xs \langle ?bp \rangle (\mathcal{B} ?sp)$ 
    using neg-bounds-compare(2) neg-bounds-compare(6)
    by (auto simp add: bound-compare''-defs)
then have  $\neg \triangleright_{lb} (lt dir2) (\langle ?bp \rangle xs) (LB dir2 ?sp xs)$ 
    using bounds-compare-contradictory(3) bounds-compare-contradictory(7)
neg-bounds-compare(6) dir2
    unfolding bound-compare''-defs
    by auto force
then have  $\forall x \in rvars-eq ?eq. x < xr \longrightarrow \neg \text{reasable-var } dir2 x ?eq ?sp$ 
    using succ-min-rvar[of ?sp l' ! (sp + 1) xs xr ?eq]
    using  $\langle l' ! sp \succ (l' ! (sp + 1)) \rangle$ 
    using swap-lr l' sp xs swap-rl l' sp xr dir2

```

```

unfolding bound-compare''-defs
by auto

have LB dir2 ?sp xs ≠ None
  using ⟨ $\triangleleft_{lb}$  (lt dir2) ((?bp) xs) (LB dir2 ?sp xs)⟩
  by (cases LB dir2 ?sp xs) (auto simp add: bound-compare-defs)

have *:  $\forall x \in rvars\text{-eq} \ ?eq. \ x < xr \longrightarrow$ 
   $((coeff \ (rhs \ ?eq) \ x > 0 \longrightarrow \triangleleft_{ub} \ (lt \ dir2) \ ((?bp) \ x) \ (UB \ dir2 \ ?sp \ x)) \wedge$ 
   $(coeff \ (rhs \ ?eq) \ x < 0 \longrightarrow \triangleleft_{lb} \ (lt \ dir2) \ ((?bp) \ x) \ (LB \ dir2 \ ?sp \ x)))$ 
proof (safe)
  fix x
  assume  $x \in rvars\text{-eq} \ ?eq \ x < xr \ coeff \ (rhs \ ?eq) \ x > 0$ 
  then have  $\neg \triangleleft_{ub} \ (lt \ dir2) \ ((?bp) \ x) \ (UB \ dir2 \ ?sp \ x)$ 
    using ⟨ $\forall x \in rvars\text{-eq} \ ?eq. \ x < xr \longrightarrow \neg \text{reasable-var} \ dir2 \ x \ ?eq \ ?sp$ ⟩
    by simp
  then show  $\triangleleft_{ub} \ (lt \ dir2) \ ((?bp) \ x) \ (UB \ dir2 \ ?sp \ x)$ 
    using dir2 neg-bounds-compare(4) neg-bounds-compare(8)
    unfolding bound-compare''-defs
    by force
next
  fix x
  assume  $x \in rvars\text{-eq} \ ?eq \ x < xr \ coeff \ (rhs \ ?eq) \ x < 0$ 
  then have  $\neg \triangleleft_{lb} \ (lt \ dir2) \ ((?bp) \ x) \ (LB \ dir2 \ ?sp \ x)$ 
    using ⟨ $\forall x \in rvars\text{-eq} \ ?eq. \ x < xr \longrightarrow \neg \text{reasable-var} \ dir2 \ x \ ?eq \ ?sp$ ⟩
    by simp
  then show  $\triangleleft_{lb} \ (lt \ dir2) \ ((?bp) \ x) \ (LB \ dir2 \ ?sp \ x)$ 
    using dir2 neg-bounds-compare(4) neg-bounds-compare(8) dir2
    unfolding bound-compare''-defs
    by force
qed

have (lt dir2) ((?bp) xs) ((?bl) xs)
  using ⟨ $\triangleleft_{lb}$  (lt dir2) ((?bp) xs) (LB dir2 ?sp xs)⟩
  using ⟨ $\mathcal{B}_i \ ?sp = \mathcal{B}_i \ ?sl \ dir2$ ⟩
  using ⟨in-bounds xs (?bl) ( $\mathcal{B} \ ?sl$ )⟩
  by (auto simp add: bound-compare''-defs
    simp: indexl-def indexu-def boundsl-def boundsu-def)
then have (lt dir2) 0 ((?bl) xs - (?bp) xs)
  using dir2
  by (auto simp add: minus-gt[THEN sym] minus-lt[THEN sym])

moreover

have le (lt dir2) ((rhs ?eq) {?bl}) - (rhs ?eq) {?bp} 0
proof-
  have  $\forall x \in rvars\text{-eq} \ ?eq. \ (0 < coeff \ (rhs \ ?eq) \ x \longrightarrow le \ (lt \ dir2) \ 0 \ ((?bp) \ x$ 
   $- (?bl) \ x)) \wedge$ 
     $(coeff \ (rhs \ ?eq) \ x < 0 \longrightarrow le \ (lt \ dir2) \ ((?bp) \ x - (?bl) \ x) \ 0)$ 

```

```

proof
  fix  $x$ 
  assume  $x \in rvars\text{-}eq ?eq$ 
  show  $(0 < coeff (rhs ?eq) x \longrightarrow le (lt dir2) 0 (\langle ?bp \rangle x - \langle ?bl \rangle x)) \wedge$ 
     $(coeff (rhs ?eq) x < 0 \longrightarrow le (lt dir2) (\langle ?bp \rangle x - \langle ?bl \rangle x) 0)$ 
  proof (cases  $x < xr$ )
    case True
      then have in-bounds  $x \langle ?bl \rangle (\mathcal{B} ?sl)$ 
      using  $\langle \forall x. x < xr \longrightarrow \text{in-bounds } x \langle ?bl \rangle (\mathcal{B} ?sl) \rangle$ 
      by simp
      show ?thesis
      proof (safe)
        assume  $coeff (rhs ?eq) x > 0 0 \neq \langle ?bp \rangle x - \langle ?bl \rangle x$ 
        then have  $\triangleleft_{ub} (lt dir2) (\langle \mathcal{V} (l' ! sp) \rangle x) (UB dir2 (l' ! sp) x)$ 
        using *  $\langle x < xr \rangle \langle x \in rvars\text{-}eq ?eq \rangle$ 
        by simp
        then have  $le (lt dir2) (\langle ?bl \rangle x) (\langle ?bp \rangle x)$ 
        using  $\langle \text{in-bounds } x \langle ?bl \rangle (\mathcal{B} ?sl) \rangle \langle \mathcal{B}_i ?sp = \mathcal{B}_i ?sl \rangle \text{ dir2}$ 
        apply (auto simp add: bound-compare''-defs)
        using bounds-lg(3)[of ?bp x B_u (l ! sl) x ?bl x]
        using bounds-lg(6)[of ?bp x B_l (l ! sl) x ?bl x]
        unfolding bound-compare''-defs
        by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)
        then show  $lt dir2 0 (\langle ?bp \rangle x - \langle ?bl \rangle x)$ 
        using  $\langle 0 \neq \langle ?bp \rangle x - \langle ?bl \rangle x \rangle$ 
        using minus-gt[of ?bl x ?bp x] minus-lt[of ?bp x ?bl x] dir2
        by (auto simp del: Simplex.bounds-lg)
    next
      assume  $coeff (rhs ?eq) x < 0 \langle ?bp \rangle x - \langle ?bl \rangle x \neq 0$ 
      then have  $\trianglelefteq_{lb} (lt dir2) (\langle \mathcal{V} (l' ! sp) \rangle x) (LB dir2 (l' ! sp) x)$ 
      using *  $\langle x < xr \rangle \langle x \in rvars\text{-}eq ?eq \rangle$ 
      by simp
      then have  $le (lt dir2) (\langle ?bp \rangle x) (\langle ?bl \rangle x)$ 
      using  $\langle \text{in-bounds } x \langle ?bl \rangle (\mathcal{B} ?sl) \rangle \langle \mathcal{B}_i ?sp = \mathcal{B}_i ?sl \rangle \text{ dir2}$ 
      apply (auto simp add: bound-compare''-defs)
      using bounds-lg(3)[of ?bp x B_u (l ! sl) x ?bl x]
      using bounds-lg(6)[of ?bp x B_l (l ! sl) x ?bl x]
      unfolding bound-compare''-defs
      by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)
      then show  $lt dir2 (\langle ?bp \rangle x - \langle ?bl \rangle x) 0$ 
      using  $\langle \langle ?bp \rangle x - \langle ?bl \rangle x \neq 0 \rangle$ 
      using minus-gt[of ?bl x ?bp x] minus-lt[of ?bp x ?bl x] dir2
      by (auto simp del: Simplex.bounds-lg)
    qed
  next
    case False
    show ?thesis
    proof (cases  $x = xr$ )
      case True

```

```

have ⟨V (l ! (sl + 1))⟩ xr = the (LB dir1 ?sl xr)
  using ⟨l ! sl ∙ (l ! (sl + 1))⟩
  using ⟨swap-lr l sl xr⟩
  using succ-set-on-bound(1)[of l ! sl l ! (sl + 1) xr]
  using ⟨¬ ⊇_{lb} (lt dir1) (?bl) xr (LB dir1 ?sl xr)⟩ dir1
  unfolding bound-compare"-defs
  by auto
then have ?bp xr = the (LB dir1 ?sl xr)
  using ⟨?bp⟩ xr = ⟨V (l ! (sl + 1))⟩ xr
  by simp
then have lt dir1 (?bl) xr (?bp) xr
  using ⟨LB dir1 ?sl xr ≠ None⟩
  using ⟨¬ ⊲_{lb} (lt dir1) (?bl) xr (LB dir1 ?sl xr)⟩ dir1
  by (auto simp add: bound-compare-defs)

```

moreover

```

have reusable-var dir2 xr ?eq ?sp
  using ⟨¬ ⊇_{lb} (lt dir2) (?bp) xs (LB dir2 ?sp xs)⟩
  using ⟨l' ! sp ∙ (l' ! (sp + 1))⟩
  using ⟨swap-lr l' sp xs⟩ ⟨swap-rl l' sp xr⟩
  using succ-min-rvar[of l' ! sp l' ! (sp + 1)xs xr ?eq] dir2
  unfolding bound-compare"-defs
  by auto

```

```

then have if dir1 = dir2 then coeff (rhs ?eq) xr > 0 else coeff (rhs
?eq) xr < 0
  using ⟨?bp⟩ xr = the (LB dir1 ?sl xr)⟩
  using ⟨B_i ?sp = B_i ?sl⟩ [THEN sym] dir1
  using ⟨LB dir1 ?sl xr ≠ None⟩ dir1 dir2
  by (auto split: if-splits simp add: bound-compare-defs
    indexl-def indexu-def boundsl-def boundsu-def)

```

moreover

```

have dir1 = Positive ∨ dir1 = Negative dir2 = Positive ∨ dir2 =
Negative
  using dir1 dir2
  by auto

```

ultimately

show ?thesis

```

  using ⟨x = xr⟩
  using minus-lt[of ⟨?bp⟩ xr ⟨?bl⟩ xr] minus-gt[of ⟨?bl⟩ xr ⟨?bp⟩ xr]
  by (auto split: if-splits simp del: Simplex.bounds-lg)

```

next

case False

then have x > xr

```

  using ⟨¬ x < xr⟩
  by simp

```

then have ?bp x = ?bl x

```

  using ⟨∀ x ∈ rvars-eq ?eq. x > xr ⟶ ⟨?bp⟩ x = ⟨?bl⟩ x⟩

```

```

using ⟨x ∈ rvars-eq ?eq⟩
by simp
then show ?thesis
by simp
qed
qed
qed
then have le (lt dir2) 0 (rhs ?eq {λ x. ⟨?bp⟩ x − ⟨?bl⟩ x })
using dir2
apply auto
using valuate-nonneg[of rhs ?eq λ x. ⟨?bp⟩ x − ⟨?bl⟩ x]
apply (force simp del: Simplex.bounds-lg)
using valuate-nonpos[of rhs ?eq λ x. ⟨?bp⟩ x − ⟨?bl⟩ x]
apply (force simp del: Simplex.bounds-lg)
done
then have le (lt dir2) 0 ((rhs ?eq) {⟨?bp⟩} − (rhs ?eq) {⟨?bl⟩})
by (subst valuate-diff)+ simp
then have le (lt dir2) ((rhs ?eq) {⟨?bl⟩}) ((rhs ?eq) {⟨?bp⟩})
using minus-lt[of (rhs ?eq) {⟨?bp⟩} (rhs ?eq) {⟨?bl⟩}] dir2
by (auto simp del: Simplex.bounds-lg)
then show ?thesis
using dir2
using minus-lt[of (rhs ?eq) {⟨?bl⟩} (rhs ?eq) {⟨?bp⟩}]
using minus-gt[of (rhs ?eq) {⟨?bp⟩} (rhs ?eq) {⟨?bl⟩}]
by (auto simp del: Simplex.bounds-lg)
qed
ultimately
have False
using diff-satisfied dir2
by (auto split: if-splits simp del: Simplex.bounds-lg)
}
then show False
by auto
qed
qed

```

```

lemma check-unsat-terminates:
assumes U s
shows check-dom s
by (rule check-dom.intros) (auto simp add: assms)

lemma check-sat-terminates'-aux:
assumes
dir: dir = (if ⟨V s⟩ xi <_l B_l s xi then Positive else Negative) and
*: ⋀ s'. [s ⊢ s'; ∇ s'; △ (T s'); ◇ s'; ⊨_nolhs s' ] ⟹ check-dom s' and
∇ s △ (T s) ◇ s ⊨_nolhs s
¬ U s min-lvar-not-in-bounds s = Some xi

```

```

 $\triangleleft_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i)$ 
shows check-dom
  (case min-rvar-incdec dir s x_i of Inl I  $\Rightarrow$  set-unsat I s
   | Inr x_j  $\Rightarrow$  pivot-and-update x_i x_j (the (LB dir s x_i)) s)
proof (cases min-rvar-incdec dir s x_i)
  case Inl
  then show ?thesis
    using check-unsat-terminates by simp
next
  case (Inr x_j)
  then have x_j:  $x_j \in rvars\text{-of-lvar } (\mathcal{T} s) x_i$ 
    using min-rvar-incdec-eq-Some-rvars[of - s eq-for-lvar ( $\mathcal{T} s$ ) x_i x_j]
    using dir
    by simp
  let ?s' = pivot-and-update x_i x_j (the (LB dir s x_i)) s
  have check-dom ?s'
  proof (rule *)
    show **:  $\nabla ?s' \Delta (\mathcal{T} ?s') \Diamond ?s' \models_{nolhs} ?s'$ 
      using <min-lvar-not-in-bounds s = Some x_i> Inr
      using < $\nabla s$ > < $\Delta (\mathcal{T} s)$ > < $\Diamond s$ > < $\models_{nolhs} s$ > dir
      using pivotandupdate-check-precond
      by auto
    have xi:  $x_i \in lvars (\mathcal{T} s)$ 
      using assms(8) min-lvar-not-in-bounds-lvars by blast
    show s  $\succ$  ?s'
      unfolding gt-state-def
      using < $\Delta (\mathcal{T} s)$ > < $\Diamond s$ > < $\models_{nolhs} s$ > < $\nabla s$ >
      using <min-lvar-not-in-bounds s = Some x_i> < $\triangleleft_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i)$ >
      Inr dir
      by (intro conjI pivotandupdate-bounds-id pivotandupdate-unsat-core-id,
           auto intro!: x_j xi)
    qed
    then show ?thesis using Inr by simp
  qed

lemma check-sat-terminates':
  assumes  $\nabla s \Delta (\mathcal{T} s) \Diamond s \models_{nolhs} s\ s_0 \succ^* s$ 
  shows check-dom s
  using assms
proof (induct s rule: wf-induct[of {(y, x). s_0  $\succ^*$  x  $\wedge$  x  $\succ$  y}])
  show wf {(y, x). s_0  $\succ^*$  x  $\wedge$  x  $\succ$  y}
  proof (rule finite-acyclic-wf)
    let ?A = {(s', s). s_0  $\succ^*$  s  $\wedge$  s  $\succ$  s'}
    let ?B = {s. s_0  $\succ^*$  s}
    have ?A  $\subseteq$  ?B  $\times$  ?B
  proof
    fix p
    assume p  $\in$  ?A

```

```

then have  $\text{fst } p \in ?B$   $\text{snd } p \in ?B$ 
  using  $\text{rtrancl-into-trancl1}[\text{of } s_0 \text{ snd } p \text{ succ-rel } \text{fst } p]$ 
  by auto
then show  $p \in ?B \times ?B$ 
  using  $\text{mem-Sigma-iff}[\text{of } \text{fst } p \text{ snd } p]$ 
  by auto
qed
then show  $\text{finite } ?A$ 
  using  $\text{finite-accessible-states}[\text{of } s_0]$ 
  using  $\text{finite-subset}[\text{of } ?A \text{ } ?B \times ?B]$ 
  by simp

show  $\text{acyclic } ?A$ 
proof-
  have  $?A \subseteq \text{succ-rel}^{-1}$ 
  by auto
then show  $?thesis$ 
  using  $\text{acyclic-converse acyclic-subset}$ 
  using  $\text{acyclic-suc-rel}$ 
  by auto
qed
qed
next
fix  $s$ 
assume  $\forall s'. (s', s) \in \{(y, x). s_0 \succ^* x \wedge x \succ y\} \longrightarrow \nabla s' \longrightarrow \Delta (\mathcal{T} s') \longrightarrow \Diamond$ 
 $s' \longrightarrow \models_{\text{nolhs}} s' \longrightarrow s_0 \succ^* s' \longrightarrow \text{check-dom } s'$ 
 $\nabla s \Delta (\mathcal{T} s) \Diamond s \models_{\text{nolhs}} s_0 \succ^* s$ 
then have  $*: \bigwedge s'. [s \succ s'; \nabla s'; \Delta (\mathcal{T} s'); \Diamond s'; \models_{\text{nolhs}} s'] \implies \text{check-dom } s'$ 
  using  $\text{rtrancl-into-trancl1}[\text{of } s_0 \text{ } s \text{ succ-rel}]$ 
  using  $\text{trancl-into-rtrancl}[\text{of } s_0 \text{ - succ-rel}]$ 
  by auto
show  $\text{check-dom } s$ 
proof (rule  $\text{check-dom.intros}$ ,  $\text{simp-all add: check'-def, unfold Positive-def[symmetric]}$ ,
 $\text{unfold Negative-def[symmetric]}$ )
  fix  $x_i$ 
  assume  $\neg \mathcal{U} s \text{ Some } x_i = \text{min-lvar-not-in-bounds } s \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i$ 
  have  $\mathcal{B}_l s x_i = LB \text{ Positive } s x_i$ 
  by simp
show  $\text{check-dom}$ 
  (case  $\text{min-rvar-incdec Positive } s x_i$  of
    Inl  $I \Rightarrow \text{set-unsat } I s$ 
    | Inr  $x_j \Rightarrow \text{pivot-and-update } x_i x_j (\text{the } (\mathcal{B}_l s x_i)) s$ 
  apply (subst  $\langle \mathcal{B}_l s x_i = LB \text{ Positive } s x_i \rangle$ )
  apply (rule  $\text{check-sat-terminates'-aux}[\text{of Positive } s x_i]$ )
  using  $\langle \nabla s \rangle \langle \Delta (\mathcal{T} s) \rangle \langle \Diamond s \rangle \langle \models_{\text{nolhs}} s \rangle *$ 
  using  $\langle \neg \mathcal{U} s \rangle \langle \text{Some } x_i = \text{min-lvar-not-in-bounds } s \rangle \langle \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i \rangle$ 
  by (simp-all add:  $\text{bound-compare''-defs}$ )
next
fix  $x_i$ 

```

```

assume  $\neg \mathcal{U} s \text{ Some } x_i = \text{min-lvar-not-in-bounds } s \neg \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i$ 
then have  $\langle \mathcal{V} s \rangle x_i >_{ub} \mathcal{B}_u s x_i$ 
  using min-lvar-not-in-bounds-Some[of s xi]
  using neg-bounds-compare(7) neg-bounds-compare(2)
  by auto
have  $\mathcal{B}_u s x_i = LB \text{ Negative } s x_i$ 
  by simp
show check-dom
  (case min-rvar-incdec Negative s xi of
   Inl I  $\Rightarrow$  set-unsat I s
   | Inr xj  $\Rightarrow$  pivot-and-update xi xj (the (Bu s xi)) s)
  apply (subst <Bu s xi = LB Negative s xi>)
  apply (rule check-sat-terminates'-aux)
  using  $\langle \nabla s \rangle \langle \Delta (\mathcal{T} s) \rangle \langle \Diamond s \rangle \models_{nolhs} s *$ 
  using  $\neg \mathcal{U} s \langle \text{Some } x_i = \text{min-lvar-not-in-bounds } s \rangle \langle \langle \mathcal{V} s \rangle x_i >_{ub} \mathcal{B}_u s x_i \rangle$ 
   $\neg \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i \rangle$ 
  by (simp-all add: bound-compare''-defs)
qed
qed

lemma check-sat-terminates:
assumes  $\nabla s \Delta (\mathcal{T} s) \Diamond s \models_{nolhs} s$ 
shows check-dom s
using assms
using check-sat-terminates'[of s s]
by simp

lemma check-cases:
assumes  $\mathcal{U} s \Rightarrow P s$ 
assumes  $\llbracket \neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{None} \rrbracket \Rightarrow P s$ 
assumes  $\bigwedge x_i \text{ dir } I.$ 
 $\llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative};$ 
 $\neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{Some } x_i;$ 
 $\triangleleft_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x_i) (LB \text{ dir } s x_i);$ 
 $\text{min-rvar-incdec dir } s x_i = \text{Inl } I \rrbracket \Rightarrow$ 
 $P (\text{set-unsat } I s)$ 
assumes  $\bigwedge x_i x_j l_i \text{ dir}.$ 
 $\llbracket \text{dir} = (\text{if } \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i \text{ then Positive else Negative});$ 
 $\neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{Some } x_i;$ 
 $\triangleleft_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x_i) (LB \text{ dir } s x_i);$ 
 $\text{min-rvar-incdec dir } s x_i = \text{Inr } x_j;$ 
 $l_i = \text{the } (LB \text{ dir } s x_i);$ 
 $\text{check}' \text{ dir } x_i s = \text{pivot-and-update } x_i x_j l_i s \rrbracket \Rightarrow$ 
 $P (\text{check } (\text{pivot-and-update } x_i x_j l_i s))$ 
assumes  $\Delta (\mathcal{T} s) \Diamond s \models_{nolhs} s$ 
shows  $P (\text{check } s)$ 
proof (cases U s)
  case True

```

```

then show ?thesis
  using assms(1)
  using check.simps[of s]
  by simp
next
  case False
  show ?thesis
  proof (cases min-lvar-not-in-bounds s)
    case None
    then show ?thesis
    using ‐U s
    using assms(2) △ (T s) ◊ s ⊨nolhs s
    using check.simps[of s]
    by simp
next
  case (Some xi)
    let ?dir = if (V s) xi <lb Bl s xi) then (Positive :: ('i,'a)Direction) else
    Negative
    let ?s' = check' ?dir xi s
    have ▲lb (lt ?dir) (V s) xi (LB ?dir s xi)
    using min-lvar-not-in-bounds = Some xi
    using min-lvar-not-in-bounds-Some[of s xi]
    using not-in-bounds[of xi (V s) Bl s Bu s]
    by (auto split: if-splits simp add: bound-compare"-defs)

    have P (check ?s')
      apply (rule check'-cases)
      using ‐U s ▲ min-lvar-not-in-bounds s = Some xi ▲lb (lt ?dir) (V s) xi
      (LB ?dir s xi)
      using assms(3)[of ?dir xi]
      using assms(4)[of ?dir xi]
      using check.simps[of set-unsat (- :: 'i list) s]
      using △ (T s) ◊ s ⊨nolhs s
      by (auto simp add: bounds-consistent-def curr-val-satisfies-no-lhs-def)
then show ?thesis
  using ‐U s ▲ min-lvar-not-in-bounds s = Some xi
  using check.simps[of s]
  using △ (T s) ◊ s ⊨nolhs s
  by auto
qed
qed

```

```

lemma check-induct:
  fixes s :: ('i,'a) state
  assumes *: V s △ (T s) ⊨nolhs s ◊ s
  assumes **:
    △ s. U s ⟹ P s s
    △ s. [¬ U s; min-lvar-not-in-bounds s = None] ⟹ P s s

```

$\bigwedge s x_i \text{ dir } I. [\text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}; \neg \mathcal{U} s; \text{min-lvar-not-in-bounds}$
 $s = \text{Some } x_i;$
 $\quad \triangleleft_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x_i) (\text{LB dir } s x_i); \text{min-rvar-incdec dir } s x_i = \text{Inl } I]$
 $\implies P s (\text{set-unsat } I s)$
assumes $\text{step}' : \bigwedge s x_i x_j l_i. [\triangle (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-eq}$
 $(\text{eq-for-lvar } (\mathcal{T} s) x_i)] \implies P s (\text{pivot-and-update } x_i x_j l_i s)$
assumes $\text{trans}' : \bigwedge s_i s_j sk. [P s_i s_j; P s_j sk] \implies P s_i sk$
shows $P s (\text{check } s)$
proof-
have $\text{check-dom } s$
using*
by (*simp add: check-sat-terminates*)
then show ?thesis
using*
proof (*induct s rule: check-dom.induct*)
case ($\text{step } s'$)
show ?case
proof (*rule check-cases*)
fix $x_i x_j l_i \text{ dir}$
let $?dir = \text{if } \langle \mathcal{V} s' \rangle x_i <_{lb} \mathcal{B}_l s' x_i \text{ then Positive else Negative}$
let $?s' = \text{check}' \text{ dir } x_i s'$
assume $\neg \mathcal{U} s' \text{ min-lvar-not-in-bounds } s' = \text{Some } x_i \text{ min-rvar-incdec dir } s'$
 $x_i = \text{Inr } x_j l_i = \text{the } (\text{LB dir } s' x_i)$
 $?s' = \text{pivot-and-update } x_i x_j l_i s' \text{ dir} = ?dir$
moreover
then have $\nabla ?s' \triangle (\mathcal{T} ?s') \models_{nolhs} ?s' \diamond ?s'$
using $\langle \nabla s' \rangle \langle \triangle (\mathcal{T} s') \rangle \langle \models_{nolhs} s' \rangle \langle \diamond s' \rangle$
using $\langle ?s' = \text{pivot-and-update } x_i x_j l_i s' \rangle$
using $\text{pivotandupdate-check-precond}[\text{of dir } s' x_i x_j l_i]$
by *auto*
ultimately
have $P (\text{check}' \text{ dir } x_i s') (\text{check} (\text{check}' \text{ dir } x_i s'))$
using $\text{step}(2)[\text{of } x_i] \text{ step}(4)[\text{of } x_i] \langle \triangle (\mathcal{T} s') \rangle \langle \nabla s' \rangle$
by *auto*
then show $P s' (\text{check} (\text{pivot-and-update } x_i x_j l_i s'))$
using $\langle ?s' = \text{pivot-and-update } x_i x_j l_i s' \rangle \langle \triangle (\mathcal{T} s') \rangle \langle \nabla s' \rangle$
using $\langle \text{min-lvar-not-in-bounds } s' = \text{Some } x_i \rangle \langle \text{min-rvar-incdec dir } s' x_i =$
 $\text{Inr } x_j \rangle$
using $\text{step}'[\text{of } s' x_i x_j l_i]$
using $\text{trans}'[\text{of } s' ?s' \text{ check } ?s']$
by (*auto simp add: min-lvar-not-in-bounds-lvars min-rvar-incdec-eq-Some-rvars*)
qed (*simp-all add:* $\langle \nabla s' \rangle \langle \triangle (\mathcal{T} s') \rangle \langle \models_{nolhs} s' \rangle \langle \diamond s' \rangle$ **)
qed
qed
lemma $\text{check-induct}'$:
fixes $s :: ('i, 'a) \text{ state}$
assumes $\nabla s \triangle (\mathcal{T} s) \models_{nolhs} s \diamond s$
assumes $\bigwedge s x_i \text{ dir } I. [\text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}; \neg \mathcal{U} s; \text{min-lvar-not-in-bounds}$

```

s = Some xi;
  ⋲lb (lt dir) ((V s) xi) (LB dir s xi); min-rvar-incdec dir s xi = Inl I; P s]
    ==> P (set-unsat I s)
  assumes ⋀ s xi xj li. [△ (T s); ∇ s; xi ∈ lvars (T s); xj ∈ rvars-eq (eq-for-lvar
  (T s) xi); P s] ==> P (pivot-and-update xi xj li s)
  assumes P s
  shows P (check s)
proof-
  have P s —> P (check s)
    by (rule check-induct) (simp-all add: assms)
  then show ?thesis
    using ⟨P s⟩
    by simp
qed

lemma check-induct'':
  fixes s :: ('i,'a) state
  assumes *: ∇ s △ (T s) ⊨nolhs s ◇ s
  assumes **:
    U s ==> P s
    ⋀ s. [∇ s; △ (T s); ⊨nolhs s; ◇ s; ⋄ U s; min-lvar-not-in-bounds s = None]
  ==> P s
    ⋀ s xi dir I. [dir = Positive ∨ dir = Negative; ∇ s; △ (T s); ⊨nolhs s; ◇ s;
    ⋄ U s;
    min-lvar-not-in-bounds s = Some xi; ⋲lb (lt dir) ((V s) xi) (LB dir s xi);
    min-rvar-incdec dir s xi = Inl I]
    ==> P (set-unsat I s)
  shows P (check s)
proof (cases U s)
  case True
  then show ?thesis
    using ⟨U s ==> P s⟩
    by (simp add: check.simps)
next
  case False
  have check-dom s
    using *
    by (simp add: check-sat-terminates)
  then show ?thesis
    using * False
  proof (induct s rule: check-dom.induct)
    case (step s')
    show ?case
    proof (rule check-cases)
      fix xi xj li dir
      let ?dir = if (V s') xi <lb Bl s' xi then Positive else Negative
      let ?s' = check' dir xi s'
      assume ⋄ U s' min-lvar-not-in-bounds s' = Some xi min-rvar-incdec dir s'
      xi = Inr xj li = the (LB dir s' xi)
    qed
  qed

```

```

? $s' = \text{pivot-and-update } x_i \ x_j \ l_i \ s' \ \text{dir} = ?\text{dir}$ 
moreover
then have  $\nabla ?s' \triangle (\mathcal{T} ?s') \models_{nolhs} ?s' \diamondsuit ?s' \neg \mathcal{U} ?s'$ 
  using  $\langle \nabla s' \rangle \langle \triangle (\mathcal{T} s') \rangle \models_{nolhs} s' \langle \diamondsuit s' \rangle$ 
  using  $\langle ?s' = \text{pivot-and-update } x_i \ x_j \ l_i \ s' \rangle$ 
  using  $\text{pivotandupdate-check-precond}[\text{of dir } s' \ x_i \ x_j \ l_i]$ 
  using  $\text{pivotandupdate-unsat-id}[\text{of } s' \ x_i \ x_j \ l_i]$ 
by (auto simp add: min-lvar-not-in-bounds-lvars min-rvar-incdec-eq-Some-rvars)
ultimately
have  $P (\text{check} (\text{check}' \text{dir } x_i \ s'))$ 
  using step(2)[of  $x_i$ ] step(4)[of  $x_i$ ]  $\langle \triangle (\mathcal{T} s') \rangle \langle \nabla s' \rangle$ 
  by auto
then show  $P (\text{check} (\text{pivot-and-update } x_i \ x_j \ l_i \ s'))$ 
  using  $\langle ?s' = \text{pivot-and-update } x_i \ x_j \ l_i \ s' \rangle$ 
  by simp
qed (simp-all add:  $\langle \nabla s' \rangle \langle \triangle (\mathcal{T} s') \rangle \models_{nolhs} s' \langle \diamondsuit s' \rangle \neg \mathcal{U} s' \ast \ast$ )
qed
qed

```

end

```

lemma poly-eval-update:  $(p \setminus v (x := c :: 'a :: lrv) \setminus) = (p \setminus v \setminus) + \text{coeff } p \ x \ *R (c - v \ x)$ 
proof (transfer, simp, goal-cases)
  case (1  $p \ v \ x \ c$ )
    hence fin: finite { $v. p \ v \neq 0$ } by simp
    have  $(\sum y \in \{v. p \ v \neq 0\}. p \ y \ *R (\text{if } y = x \text{ then } c \text{ else } v \ y)) =$ 
       $(\sum y \in \{v. p \ v \neq 0\} \cap \{x\}. p \ y \ *R (\text{if } y = x \text{ then } c \text{ else } v \ y))$ 
       $+ (\sum y \in \{v. p \ v \neq 0\} \cap (\text{UNIV} - \{x\}). p \ y \ *R (\text{if } y = x \text{ then } c \text{ else } v \ y))$  (is  $?l = ?a + ?b$ )
        by (subst sum.union-disjoint[symmetric], auto intro: sum.cong fin)
    also have  $?a = (\text{if } p \ x = 0 \text{ then } 0 \text{ else } p \ x \ *R \ c)$  by auto
    also have ... =  $p \ x \ *R \ c$  by auto
    also have  $?b = (\sum y \in \{v. p \ v \neq 0\} \cap (\text{UNIV} - \{x\}). p \ y \ *R \ v \ y)$  (is  $- = ?c$ )
    by (rule sum.cong, auto)
    finally have  $?l = p \ x \ *R \ c + ?c$ .
    define  $r$  where  $r = (\sum y \in \{v. p \ v \neq 0\}. p \ y \ *R \ v \ y) + p \ x \ *R (c - v \ x)$ 
    have  $r = (\sum y \in \{v. p \ v \neq 0\}. p \ y \ *R \ v \ y) + p \ x \ *R (c - v \ x)$  by (simp add: r-def)
    also have  $(\sum y \in \{v. p \ v \neq 0\}. p \ y \ *R \ v \ y) =$ 
       $(\sum y \in \{v. p \ v \neq 0\} \cap \{x\}. p \ y \ *R \ v \ y) + ?c$  (is  $- = ?d + -$ )
      by (subst sum.union-disjoint[symmetric], auto intro: sum.cong fin)
    also have  $?d = (\text{if } p \ x = 0 \text{ then } 0 \text{ else } p \ x \ *R \ v \ x)$  by auto
    also have ... =  $p \ x \ *R \ v \ x$  by auto
    finally have  $(p \ x \ *R (c - v \ x) + p \ x \ *R \ v \ x) + ?c = r$  by simp
    also have  $(p \ x \ *R (c - v \ x) + p \ x \ *R \ v \ x) = p \ x \ *R \ c$  unfolding scaleRat-right-distrib[symmetric]
    by simp

```

```

finally have r: p x *R c + ?c = r .
show ?case unfolding l r r-def ..
qed

lemma bounds-consistent-set-unsat[simp]:  $\Diamond (\text{set-unsat } I s) = \Diamond s$ 
unfolding bounds-consistent-def boundsl-def boundsu-def set-unsat-simps by simp

lemma curr-val-satisfies-no-lhs-set-unsat[simp]:  $(\models_{nolhs} (\text{set-unsat } I s)) = (\models_{nolhs} s)$ 
unfolding curr-val-satisfies-no-lhs-def boundsl-def boundsu-def set-unsat-simps by auto

context PivotUpdateMinVars
begin
context
  fixes rhs-eq-val :: (var, 'a::lrv) mapping  $\Rightarrow$  var  $\Rightarrow$  'a  $\Rightarrow$  eq  $\Rightarrow$  'a
  assumes RhsEqVal rhs-eq-val
begin

lemma check-minimal-unsat-state-core:
  assumes *:  $\neg \mathcal{U} s \models_{nolhs} s \Diamond s \Delta (\mathcal{T} s) \nabla s$ 
  shows  $\mathcal{U} (\text{check } s) \longrightarrow \text{minimal-unsat-state-core } (\text{check } s)$ 
    (is ?P (check s))
  proof (rule check-induct'')
    fix s' :: ('i,'a) state and xi dir I
    assume nolhs:  $\models_{nolhs} s'$ 
      and min-rvar: min-rvar-incdec dir s' xi = Inl I
      and sat:  $\neg \mathcal{U} s'$ 
      and min-lvar: min-lvar-not-in-bounds s' = Some xi
      and dir: dir = Positive  $\vee$  dir = Negative
      and lt:  $\triangleleft_{lb} (lt \text{ dir}) ((\mathcal{V} s') x_i) (LB \text{ dir } s' x_i)$ 
      and norm:  $\Delta (\mathcal{T} s')$ 
      and valued:  $\nabla s'$ 
    let ?eq = eq-for-lvar (T s') xi
    have unsat-core: set (the (Uc (set-unsat I s'))) = set I
      by auto

    obtain li where LB-Some: LB dir s' xi = Some li and lt: lt dir ((V s') xi) li
      using lt by (cases LB dir s' xi) (auto simp add: bound-compare-defs)

    from LB-Some dir obtain i where LBI: look (LBI dir s') xi = Some (i,li) and
      LI: LI dir s' xi = i
      by (auto simp: simp: indexl-def indexu-def boundsl-def boundsu-def)

    from min-rvar-incdec-eq-None[OF min-rvar] dir
    have Is': LI dir s' (lhs (eq-for-lvar (T s') xi))  $\in$  indices-state s'  $\implies$  set I  $\subseteq$ 
      indices-state s' and
      reasable:  $\bigwedge x. x \in rvars-eq ?eq \implies \neg \text{reasable-var dir } x ?eq s'$  and

```

```

setI: set I =
  {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 ∧ 0 < coeff (rhs ?eq) x} (is - = ?L ∪ ?R1
  ∪ ?R2) by auto
note setI also have id: lhs ?eq = xi
  by (simp add: EqForLVar.eq-for-lvar EqForLVar-axioms min-lvar min-lvar-not-in-bounds-lvars)
finally have iI: i ∈ set I unfolding LI by auto
note setI = setI[unfolded id]
have LI dir s' xi ∈ indices-state s' using LBI LI
  unfolding indices-state-def using dir by force
from Is'[unfolded id, OF this]
have Is': set I ⊆ indices-state s' .

have xi ∈ lvars (T s')
  using min-lvar
  by (simp add: min-lvar-not-in-bounds-lvars)
then have **: ?eq ∈ set (T s') lhs ?eq = xi
  by (auto simp add: eq-for-lvar)

have Is': set I ⊆ indices-state (set-unsat I s')
  using Is' * unfolding indices-state-def by auto

have ⟨V s'⟩ ⊨t T s' and b: ⟨V s'⟩ ⊨b B s' || - lvars (T s')
  using nolhs[unfolded curr-val-satisfies-no-lhs-def] by auto
from norm[unfolded normalized-tableau-def]
have lvars-rvars: lvars (T s') ∩ rvars (T s') = {} by auto
hence in-bnds: x ∈ rvars (T s') ⇒ in-bounds x ⟨V s'⟩ (B s') for x
  by (intro b[unfolded satisfies-bounds-set.simps, rule-format, of x], auto)
{
  assume dist: distinct-indices-state (set-unsat I s')
  hence distinct-indices-state s' unfolding distinct-indices-state-def by auto
  note dist = this[unfolded distinct-indices-state-def, rule-format]
  {
    fix x c i y
    assume c: look (Bil s') x = Some (i, c) ∨ look (Biu s') x = Some (i, c)
      and y: y ∈ rvars-eq ?eq and
        coeff: coeff (rhs ?eq) y < 0 ∧ i = LI dir s' y ∨ coeff (rhs ?eq) y > 0 ∧ i
        = UI dir s' y
    {
      assume coeff: coeff (rhs ?eq) y < 0 and i: i = LI dir s' y
      from reusable[OF y] coeff have not-gt: ¬ (Dlb (lt dir) ((V s') y) (LB dir s'
      y)) by auto
      then obtain d where LB: LB dir s' y = Some d using dir by (cases LB
      dir s' y, auto simp: bound-compare-defs)
        with not-gt have le: le (lt dir) ((V s') y) d using dir by (auto simp:
        bound-compare-defs)
        from LB have look (LBI dir s') y = Some (i, d) unfolding i using dir
          by (auto simp: boundsl-def boundsu-def indexl-def indexu-def)
    }
  }
}

```

```

with c dist[of x i c y d] dir
have yx: y = x d = c by auto
from y[unfolded yx] have x ∈ rvars (T s') using **(1) unfolding rvars-def
by force
from in-bnds[OF this] le LB not-gt i have ⟨V s'⟩ x = c unfolding yx using
dir
    by (auto simp del: Simplex.bounds-lg)
    note yx(1) this
}
moreover
{
    assume coeff: coeff (rhs ?eq) y > 0 and i: i = UI dir s' y
    from reusable[OF y] coeff have not-gt: ¬ (⟨ub (lt dir) ((⟨V s'⟩ y) (UB dir
s' y))) by auto
    then obtain d where UB: UB dir s' y = Some d using dir by (cases UB
dir s' y, auto simp: bound-compare-defs)
    with not-gt have le: le (lt dir) d ((⟨V s'⟩ y)) using dir by (auto simp:
bound-compare-defs)
    from UB have look (UBI dir s') y = Some (i, d) unfolding i using dir
        by (auto simp: boundsl-def boundsu-def indexl-def indexu-def)
    with c dist[of x i c y d] dir
    have yx: y = x d = c by auto
    from y[unfolded yx] have x ∈ rvars (T s') using **(1) unfolding rvars-def
by force
from in-bnds[OF this] le UB not-gt i have ⟨V s'⟩ x = c unfolding yx using
dir
    by (auto simp del: Simplex.bounds-lg)
    note yx(1) this
}
ultimately have y = x ⟨V s'⟩ x = c using coeff by blast+
} note x-vars-main = this
{
fix x c i
assume c: look (Bil s') x = Some (i,c) ∨ look (Biu s') x = Some (i,c) and
i: i ∈ ?R1 ∪ ?R2
from i obtain y where y: y ∈ rvars-eq ?eq and
coeff: coeff (rhs ?eq) y < 0 ∧ i = LI dir s' y ∨ coeff (rhs ?eq) y > 0 ∧ i
= UI dir s' y
    by auto
from x-vars-main[OF c y coeff]
have y = x ⟨V s'⟩ x = c using coeff by blast+
    with y have x ∈ rvars-eq ?eq x ∈ rvars (T s') ⟨V s'⟩ x = c using **(1)
unfolding rvars-def by force+
} note x-rvars = this

have R1R2: (?R1 ∪ ?R2, ⟨V s'⟩) ⊨ise s'
    unfolding satisfies-state-index'.simp
proof (intro conjI)
    show ⟨V s'⟩ ⊨t T s' by fact

```

```

show (?R1 ∪ ?R2, ⟨V s'⟩) ⊨ibe BI s'
  unfolding satisfies-bounds-index'.simp
proof (intro conjI impI allI)
  fix x c
  assume c: Bl s' x = Some c and i: Il s' x ∈ ?R1 ∪ ?R2
  from c have ci: look (Bil s') x = Some (Il s' x, c) unfolding boundsl-def
indexl-def by auto
  from x-rvars[OF - i] ci show ⟨V s'⟩ x = c by auto
next
  fix x c
  assume c: Bu s' x = Some c and i: Iu s' x ∈ ?R1 ∪ ?R2
  from c have ci: look (Biu s') x = Some (Iu s' x, c) unfolding boundsu-def
indexu-def by auto
  from x-rvars[OF - i] ci show ⟨V s'⟩ x = c by auto
qed
qed

have id1: set (the (Uc (set-unsat I s'))) = set I
  ∧ x. x ⊨ise set-unsat I s' ↔ x ⊨ise s'
  by (auto simp: satisfies-state-index'.simp boundsl-def boundsu-def indexl-def
indexu-def)
have subsets-sat-core (set-unsat I s') unfolding subsets-sat-core-def id1
proof (intro allI impI)
  fix J
  assume sub: J ⊂ set I
  show ∃ v. (J, v) ⊨ise s'
  proof (cases J ⊆ ?R1 ∪ ?R2)
    case True
    with R1R2 have (J, ⟨V s'⟩) ⊨ise s'
    unfolding satisfies-state-index'.simp satisfies-bounds-index'.simp by blast
    thus ?thesis by blast
  next
    case False
    with sub obtain k where k: k ∈ ?R1 ∪ ?R2 k ∉ J k ∈ set I unfolding
setI by auto
    from k(1) obtain y where y: y ∈ rvars-eq ?eq
      and coeff: coeff (rhs ?eq) y < 0 ∧ k = LI dir s' y ∨ coeff (rhs ?eq) y >
0 ∧ k = UI dir s' y by auto
      hence cy0: coeff (rhs ?eq) y ≠ 0 by auto
    from y **(1) have ry: y ∈ rvars (T s') unfolding rvars-def by force
      hence yl: y ∉ lvars (T s') using lvars-rvars by blast
      interpret rev: RhsEqVal rhs-eq-val by fact
      note update = rev.update-valuation-nonlhs[THEN mp, OF norm valued yl]
      define diff where diff = li - ⟨V s'⟩ xi
      have ⟨V s'⟩ xi < li ⇒ 0 < li - ⟨V s'⟩ xi li < ⟨V s'⟩ xi ⇒ li - ⟨V s'⟩ xi
< 0
      using minus-gt by (blast, insert minus-lt, blast)
      with lt dir have diff: lt dir 0 diff by (auto simp: diff-def simp del:

```

```

Simplex.bounds-lg)
  define up where up = inverse (coeff (rhs ?eq) y) *R diff
  define v where v = ⟨V (rev.update y ((V s') y + up) s')⟩
  show ?thesis unfolding satisfies-state-index'.simp
  proof (intro exI[of - v] conjI)
    show v ⊨t T s' unfolding v-def
    using rev.update-satisfies-tableau[OF norm valuated yl] ⟨V s'⟩ ⊨t T s'
  by auto
  with **(1) have v ⊨e ?eq unfolding satisfies-tableau-def by auto
  from this[unfolded satisfies-eq-def id]
  have v-xi: v xi = (rhs ?eq { v }) .
  from ⟨V s'⟩ ⊨t T s' **(1) have ⟨V s'⟩ ⊨e ?eq unfolding satisfies-tableau-def by auto
  hence V-xi: ⟨V s'⟩ xi = (rhs ?eq { ⟨V s'⟩ }) unfolding satisfies-eq-def id .
  have v xi = ⟨V s'⟩ xi + coeff (rhs ?eq) y *R up
  unfolding v-xi unfolding v-def rev.update-valuate-rhs[OF **(1) norm]
  poly-eval-update V-xi by simp
  also have ... = li unfolding up-def diff-def scaleRat-scaleRat using cy0
  by simp
  finally have v-xi-l: v xi = li .

  {
    assume both: Iu s' y ∈ ?R1 ∪ ?R2 Bu s' y ≠ None Il s' y ∈ ?R1 ∪
    ?R2 Bl s' y ≠ None
    and diff: Il s' y ≠ Iu s' y
    from both(1) dir obtain xu cu where
      looku: look (Bil s') xu = Some (Iu s' y, cu) ∨ look (Biu s') xu = Some
      (Iu s' y, cu)
      by (smt (verit) Is' Un-insert-left indices-state-def indices-state-set-unsat
      insert-iff mem-Collect-eq setI subsetD
      sup-bot-left)
    from both(1) obtain xu' where xu' ∈ rvars-eq ?eq coeff (rhs ?eq) xu'
    < 0 ∧ Iu s' y = LI dir s' xu' ∨
      coeff (rhs ?eq) xu' > 0 ∧ Iu s' y = UI dir s' xu' by blast
    with x-vars-main(1)[OF looku this]
    have xu: xu ∈ rvars-eq ?eq coeff (rhs ?eq) xu < 0 ∧ Iu s' y = LI dir s'
    xu ∨
      coeff (rhs ?eq) xu > 0 ∧ Iu s' y = UI dir s' xu by auto
  {
    assume xu ≠ y
    with dist[OF looku, of y] have look (Biu s') y = None
    by (cases look (Biu s') y, auto simp: boundsu-def indexu-def, blast)
    with both(2) have False by (simp add: boundsu-def)
  }
  hence xu-y: xu = y by blast
  from both(3) dir obtain xl cl where
    lookl: look (Bil s') xl = Some (Il s' y, cl) ∨ look (Biu s') xl = Some
    (Il s' y, cl)
    by (smt (verit) Is' Un-insert-right in-mono indices-state-def in-

```

```

dices-state-set-unsat insert-compr mem-Collect-eq setI
    sup-bot.right-neutral sup-commute)
from both(3) obtain xl' where xl' ∈ rvars-eq ?eq coeff (rhs ?eq) xl' <
0 ∧ Il s' y = LI dir s' xl' ∨
    coeff (rhs ?eq) xl' > 0 ∧ Il s' y = UI dir s' xl' by blast
with x-vars-main(1)[OF lookl this]
have xl: xl ∈ rvars-eq ?eq coeff (rhs ?eq) xl < 0 ∧ Il s' y = LI dir s' xl
∨
    coeff (rhs ?eq) xl > 0 ∧ Il s' y = UI dir s' xl by auto
{
assume xl ≠ y
with dist[OF lookl, of y] have look (Bil s') y = None
    by (cases look (Bil s') y, auto simp: boundsl-def indexl-def, blast)
with both(4) have False by (simp add: boundsl-def)
}
hence xl-y: xl = y by blast
from xu(2) xl(2) diff have diff: xu ≠ xl by auto
with xu-y xl-y have False by simp
} note both-y-False = this
show (J, v) ⊨ibe BI s' unfolding satisfies-bounds-index'.simps
proof (intro conjI allI impI)
fix x c
assume x: Bl s' x = Some c Il s' x ∈ J
with k have not-k: Il s' x ≠ k by auto
from x have ci: look (Bil s') x = Some (Il s' x, c) unfolding boundsl-def
indexl-def by auto
show v x = c
proof (cases Il s' x = i)
case False
hence iR12: Il s' x ∈ ?R1 ∪ ?R2 using sub x unfolding setI LI by
blast
from x-rvars(2–3)[OF - iR12] ci have xr: x ∈ rvars (T s') and val:
⟨v s'⟩ x = c by auto
with lvars-rvars have xl: x ∉ lvars (T s') by auto
show ?thesis
proof (cases x = y)
case False
thus ?thesis using val unfolding v-def map2fun-def' update[OF xl]
using val by auto
next
case True
note coeff = coeff[folded True]
from coeff not-k dir ci have Iu: Iu s' x = k by auto
with ci Iu x(2) k sub False True
have both: Iu s' y ∈ ?R1 ∪ ?R2 Il s' y ∈ ?R1 ∪ ?R2 and diff: Il
s' y ≠ Iu s' y
unfolding setI LI by auto
have Bl s' y ≠ None using x True by simp
from both-y-False[OF both(1) - both(2) this diff]

```

```

have  $\mathcal{B}_u s' y = \text{None}$  by metis
with  $\text{reasable}[OF y]$  dir coeff True
have dir = Negative  $\implies 0 < \text{coeff}(\text{rhs } ?eq) y$  dir = Positive  $\implies 0 > \text{coeff}(\text{rhs } ?eq) y$  by (auto simp: bound-compare-defs)
with dir coeff[unfolded True] have k =  $\mathcal{I}_l s' y$  by auto
with diff Iu False True
have False by auto
thus ?thesis ..
qed
next
case True
from LBI ci[unfolded True] dir
dist[unfolded distinct-indices-state-def, rule-format, of x i c x_i l_i]
have xxi:  $x = x_i$  and c:  $c = l_i$  by auto
have vxi:  $v x = l_i$  unfolding xxi v-xi-l ..
thus ?thesis unfolding c by simp
qed
next
fix x c
assume x:  $\mathcal{B}_u s' x = \text{Some } c$   $\mathcal{I}_u s' x \in J$ 
with k have not-k:  $\mathcal{I}_u s' x \neq k$  by auto
from x have ci: look ( $\mathcal{B}_{iu} s'$ )  $x = \text{Some } (\mathcal{I}_u s' x, c)$  unfolding
boundsu-def indexu-def by auto
show  $v x = c$ 
proof (cases  $\mathcal{I}_u s' x = i$ )
case False
hence iR12:  $\mathcal{I}_u s' x \in ?R1 \cup ?R2$  using sub x unfolding setI LI by
blast
from x-rvars(2-3)[OF - iR12] ci have xr:  $x \in rvars(\mathcal{T} s')$  and val:
⟨V s'⟩ x = c by auto
with lvars-rvars have xl:  $x \notin lvars(\mathcal{T} s')$  by auto
show ?thesis
proof (cases x = y)
case False
thus ?thesis using val unfolding v-def map2fun-def' update[OF xl]
using val by auto
next
case True
note coeff = coeff[folded True]
from coeff not-k dir ci have Iu:  $\mathcal{I}_l s' x = k$  by auto
with ci Iu x(2) k sub False True
have both:  $\mathcal{I}_u s' y \in ?R1 \cup ?R2$   $\mathcal{I}_l s' y \in ?R1 \cup ?R2$  and diff:  $\mathcal{I}_l s' y \neq \mathcal{I}_u s' y$ 
unfolding setI LI by auto
have  $\mathcal{B}_u s' y \neq \text{None}$  using x True by simp
from both-y-False[OF both(1) this both(2) - diff]
have  $\mathcal{B}_l s' y = \text{None}$  by metis
with  $\text{reasable}[OF y]$  dir coeff True
have dir = Negative  $\implies 0 > \text{coeff}(\text{rhs } ?eq) y$  dir = Positive  $\implies 0$ 

```

```

< coeff (rhs ?eq) y by (auto simp: bound-compare-defs)
  with dir coeff[unfolded True] have k = Iu s' y by auto
  with diff Iu False True
  have False by auto
  thus ?thesis ..
qed
next
case True
from LBI ci[unfolded True] dir
dist[unfolded distinct-indices-state-def, rule-format, of x i c xi li]
have xxi: x = xi and c: c = li by auto
have vxi: v x = li unfolding xxi v-xi-l ..
thus ?thesis unfolding c by simp
qed
qed
qed
qed
qed
qed
qed
} note minimal-core = this

have unsat-core: unsat-state-core (set-unsat I s')
  unfolding unsat-state-core-def unsat-core
proof (intro impI conjI Is', clarify)
fix v
assume (set I, v) ⊨is set-unsat I s'
then have Iv: (set I, v) ⊨is s'
  unfolding satisfies-state-index.simps
  by (auto simp: indexl-def indexu-def boundsl-def boundsu-def)
from Iv have vt: v ⊨t T s' and Iv: (set I, v) ⊨ib BI s'
  unfolding satisfies-state-index.simps by auto

have lt-le-eq: ∏ x y :: 'a. (x < y) ↔ (x ≤ y ∧ x ≠ y) by auto
from Iv dir
have lb: ∏ x i c l. look (LBI dir s') x = Some (i,l) ⇒ i ∈ set I ⇒ le (lt dir)
l (v x)
  unfolding satisfies-bounds-index.simps
  by (auto simp: lt-le-eq indexl-def indexu-def boundsl-def boundsu-def)

from lb[OF LBI iI] have li-x: le (lt dir) li (v xi) .

have ⟨V s'⟩ ⊨e ?eq
  using nolhs ⟨?eq ∈ set (T s')⟩
  unfolding curr-val-satisfies-no-lhs-def
  by (simp add: satisfies-tableau-def)
then have ⟨V s'⟩ xi = (rhs ?eq) {⟨V s'⟩}
  using ⟨lhs ?eq = xi⟩
  by (simp add: satisfies-eq-def)

moreover

```

```

have  $v \models_e ?eq$ 
  using  $vt \langle ?eq \in set (\mathcal{T} s') \rangle$ 
  by (simp add: satisfies-state-def satisfies-tableau-def)
then have  $v x_i = (rhs ?eq) \{ v \}$ 
  using  $\langle lhs ?eq = x_i \rangle$ 
  by (simp add: satisfies-eq-def)

moreover

have  $\triangleright_{lb} (lt dir) (v x_i) (LB dir s' x_i)$ 
  using li-x dir unfolding LB-Some by (auto simp: bound-compare'-defs)

moreover

from min-rvar-incdec-eq-None'[rule-format, OF dir min-rvar refl Iv]
have  $le (lt dir) (rhs (?eq) \{ v \}) (rhs (?eq) \{ \langle \mathcal{V} s' \rangle \})$  .

ultimately

show False
  using dir lt LB-Some
  by (auto simp add: bound-compare-defs)
qed

thus  $\mathcal{U} (set-unsat I s') \longrightarrow minimal-unsat-state-core (set-unsat I s')$  using min-
imal-core
  by (auto simp: minimal-unsat-state-core-def)
qed (simp-all add: *)

lemma Check-check: Check check
proof
  fix  $s :: ('i, 'a) state$ 
  assume  $\mathcal{U} s$ 
  then show check  $s = s$ 
    by (simp add: check.simps)
next
  fix  $s :: ('i, 'a) state$  and  $v :: 'a valuation$ 
  assume  $*: \nabla s \Delta (\mathcal{T} s) \models_{nolhs} s \diamondsuit s$ 
  then have  $v \models_t \mathcal{T} s = v \models_t \mathcal{T} (check s)$ 
    by (rule check-induct, simp-all add: pivotandupdate-tableau-equiv)
moreover
  have  $\Delta (\mathcal{T} (check s))$ 
    by (rule check-induct', simp-all add: * pivotandupdate-tableau-normalized)
moreover
  have  $\diamondsuit (check s)$ 
    by (rule check-induct', simp-all add: * pivotandupdate-tableau-normalized piv-
otandupdate-bounds-consistent)
moreover

```

```

have  $\models_{nolhs} (\text{check } s)$ 
  by (rule check-induct'', simp-all add: *)
moreover
have  $\nabla (\text{check } s)$ 
proof (rule check-induct', simp-all add: * pivotandupdate-tableau-valuated)
fix  $s I$ 
show  $\nabla s \implies \nabla (\text{set-unsat } I s)$ 
  by (simp add: tableau-valuated-def)
qed
ultimately
show let  $s' = \text{check } s \text{ in } v \models_t \mathcal{T} s = v \models_t \mathcal{T} s' \wedge \Delta (\mathcal{T} s') \wedge \nabla s' \wedge \models_{nolhs} s'$ 
 $\wedge \Diamond s'$ 
  by (simp add: Let-def)
next
fix  $s :: ('i,'a) state$ 
assume  $*: \nabla s \Delta (\mathcal{T} s) \models_{nolhs} s \Diamond s$ 
from * show  $\mathcal{B}_i (\text{check } s) = \mathcal{B}_i s$ 
  by (rule check-induct, simp-all add: pivotandupdate-bounds-id)
next
fix  $s :: ('i,'a) state$ 
assume  $*: \neg \mathcal{U} s \models_{nolhs} s \Diamond s \Delta (\mathcal{T} s) \nabla s$ 
have  $\neg \mathcal{U} (\text{check } s) \longrightarrow \models (\text{check } s)$ 
proof (rule check-induct'', simp-all add: *)
fix  $s$ 
assume min-lvar-not-in-bounds  $s = \text{None} \neg \mathcal{U} s \models_{nolhs} s$ 
then show  $\models s$ 
  using min-lvar-not-in-bounds-None[of s]
  unfolding curr-val-satisfies-state-def satisfies-state-def
  unfolding curr-val-satisfies-no-lhs-def
  by (auto simp add: satisfies-bounds-set.simps satisfies-bounds.simps)
qed
then show  $\neg \mathcal{U} (\text{check } s) \implies \models (\text{check } s)$  by blast
next
fix  $s :: ('i,'a) state$ 
assume  $*: \neg \mathcal{U} s \models_{nolhs} s \Diamond s \Delta (\mathcal{T} s) \nabla s$ 
have  $\mathcal{U} (\text{check } s) \longrightarrow \text{minimal-unsat-state-core} (\text{check } s)$ 
  by (rule check-minimal-unsat-state-core[OF *])
then show  $\mathcal{U} (\text{check } s) \implies \text{minimal-unsat-state-core} (\text{check } s)$  by blast
qed
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, compar-

isons 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 :: -  $\Rightarrow$  ('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 :: -  $\Rightarrow$  ('i,'a :: lrv)state
proof
  fix t
  let ?init = init-state t :: ('i,'a)state
  show ⟨V ?init⟩  $\models_t$  t
    unfolding satisfies-tableau-def satisfies-eq-def
    proof (safe)
      fix l r
      assume (l, r)  $\in$  set t
      then have l  $\in$  set (vars-list t) vars r  $\subseteq$  set (vars-list t)
        by (auto simp: set-vars-list) (transfer, force)
      then have *: vars r  $\subseteq$  lhs `set t  $\cup$  (Union x $\in$ set t. rvars-eq x) by (auto simp: set-vars-list)
      have ⟨V ?init⟩ l = (0::'a)
        using ⟨l  $\in$  set (vars-list t)⟩
        unfolding init-state-def by (auto simp: map2fun-def lookup-tabulate)
      moreover
      have r ∋ ⟨V ?init⟩ = (0::'a) using *
      proof (transfer fixing: t, goal-cases)
        case (1 r)
        {
          fix x
          assume x $\in$ {v. r v  $\neq$  0}
```

```

then have r x *R ⟨V ?init⟩ x = (0::'a)
  using 1
  unfolding init-state-def
  by (auto simp add: map2fun-def lookup-tabulate comp-def restrict-map-def
set-vars-list Abstract-Linear-Poly.vars-def)
}
then show ?case by auto
qed
ultimately
show ⟨V ?init⟩ (lhs (l, r)) = rhs (l, r) {⟨V ?init⟩}
  by auto
qed
next
fix t
show ∇ (init-state t)
  unfolding init-state-def
  by (auto simp add: lookup-tabulate tableau-valuated-def comp-def restrict-map-def
set-vars-list lvars-def rvars-def)
qed (simp-all add: init-state-def add: boundsl-def boundsu-def indexl-def indexu-def)

```

```

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
fix s::('i,'a) state
show min-lvar-not-in-bounds s = None →
  ( ∀ x ∈ lvars (T s). in-bounds x (⟨V s⟩) (B s))
  unfolding min-lvar-not-in-bounds-def lvars-def
  using min-satisfying-None
  by blast
next
fix s xi
show min-lvar-not-in-bounds s = Some xi →
  xi ∈ lvars (T s) ∧
  ¬ in-bounds xi (⟨V s⟩) (B s) ∧
  ( ∀ x ∈ lvars (T s). x < xi → in-bounds x (⟨V s⟩) (B s))
  unfolding min-lvar-not-in-bounds-def lvars-def
  using min-satisfying-Some
  by blast+
qed

```

— all variables in vs have either a positive or a negative coefficient, so no equal-zero test required.

```
definition unsat-indices :: ('i,'a :: linorder) Direction  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  var list
 $\Rightarrow$  eq  $\Rightarrow$  '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 ( $\lambda$  x. if coeff r x < 0 then li x else ui x) vs))
```

```
definition min-rvar-incdec-eq :: ('i,'a) Direction  $\Rightarrow$  ('i,'a::lrv) state  $\Rightarrow$  eq  $\Rightarrow$  'i list
+ var where
  min-rvar-incdec-eq dir s eq = (let rvars = Abstract-Linear-Poly.vars-list (rhs eq)
    in case min-satisfying ( $\lambda$  x. reusable-var dir x eq s) rvars of
      None  $\Rightarrow$  Inl (unsat-indices dir s rvars eq)
      | Some xj  $\Rightarrow$  Inr xj)
```

```
interpretation MinRVarsEq min-rvar-incdec-eq :: ('i,'a :: lrv) Direction  $\Rightarrow$  -
proof
  fix s eq is and dir :: ('i,'a) Direction
  let ?min = min-satisfying ( $\lambda$  x. reusable-var dir x eq s) (Abstract-Linear-Poly.vars-list
  (rhs eq))
  let ?vars = Abstract-Linear-Poly.vars-list (rhs eq)
  {
    assume min-rvar-incdec-eq dir s eq = Inl is
    from this[unfolded min-rvar-incdec-eq-def Let-def, simplified]
    have ?min = None and I: set is = set (unsat-indices dir s ?vars eq) by (cases
    ?min, auto)+
    from this min-satisfying-None set-vars-list
    have 1:  $\bigwedge$  x. x  $\in$  rvars-eq eq  $\Rightarrow$   $\neg$  reusable-var dir x eq s by blast
    {
      fix i
      assume i  $\in$  set is and dir: dir = Positive  $\vee$  dir = Negative and lhs-eq: LI
      dir s (lhs eq)  $\in$  indices-state s
      from this[unfolded I unsat-indices-def Let-def]
      consider (lhs) i = LI dir s (lhs eq)
        | (LI-rhs) x where i = LI dir s x x  $\in$  rvars-eq eq coeff (rhs eq) x < 0
        | (UI-rhs) x where i = UI dir s x x  $\in$  rvars-eq eq coeff (rhs eq) x  $\geq$  0
        by (auto split: if-splits simp: set-vars-list)
      then have i  $\in$  indices-state s
      proof cases
        case lhs
        show ?thesis unfolding lhs using lhs-eq by auto
      next
        case LI-rhs
        from 1[OF LI-rhs(2)] LI-rhs(3)
        have  $\neg$  ( $\triangleright_{lb}$  (lt dir) ( $\langle V$  s) x) (LB dir s x)) by auto
        then show ?thesis unfolding LI-rhs(1) unfolding indices-state-def using
        dir
        by (auto simp: bound-compare'-defs boundsL-def boundsU-def indexL-def
        indexU-def
          split: option.splits intro!: exI[of - x]) auto
```

```

next
  case UI-rhs
    from UI-rhs(2) have coeff (rhs eq) x ≠ 0
      by (simp add: coeff-zero)
    with UI-rhs(3) have 0 < coeff (rhs eq) x by auto
    from 1[OF UI-rhs(2)] this have ¬ (△ub (lt dir) (⟨V s⟩ x) (UB dir s x)) by
  auto
  then show ?thesis unfolding UI-rhs(1) unfolding indices-state-def using
  dir
    by (auto simp: bound-compare'-defs boundsl-def boundsu-def indexl-def
  indexu-def
      split: option.splits intro!: exI[of - x]) auto
  qed
}
then have 2: dir = Positive ∨ dir = Negative ⟹ LI dir s (lhs eq) ∈ indices-state s ⟹
set is ⊆ indices-state s by auto
show
  (∀ x ∈ rvars-eq eq. ¬ reusable-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 ∧ 0 < coeff (rhs
eq) x} ∧
  (dir = Positive ∨ dir = Negative → LI dir s (lhs eq) ∈ indices-state s →
set is ⊆ indices-state s)
proof (intro conjI impI 2, goal-cases)
  case 2
  have set is = {LI dir s (lhs eq)} ∪ LI dir s ` (rvars-eq eq ∩ {x. coeff (rhs eq)
x < 0}) ∪ UI dir s ` (rvars-eq eq ∩ {x. ¬ coeff (rhs eq) x < 0})
  unfolding I unsat-indices-def Let-def
  by (auto simp add: set-vars-list)
  also have ... = {LI dir s (lhs eq)} ∪ LI dir s ` {x. x ∈ rvars-eq eq ∧ coeff
(rhs eq) x < 0}
    ∪ UI dir s ` {x. x ∈ rvars-eq eq ∧ 0 < coeff (rhs eq) x}
  proof (intro arg-cong2[of ---- (∪)] arg-cong[of - - λ x. - ` x] refl, goal-cases)
    case 2
    {
      fix x
      assume x ∈ rvars-eq eq
      hence coeff (rhs eq) x ≠ 0
        by (simp add: coeff-zero)
      hence or: coeff (rhs eq) x < 0 ∨ coeff (rhs eq) x > 0 by auto
      assume ¬ coeff (rhs eq) x < 0
      hence coeff (rhs eq) x > 0 using or by simp
    } note [dest] = this
    show ?case by auto
  qed auto
  finally
  show set is = {LI dir s (lhs eq)} ∪ {LI dir s x | x. x ∈ rvars-eq eq ∧ coeff
(rhs eq) x < 0}

```

```

 $\cup \{UI\ dir\ s\ x\mid x.\ x\in rvars-eq\ eq\wedge 0<coeff\ (rhs\ eq)\ x\}\ by\ auto$ 
qed (insert 1, auto)
}
fix  $x_j$ 
assume min-rvar-incdec-eq dir s eq = Inr  $x_j$ 
from this[unfolded min-rvar-incdec-eq-def Let-def]
have ?min = Some  $x_j$  by (cases ?min, auto)
then show  $x_j \in rvars-eq eq$  reasable-var dir  $x_j$  eq s
( $\forall x' \in rvars-eq eq. x' < x_j \longrightarrow \neg$  reasable-var dir  $x'$  eq s)
using min-satisfying-Some set-vars-list by blast+
qed

```

```

primrec eq-idx-for-lvar-aux :: tableau  $\Rightarrow$  var  $\Rightarrow$  nat  $\Rightarrow$  nat where
eq-idx-for-lvar-aux []  $x = i$ 
| 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-idx-for-lvar-aux t x i in
i ≤ idx  $\wedge$  idx < i + length t  $\wedge$  lhs (t ! (idx - i)) = x
using assms
proof (induct t arbitrary: i)
case Nil
then show ?case
by (simp add: lvars-def)
next
case (Cons eq t)
show ?case
using Cons(1)[of i+1] Cons(2)
by (cases x = lhs eq) (auto simp add: Let-def lvars-def nth-Cons')
qed

global-interpretation EqForLVarDefault: EqForLVar eq-idx-for-lvar
defines eq-for-lvar-code = EqForLVarDefault.eq-for-lvar
proof (unfold-locales)
fix x t
assume  $x \in lvars t$ 
then show eq-idx-for-lvar t x < length t  $\wedge$ 
lhs (t ! eq-idx-for-lvar t x) = x
using eq-idx-for-lvar-aux[of x t 0]
by (simp add: Let-def eq-idx-for-lvar-def)

```

qed

```
definition pivot-eq :: eq ⇒ var ⇒ eq where
  pivot-eq e y ≡ 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 ∈ rvars-eq e
  shows v ⊨e e = v ⊨e pivot-eq e y
  using assms
  using scaleRat-right-distrib[of 1 / Rep-linear-poly (rhs e) y - (rhs e { v }) v
  (lhs e)]
  using Groups.group-add-class.minus-unique[of - ((rhs e) { v }) v (lhs e)]
  unfolding coeff-def vars-def
  by (simp add: coeff-def vars-def Let-def pivot-eq-def satisfies-eq-def)
  (auto simp add: rational-vector.scale-right-diff-distrib valueate-add valueate-minus
  valueate-uminus valueate-scaleRat valueate-Var)

lemma pivot-eq-rvars:
  assumes x ∈ vars (rhs (pivot-eq e v)) x ≠ lhs e coeff (rhs e) v ≠ 0 v ≠ lhs e
  shows x ∈ vars (rhs e)
proof-
  have v ∉ vars ((1 / coeff (rhs e) v) *R (rhs e - coeff (rhs e) v *R Var v))
  using coeff-zero
  by force
  then have x ≠ v
  using assms(1) assms(3) assms(4)
  using vars-plus[of (-1 / coeff (rhs e) v) *R (rhs e - coeff (rhs e) v *R Var
  v) (1 / coeff (rhs e) v) *R Var (lhs e)]
  by (auto simp add: Let-def vars-scaleRat pivot-eq-def)
  then show ?thesis
  using assms
  using vars-plus[of (-1 / coeff (rhs e) v) *R (rhs e - coeff (rhs e) v *R Var
  v) (1 / coeff (rhs e) v) *R Var (lhs e)]
  using vars-minus[of rhs e coeff (rhs e) v *R Var v]
  by (auto simp add: vars-scaleRat Let-def pivot-eq-def)
qed

interpretation PivotEq pivot-eq
proof
  fix eq xj
  assume xj ∈ rvars-eq eq lhs eq ∉ rvars-eq eq
  have lhs (pivot-eq eq xj) = xj
  unfolding pivot-eq-def
  by (simp add: Let-def)
```

```

moreover
have rvars-eq (pivot-eq eq xj) =
  {lhs eq} ∪ (rvars-eq eq - {xj})
proof
  show rvars-eq (pivot-eq eq xj) ⊆ {lhs eq} ∪ (rvars-eq eq - {xj})
  proof
    fix x
    assume x ∈ rvars-eq (pivot-eq eq xj)
    have *: coeff (rhs (pivot-eq eq xj)) xj = 0
      using ⟨xj ∈ rvars-eq eq⟩ ⟨lhs eq ∉ rvars-eq eq⟩
      using coeff-Var2[of lhs eq xj]
      by (auto simp add: Let-def pivot-eq-def)
    have coeff (rhs eq) xj ≠ 0
      using ⟨xj ∈ rvars-eq eq⟩
      using coeff-zero
      by (cases eq) (auto simp add:)
    then show x ∈ {lhs eq} ∪ (rvars-eq eq - {xj})
      using pivot-eq-rvars[of x eq xj]
      using ⟨x ∈ rvars-eq (pivot-eq eq xj)⟩ ⟨xj ∈ rvars-eq eq⟩ ⟨lhs eq ∉ rvars-eq
eq⟩
      using coeff-zero *
      by auto
  qed
  show {lhs eq} ∪ (rvars-eq eq - {xj}) ⊆ rvars-eq (pivot-eq eq xj)
  proof
    fix x
    assume x ∈ {lhs eq} ∪ (rvars-eq eq - {xj})
    have *: coeff (rhs eq) (lhs eq) = 0
      using coeff-zero
      using ⟨lhs eq ∉ rvars-eq eq⟩
      by auto
    have **: coeff (rhs eq) xj ≠ 0
      using ⟨xj ∈ rvars-eq eq⟩
      by (simp add: coeff-zero)
    have ***: x ∈ rvars-eq eq ==> coeff (Var (lhs eq)) x = 0
      using ⟨lhs eq ∉ rvars-eq eq⟩
      using coeff-Var2[of lhs eq x]
      by auto
    have coeff (Var xj) (lhs eq) = 0
      using ⟨xj ∈ rvars-eq eq⟩ ⟨lhs eq ∉ rvars-eq eq⟩
      using coeff-Var2[of xj lhs eq]
      by auto
    then have coeff (rhs (pivot-eq eq xj)) x ≠ 0
      using ⟨x ∈ {lhs eq} ∪ (rvars-eq eq - {xj})⟩ * *** ***
      using coeff-zero[of rhs eq x]
      by (auto simp add: Let-def coeff-Var2 pivot-eq-def)
    then show x ∈ rvars-eq (pivot-eq eq xj)
      by (simp add: coeff-zero)
  qed

```

```

qed
ultimately
show let eq' = pivot-eq eq xj in lhs eq' = xj ∧ rvars-eq eq' = {lhs eq} ∪ (rvars-eq
eq − {xj})
  by (simp add: Let-def)
next
fix v eq xj
assume xj ∈ rvars-eq eq
then show v ⊨e pivot-eq eq xj = v ⊨e eq
  using pivot-eq-satisfies-eq
  by blast
qed

```

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

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 (unfold-locales)
fix xj lp' lp
have *: ∀x. [x ∈ vars (lp + coeff lp xj *R lp' − coeff lp xj *R Var xj); x ∉ vars
lp] ⟹ x ∈ vars lp
proof-
fix x
assume x ∈ vars (lp + coeff lp xj *R lp' − coeff lp xj *R Var xj)
then have coeff (lp + coeff lp xj *R lp' − coeff lp xj *R Var xj) x ≠ 0
  using coeff-zero
  by force
assume x ∉ vars lp'
then have coeff lp' x = 0
  using coeff-zero
  by auto
show x ∈ vars lp
proof(rule ccontr)
assume x ∉ vars lp
then have coeff lp x = 0
  using coeff-zero
  by auto
then show False
  using ⟨coeff (lp + coeff lp xj *R lp' − coeff lp xj *R Var xj) x ≠ 0⟩
  using ⟨coeff lp' x = 0⟩
  by (cases x = xj) (auto simp add: coeff-Var2)
qed

```

```

qed
have vars (subst-var  $x_j$   $lp'$   $lp$ )  $\subseteq$  (vars  $lp$   $- \{x_j\}$ )  $\cup$  vars  $lp'$ 
  unfolding subst-var-def
  using coeff-zero[of  $lp + coeff lp x_j *R lp' - coeff lp x_j *R Var x_j x_j$ ]
  using coeff-zero[of  $lp' x_j$ ]
  using *
  by auto
moreover
have  $\bigwedge x. [x \notin \text{vars} (lp + coeff lp x_j *R lp' - coeff lp x_j *R Var x_j x_j); x \in \text{vars} lp; x \notin \text{vars} lp'] \implies x = x_j$ 
proof-
  fix  $x$ 
  assume  $x \in \text{vars} lp$   $x \notin \text{vars} lp'$ 
  then have  $coeff lp x \neq 0$   $coeff lp' x = 0$ 
    using coeff-zero
    by auto
  assume  $x \notin \text{vars} (lp + coeff lp x_j *R lp' - coeff lp x_j *R Var x_j x_j)$ 
  then have  $coeff (lp + coeff lp x_j *R lp' - coeff lp x_j *R Var x_j x_j) x = 0$ 
    using coeff-zero
    by force
  then show  $x = x_j$ 
    using ⟨coeff lp x ≠ 0⟩ ⟨coeff lp' x = 0⟩
    by (cases x = x_j) (auto simp add: coeff-Var2)
qed
then have  $\text{vars} lp - \{x_j\} - \text{vars} lp' \subseteq \text{vars} (\text{subst-var } x_j \text{ } lp' \text{ } lp)$ 
  by (auto simp add: subst-var-def)
ultimately show  $\text{vars} lp - \{x_j\} - \text{vars} lp' \subseteq \text{vars} (\text{subst-var } x_j \text{ } lp' \text{ } lp)$ 
   $\subseteq \text{vars} lp - \{x_j\} \cup \text{vars} lp'$ 
  by simp
next
fix  $v$   $x_j$   $lp'$   $lp$ 
show  $v x_j = lp' \{ v \} \longrightarrow lp \{ v \} = (\text{subst-var } x_j \text{ } lp' \text{ } lp) \{ v \}$ 
  unfolding subst-var-def
  using valuate-minus[of  $lp + coeff lp x_j *R lp' coeff lp x_j *R Var x_j v$ ]
  using valuate-add[of  $lp coeff lp x_j *R lp' v$ ]
  using valuate-scaleRat[of  $coeff lp x_j lp' v$ ] valuate-scaleRat[of  $coeff lp x_j Var x_j v$ ]
  using valuate-Var[of  $x_j v$ ]
  by auto
next
fix  $x_j$   $lp$   $lp'$ 
assume  $x_j \notin \text{vars} lp$ 
hence 0:  $coeff lp x_j = 0$  using coeff-zero by blast
show subst-var  $x_j$   $lp'$   $lp = lp$ 
  unfolding subst-var-def 0 by simp
next
fix  $x_j$   $lp$   $x$   $lp'$ 
assume  $x_j \in \text{vars} lp$   $x \in \text{vars} lp' - \text{vars} lp$ 
hence x:  $x \neq x_j$  and 0:  $coeff lp x = 0$  and no0:  $coeff lp x_j \neq 0$   $coeff lp' x \neq 0$ 

```

```

using coeff-zero by blast+
from x have 00: coeff (Var xj) x = 0 using coeff-Var2 by auto
show x ∈ vars (subst-var xj lp' lp)
  unfolding subst-var-def coeff-zero[symmetric]
  by (simp add: 0 00 no0)
qed (simp-all add: subst-var-eq-code-def)

definition rhs-eq-val where
  rhs-eq-val v xi c e ≡ let xj = lhs e; aij = coeff (rhs e) xi in
    ⟨v⟩ xj + aij *R (c - ⟨v⟩ xi)

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 unfold-locales
  fix v x c e
  assume ⟨v⟩ ⊨e e
  then show rhs-eq-val v x c e = rhs e {⟨v⟩(x := c)} {
    unfolding rhs-eq-val-def Let-def
    using valuate-update-x[of rhs e x ⟨v⟩ ⟨v⟩(x := c)]
    by (auto simp add: satisfies-eq-def)
  qed (auto simp: update-code-def assert-bound'-code-def assert-bound-code-def)

sublocale PivotUpdateMinVars < Check check
proof (rule Check-check)
  show RhsEqVal rhs-eq-val ..
qed

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
  by (unfold-locales, auto simp: pivot-tableau-code-def pivot-code-def subst-var-eq-code-def)

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

```

date-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

by (*unfold-locales, auto simp: pivot-and-update-code-def*)

sublocale *Update < AssertBoundNoLhs assert-bound*

proof (*rule update-to-assert-bound-no-lhs*)

show *Pivot eq-idx-for-lvar pivot-code ..*

qed

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

by (*unfold-locales*) (*simp-all add: check-code-def check'-code-def*)

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

by (*unfold-locales, auto simp: assert-code-def*)

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

AssertAllState.assert-all assert-all-state-code = assert-all-code

by (*unfold-locales*) (*simp-all add: assert-bound-loop-code-def assert-all-state-code-def*)

assert-all-code-def)

primrec

monom-to-atom:: $QDelta\ ns\text{-}constraint \Rightarrow QDelta\ atom$ **where**
 $monom\text{-}to\text{-}atom\ (LEQ\text{-}ns\ l\ r) = (if\ (monom\text{-}coeff\ l < 0)\ then\ (Geq\ (monom\text{-}var\ l)\ (r /R monom\text{-}coeff\ l))\ else\ (Leq\ (monom\text{-}var\ l)\ (r /R monom\text{-}coeff\ l)))$
 $| monom\text{-}to\text{-}atom\ (GEQ\text{-}ns\ l\ r) = (if\ (monom\text{-}coeff\ l < 0)\ then\ (Leq\ (monom\text{-}var\ l)\ (r /R monom\text{-}coeff\ l))\ else\ (Geq\ (monom\text{-}var\ l)\ (r /R monom\text{-}coeff\ l)))$

primrec

qdelta-constraint-to-atom:: $QDelta\ ns\text{-}constraint \Rightarrow var \Rightarrow QDelta\ atom$ **where**
 $qdelta\text{-}constraint\text{-}to\text{-}atom\ (LEQ\text{-}ns\ l\ r)\ v = (if\ (is\text{-}monom\ l)\ then\ (monom\text{-}to\text{-}atom\ (LEQ\text{-}ns\ l\ r))\ else\ (Leq\ v\ r))$
 $| qdelta\text{-}constraint\text{-}to\text{-}atom\ (GEQ\text{-}ns\ l\ r)\ v = (if\ (is\text{-}monom\ l)\ then\ (monom\text{-}to\text{-}atom\ (GEQ\text{-}ns\ l\ r))\ else\ (Geq\ v\ r))$

primrec

qdelta-constraint-to-atom':: $QDelta\ ns\text{-}constraint \Rightarrow var \Rightarrow QDelta\ atom$ **where**
 $qdelta\text{-}constraint\text{-}to\text{-}atom'\ (LEQ\text{-}ns\ l\ r)\ v = (Leq\ v\ r)$
 $| qdelta\text{-}constraint\text{-}to\text{-}atom'\ (GEQ\text{-}ns\ l\ r)\ v = (Geq\ v\ r)$

fun *linear-poly-to-eq*:: *linear-poly* $\Rightarrow var \Rightarrow eq$ **where**
 $linear\text{-}poly\text{-}to\text{-}eq\ p\ v = (v, p)$

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

primrec *zero-satisfies* :: '*a* :: *lrv ns-constraint* $\Rightarrow bool$ **where**
 $zero\text{-}satisfies\ (LEQ\text{-}ns\ l\ r) \longleftrightarrow 0 \leq r$
 $| zero\text{-}satisfies\ (GEQ\text{-}ns\ l\ r) \longleftrightarrow 0 \geq r$

lemma *zero-satisfies*: *poly c = 0* \implies *zero-satisfies c* $\implies v \models_{ns} c$
by (*cases c, auto simp: valuate-zero*)

lemma *not-zero-satisfies*: *poly c = 0* $\implies \neg$ *zero-satisfies c* $\implies \neg v \models_{ns} c$
by (*cases c, auto simp: valuate-zero*)

```

fun
  preprocess' :: ('i,QDelta) i-ns-constraint list  $\Rightarrow$  var  $\Rightarrow$  'i istate where
    preprocess' [] v = IState v [] []
    | 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  $\Rightarrow$ 
          IState v' t' (((i,qdelta-constraint-to-atom h v) # a') m' u'
          | None  $\Rightarrow$  IState (v' + 1) (linear-poly-to-eq p v' # t')
            (((i,qdelta-constraint-to-atom h v') # a') (m' (p  $\mapsto$  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  $\Rightarrow$ 
    IState v' t' (((i,qdelta-constraint-to-atom' h v) # a') m' u'
    | None  $\Rightarrow$  IState (v' + 1) (linear-poly-to-eq p v' # t')
      (((i,qdelta-constraint-to-atom' h v') # a') (m' (p  $\mapsto$  v')) u')
  ) by (cases h, auto simp add: Let-def split: option.splits)

```

```

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  $\Rightarrow$  '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]:  $v \models_{ns} (\text{normalize-ns-constraint } c) \longleftrightarrow v \models_{ns} (c :: 'a :: lrv ns-constraint)$ 
proof –
  let ?c = coeff (poly c) (max-var (poly c))
  consider (0) ?c = 0 | (pos) ?c > 0 | (neg) ?c < 0 by linarith
  thus ?thesis
  proof cases
    case 0
    thus ?thesis by (cases c, auto)
  next
    case pos
    from pos have id:  $a / R ?c \leq b / R ?c \longleftrightarrow (a :: 'a) \leq b$  for a b
      using scaleRat-leq1 by fastforce
    show ?thesis using pos id by (cases c, auto simp: Let-def valuate-scaleRat id)
  next
    case neg
    from neg have id:  $a / R ?c \leq b / R ?c \longleftrightarrow (a :: 'a) \geq b$  for a b
      using scaleRat-leq2 by fastforce
    show ?thesis using neg id by (cases c, auto simp: Let-def valuate-scaleRat id)
  qed
qed

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

lemma i-satisfies-normalize-ns-constraint[simp]:  $Iv \models_{inss} (\text{map-prod } id \text{ normalize-ns-constraint } 'cs) \longleftrightarrow Iv \models_{inss} cs$ 
  by (cases Iv, force)

abbreviation max-var:: QDelta ns-constraint  $\Rightarrow$  var where
  max-var C  $\equiv$  Abstract-Linear-Poly.max-var (poly C)

fun
  start-fresh-variable :: ('i, QDelta) i-ns-constraint list  $\Rightarrow$  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 \Rightarrow tableau \times ((*'i, QDelta*) *i*-atom list) \times *'i* list **where**
preprocess-part-1 l \equiv 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*
 by (cases *h*) auto

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

lemma *fresh-var-monoinc*:
 FirstFreshVariable (preprocess' *cs start*) \geq *start*
 by (induct *cs*) (auto simp add: Let-def split: option.splits)

abbreviation *vars-constraints* **where**
 vars-constraints cs \equiv \bigcup (set (map *vars* (map poly *cs*)))

lemma *start-fresh-variable-fresh*:
 \forall *var* \in *vars-constraints* (flat-list *cs*). *var* $<$ start-fresh-variable *cs*
 using max-var-max
 by (induct *cs*, auto simp add: max-def) force+

lemma *rvars-tableau-vars-constraints*:
 rvars (Tableau (preprocess' *cs start*)) \subseteq *vars-constraints* (flat-list *cs*)
 by (induct *cs start* rule: preprocess'.induct) (auto simp add: rvars-def Let-def split: option.splits)

lemma *lvars-tableau-ge-start*:
 \forall *var* \in *lvars* (Tableau (preprocess' *cs start*)). *var* \geq *start*
 by (induct *cs start* rule: preprocess'.induct) (auto simp add: Let-def lvars-def fresh-var-monoinc split: option.splits)

lemma *rhs-no-zero-tableau-start*:
 $0 \notin$ *rhs* ' set (Tableau (preprocess' *cs start*))
 by (induct *cs start* rule: preprocess'.induct, auto simp add: Let-def rvars-def fresh-var-monoinc split: option.splits)

lemma *first-fresh-variable-not-in-lvars*:
 \forall *var* \in *lvars* (Tableau (preprocess' *cs start*)). FirstFreshVariable (preprocess' *cs start*) $>$ *var*
 by (induct *cs start* rule: preprocess'.induct) (auto simp add: Let-def lvars-def split: option.splits)

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*

```

 $\neg \text{is-monom} (\text{poly } h)$ 
 $\text{shows } v \models_{ns} h$ 
 $\text{using assms}$ 
 $\text{by (cases } h) \text{ (auto simp add: satisfies-eq-def split: if-splits)}$ 

lemma qdelta-constraint-to-atom-monom:
 $\text{assumes is-monom} (\text{poly } h)$ 
 $\text{shows } v \models_a \text{qdelta-constraint-to-atom } h \text{ var} \longleftrightarrow v \models_{ns} h$ 
proof (cases h)
 $\text{case (EQ-ns l a)}$ 
 $\text{then show ?thesis}$ 
 $\text{using assms}$ 
 $\text{using monom-valuate[of - v]}$ 
 $\text{apply auto}$ 
 $\text{using scaleRat-leq2[of a /R monom-coeff l v (monom-var l) monom-coeff l]}$ 
 $\text{using divide-leq1[of monom-coeff l v (monom-var l) a]}$ 
 $\text{apply (force, simp add: divide-rat-def)}$ 
 $\text{using scaleRat-leq1[of v (monom-var l) a /R monom-coeff l monom-coeff l]}$ 
 $\text{using is-monom-monom-coeff-not-zero[of l]}$ 
 $\text{using divide-leq[of monom-coeff l v (monom-var l) a]}$ 
 $\text{using is-monom-monom-coeff-not-zero[of l]}$ 
 $\text{by (simp-all add: divide-rat-def)}$ 
next
 $\text{case (GEQ-ns l a)}$ 
 $\text{then show ?thesis}$ 
 $\text{using assms}$ 
 $\text{using monom-valuate[of - v]}$ 
 $\text{apply auto}$ 
 $\text{using scaleRat-leq2[of v (monom-var l) a /R monom-coeff l monom-coeff l]}$ 
 $\text{using divide-geq1[of a monom-coeff l v (monom-var l)]}$ 
 $\text{apply (force, simp add: divide-rat-def)}$ 
 $\text{using scaleRat-leq1[of a /R monom-coeff l v (monom-var l) monom-coeff l]}$ 
 $\text{using is-monom-monom-coeff-not-zero[of l]}$ 
 $\text{using divide-geq[of a monom-coeff l v (monom-var l)]}$ 
 $\text{using is-monom-monom-coeff-not-zero[of l]}$ 
 $\text{by (simp-all add: divide-rat-def)}$ 
qed

```

lemma preprocess'-Tableau-Poly-Mapping-None: (Poly-Mapping (preprocess' cs start))
 $p = \text{None}$
 $\implies \text{linear-poly-to-eq } p \in \text{set } (\text{Tableau } (\text{preprocess}' \text{ cs start}))$
 $\text{by (induct cs start rule: preprocess'.induct, auto simp: Let-def split: option.splits if-splits)}$

lemma preprocess'-Tableau-Poly-Mapping-Some: (Poly-Mapping (preprocess' cs start))
 $p = \text{Some } v$
 $\implies \text{linear-poly-to-eq } p \in \text{set } (\text{Tableau } (\text{preprocess}' \text{ cs start}))$
 $\text{by (induct cs start rule: preprocess'.induct, auto simp: Let-def split: option.splits if-splits)}$

```

lemma preprocess'-Tableau-Poly-Mapping-Some': (Poly-Mapping (preprocess' cs start)) p = Some v
   $\implies \exists h. \text{poly } h = p \wedge \neg \text{is-monom} (\text{poly } h) \wedge \text{qdelta-constraint-to-atom } h v \in \text{flat} (\text{set} (\text{Atoms} (\text{preprocess}' cs start)))$ 
  by (induct cs start rule: preprocess'.induct, auto simp: Let-def split: option.splits if-splits)

lemma not-one-le-zero-qdelta:  $\neg (1 \leq (0 :: QDelta))$  by code-simp

lemma one-zero-contra[dest,consumes 2]:  $1 \leq x \implies (x :: QDelta) \leq 0 \implies \text{False}$ 

using order.trans[of 1 x 0] not-one-lezero-qdelta by simp

lemma i-preprocess'-sat:
  assumes  $(I, v) \models_{ias} \text{set} (\text{Atoms} (\text{preprocess}' s start)) v \models_t \text{Tableau} (\text{preprocess}' s start)$ 
     $I \cap \text{set} (\text{UnsatIndices} (\text{preprocess}' s start)) = \{\}$ 
  shows  $(I, v) \models_{ins} \text{set } s$ 
  using assms
  by (induct s start rule: preprocess'.induct)
  (auto simp add: Let-def satisfies-atom-set-def satisfies-tableau-def qdelta-constraint-to-atom-monom
    sat-atom-sat-eq-sat-constraint-non-monom
    split: if-splits option.splits dest!: preprocess'-Tableau-Poly-Mapping-Some zero-satisfies)

lemma preprocess'-sat:
  assumes  $v \models_{as} \text{flat} (\text{set} (\text{Atoms} (\text{preprocess}' s start))) v \models_t \text{Tableau} (\text{preprocess}' s start) \text{set} (\text{UnsatIndices} (\text{preprocess}' s start)) = \{\}$ 
  shows  $v \models_{ns} \text{flat} (\text{set } s)$ 
  using i-preprocess'-sat[of UNIV v s start] assms by simp

lemma sat-constraint-valuation:
  assumes  $\forall var \in vars (\text{poly } c). v1 var = v2 var$ 
  shows  $v1 \models_{ns} c \longleftrightarrow v2 \models_{ns} c$ 
  using assms
  using valuate-depend
  by (cases c) (force)+

lemma atom-var-first:
  assumes  $a \in \text{flat} (\text{set} (\text{Atoms} (\text{preprocess}' cs start))) \forall var \in vars\text{-constraints} (\text{flat-list } cs). var < start$ 
  shows atom-var a < FirstFreshVariable (preprocess' cs start)
  using assms
  proof(induct cs arbitrary: a)
  case (Cons hh t a)
  obtain i h where hh: hh = (i,h) by force
  let ?s = preprocess' t start
  show ?case
  proof(cases a ∈ flat (set (Atoms ?s)))

```

```

case True
then show ?thesis
  using Cons(1)[of a] Cons(3) hh
  by (auto simp add: Let-def split: option.splits)
next
  case False
    consider (monom) is-monom (poly h) | (normal)  $\neg$  is-monom (poly h) poly h
     $\neq 0$  (Poly-Mapping ?s) (poly h) = None
    | (old) var where  $\neg$  is-monom (poly h) poly h  $\neq 0$  (Poly-Mapping ?s) (poly h) = Some var
    | (zero)  $\neg$  is-monom (poly h) poly h = 0
    by auto
  then show ?thesis
  proof cases
    case monom
    from Cons(3) monom-var-in-vars hh monom
    have monom-var (poly h) < start by auto
    moreover from False have a = qdelta-constraint-to-atom h (FirstFreshVariable (preprocess' t start))
      using Cons(2) hh monom by (auto simp: Let-def)
      ultimately show ?thesis
        using fresh-var-monoinc[of start t] hh monom
        by (cases a; cases h) (auto simp add: Let-def )
    next
      case normal
      have a = qdelta-constraint-to-atom h (FirstFreshVariable (preprocess' t start))
        using False normal Cons(2) hh by (auto simp: Let-def)
      then show ?thesis using hh normal
        by (cases a; cases h) (auto simp add: Let-def )
    next
      case (old var)
      from preprocess'-Tableau-Poly-Mapping-Some'[OF old(3)]
      obtain h' where poly h' = poly h qdelta-constraint-to-atom h' var  $\in$  flat (set (Atoms ?s))
        by blast
      from Cons(1)[OF this(2)] Cons(3) this(1) old(1)
      have var: var < FirstFreshVariable ?s by (cases h', auto)
      have a = qdelta-constraint-to-atom h var
        using False old Cons(2) hh by (auto simp: Let-def)
      then have a: atom-var a = var using old by (cases a; cases h; auto simp: Let-def)
        show ?thesis unfolding a hh by (simp add: old Let-def var)
    next
      case zero
      from False show ?thesis using Cons(2) hh zero by (auto simp: Let-def split: if-splits)
        qed
        qed
      qed simp

```

```

lemma satisfies-tableau-satisfies-tableau:
  assumes  $v1 \models_t t \forall var \in tvars\ t. v1\ var = v2\ var$ 
  shows  $v2 \models_t t$ 
  using assms
  using valuate-depend[of -  $v1\ v2$ ]
  by (force simp add: lvars-def rvars-def satisfies-eq-def satisfies-tableau-def)

lemma preprocess'-unsat-indices:
  assumes  $i \in set(UNSATIndices(preprocess'\ s start))$ 
  shows  $\neg(\{i\}, v) \models_{INSS} set\ s$ 
  using assms
  proof (induct s start rule: preprocess'.induct)
  case (?j h t v)
    then show ?case by (auto simp: Let-def not-zero-satisfies split: if-splits option.splits)
  qed simp

lemma preprocess'-unsat:
  assumes  $(I, v) \models_{INSS} set\ s\ vars-constraints(flat-list\ s) \subseteq V\ \forall var \in V. var < start$ 
  shows  $\exists v'. (\forall var \in V. v\ var = v'\ var)$ 
     $\wedge v' \models_{AS} restrict-to\ I\ (set(Atoms\ (preprocess'\ s start)))$ 
     $\wedge v' \models_t Tableau\ (preprocess'\ s start)$ 
  using assms
  proof (induct s)
  case Nil
  show ?case
    by (auto simp add: satisfies-atom-set-def satisfies-tableau-def)
  next
  case (Cons hh t)
  obtain i h where hh = (i, h) by force
  from Cons hh obtain v' where
    var:  $(\forall var \in V. v\ var = v'\ var)$ 
    and v'-as:  $v' \models_{AS} restrict-to\ I\ (set(Atoms\ (preprocess'\ t start)))$ 
    and v'-t:  $v' \models_t Tableau\ (preprocess'\ t start)$ 
    and vars-h:  $vars-constraints[h] \subseteq V$ 
    by auto
  from Cons(?)[unfolded hh]
  have i:  $i \in I \implies v \models_{NS} h$  by auto
  have  $\forall var \in vars(poly\ h). v\ var = v'\ var$ 
    using  $\langle(\forall var \in V. v\ var = v'\ var)\rangle$  Cons(3) hh
    by auto
  then have vh-v'h:  $v \models_{NS} h \longleftrightarrow v' \models_{NS} h$ 
    by (rule sat-constraint-valuation)
  show ?case
  proof (cases is-monom (poly h))
  case True
    then have id: is-monom (poly h) = True by simp

```

```

show ?thesis
  unfolding hh preprocess'.simps Let-def id if-True istate.simps istate.sel
  proof (intro exI[of - v'] conjI v'-t var satisfies-atom-restrict-to-Cons[OF v'-as])
    assume i ∈ I
    from i[OF this] var vh-v'h
    show v' ⊨_a qdelta-constraint-to-atom h (FirstFreshVariable (preprocess' t start))
      unfolding qdelta-constraint-to-atom-monom[OF True] by auto
    qed
  next
  case False
  then have id: is-monom (poly h) = False by simp
  let ?s = preprocess' t start
  let ?x = FirstFreshVariable ?s
  show ?thesis
  proof (cases poly h = 0)
    case zero: False
    hence id': (poly h = 0) = False by simp
    let ?look = (Poly-Mapping ?s) (poly h)
    show ?thesis
    proof (cases ?look)
      case None
      let ?y = poly h { v' }
      let ?v' = v'{?x:=?y}
      show ?thesis unfolding preprocess'.simps hh Let-def id id' if-False istate.simps istate.sel None option.simps
        proof (rule exI[of - ?v'], intro conjI satisfies-atom-restrict-to-Cons satisfies-tableau-Cons)
          show vars': (∀ var∈V. v var = ?v' var)
            using ⟨(∀ var∈V. v var = v' var)⟩
            using fresh-var-monoinc[of start t]
            using Cons(4)
            by auto
          {
            assume i ∈ I
            from vh-v'h i[OF this] False
            show ?v' ⊨_a qdelta-constraint-to-atom h (FirstFreshVariable (preprocess' t start))
              by (cases h, auto)
          }
          let ?atoms = restrict-to I (set (Atoms (preprocess' t start)))
          show ?v' ⊨_as ?atoms
            unfolding satisfies-atom-set-def
          proof
            fix a
            assume a ∈ ?atoms
            then have v' ⊨_a a
              using ⟨v' ⊨_as ?atoms⟩ hh by (force simp add: satisfies-atom-set-def)
            then show ?v' ⊨_a a
        qed
      qed
    qed
  qed
qed

```

```

using ⟨a ∈ ?atoms⟩ atom-var-first[of a t start]
using Cons(3) Cons(4)
by (cases a) auto
qed
show ?v' ⊨e linear-poly-to-eq (poly h) (FirstFreshVariable (preprocess' t
start))
using Cons(3) Cons(4)
using valuate-depend[of poly h v' v'(FirstFreshVariable (preprocess' t
start) := (poly h) { v' })]
using fresh-var-monoinc[of start t] hh
by (cases h) (force simp add: satisfies-eq-def) +
have FirstFreshVariable (preprocess' t start) ∉ tvars (Tableau (preprocess'
t start))
using first-fresh-variable-not-in-lvars[of t start]
using Cons(3) Cons(4)
using vars-tableau-vars-constraints[of t start]
using fresh-var-monoinc[of start t]
by force
then show ?v' ⊨t Tableau (preprocess' t start)
using ⟨v' ⊨t Tableau (preprocess' t start)⟩
using satisfies-tableau-satisfies-tableau[v' Tableau (preprocess' t start)
?v']
by auto
qed
next
case (Some var)
from preprocess'-Tableau-Poly-Mapping-Some[OF Some]
have linear-poly-to-eq (poly h) var ∈ set (Tableau ?s) by auto
with v'-t[unfolded satisfies-tableau-def]
have v'-h-var: v' ⊨e linear-poly-to-eq (poly h) var by auto
show ?thesis unfolding preprocess'.simpss hh Let-def id id' if-False is-
tate.simps istate.sel Some option.simps
proof (intro exI[of - v'] conjI var v'-t satisfies-atom-restrict-to-Cons satis-
fies-tableau-Cons v'-as)
assume i ∈ I
from vh-v'h i[OF this] False v'-h-var
show v' ⊨a qdelta-constraint-to-atom h var
by (cases h, auto simp: satisfies-eq-iff)
qed
qed
next
case zero: True
hence id': (poly h = 0) = True by simp
show ?thesis
proof (cases zero-satisfies h)
case True
hence id'': zero-satisfies h = True by simp
show ?thesis
unfolding hh preprocess'.simpss Let-def id id' id'' if-True if-False istate.simps

```

```

 $i\text{state}.sel$ 
  by (intro exI[of - v'] conjI v'-t var v'-as])
next
  case False
  hence id'': zero-satisfies  $h = \text{False}$  by simp
  {
    assume  $i \in I$ 
    from  $i[\text{OF this}] \text{ not-zero-satisfies } [\text{OF zero } \text{False}]$  have  $\text{False}$  by simp
  } note  $\text{no-}I = \text{this}$ 
  show ?thesis
  unfolding hh preprocess'.simps Let-def id id' id'' if-True if-False i $\text{state}.simps$ 
 $i\text{state}.sel$ 
  proof (rule Cons(1)[OF -- Cons(4)])
    show  $(I, v) \models_{\text{inss}} \text{set } t$  using Cons(2) by auto
    show vars-constraints (map snd t)  $\subseteq V$  using Cons(3) by force
    qed
    qed
    qed
    qed
    qed
  qed

lemma lvars-distinct:
  distinct (map lhs (Tableau (preprocess' cs start)))
  using first-fresh-variable-not-in-lvars[where ?'a = 'a]
  by (induct cs, auto simp add: Let-def lvars-def) (force split: option.splits)

lemma normalized-tableau-preprocess':  $\Delta (\text{Tableau} (\text{preprocess}' cs (\text{start-fresh-variable } cs)))$ 
proof -
  let  $?s = \text{start-fresh-variable } cs$ 
  show ?thesis
  using lvars-distinct[of cs ?s]
  using lvars-tableau-ge-start[of cs ?s]
  using vars-tableau-vars-constraints[of cs ?s]
  using start-fresh-variable-fresh[of cs]
  unfolding normalized-tableau-def Let-def
  by (meson disjoint-iff linorder-not-le rhs-no-zero-tableau-start subset-eq)
qed

```

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:  $\Delta t$  and  $t: t = t1 @ (x,p) \# t2$ 
  and  $t': t' = t1 @ t2$ 
  and  $tv: tv = (\lambda v. \text{upd } x (p \setminus \langle v \rangle)) v$ 
  and  $x\text{-as}: x \notin \text{atom-var } ' \text{snd } ' \text{set as}$ 
shows  $\Delta t'$  — new tableau is normalized
   $\langle w \rangle \models_t t' \implies \langle tv w \rangle \models_t t$  — solution of new tableau is translated to solution of old tableau
   $(I, \langle w \rangle) \models_{ias} \text{set as} \implies (I, \langle tv w \rangle) \models_{ias} \text{set as}$  — solution translation also works

```

for bounds

```

 $v \models_t t \implies v \models_t t'$  — solution of old tableau is also solution for new tableau
proof —
  have  $\langle tv w \rangle = \langle w \rangle (x := p \setminus \langle w \rangle)$  unfolding  $tv$   $map2fun-def'$  by auto
  from  $norm$ 
  show  $\Delta t' \text{ unfolding } t t' \text{ normalized-tableau-def by (auto simp: lvars-def rvars-def)}$ 
  show  $v \models_t t \implies v \models_t t' \text{ unfolding } t t' \text{ satisfies-tableau-def by auto}$ 
  from  $norm$  have  $dist: distinct (map lhs t) lvars t \cap rvars t = \{\}$ 
    unfolding  $normalized-tableau-def$  by auto
  from  $x\text{-as}$  have  $x\text{-as}: x \notin atom-var` snd ` (set as \cap I \times UNIV)$  by auto
  have  $(I, \langle tv w \rangle) \models_{ias} set as \longleftrightarrow (I, \langle w \rangle) \models_{ias} set as$  unfolding  $i\text{-satisfies-atom-set.simps}$ 
     $satisfies-atom-set-def$ 
  proof (intro ball-cong[OF refl])
    fix  $a$ 
    assume  $a \in snd ` (set as \cap I \times UNIV)$ 
    with  $x\text{-as}$  have  $x \neq atom-var a$  by auto
    then show  $\langle tv w \rangle \models_a a = \langle w \rangle \models_a a$  unfolding  $tv$ 
      by (cases a, auto)
  qed
  then show  $(I, \langle w \rangle) \models_{ias} set as \implies (I, \langle tv w \rangle) \models_{ias} set as$  by blast
  assume  $w: \langle w \rangle \models_t t'$ 
  from  $dist(2)[unfolded t]$  have  $xp: x \notin vars p$ 
    unfolding  $lvars-def rvars-def$  by auto
  {
    fix  $eq$ 
    assume  $mem: eq \in set t1 \cup set t2$ 
    then have  $eq \in set t' \text{ unfolding } t' \text{ by auto}$ 
    with  $w$  have  $w: \langle w \rangle \models_e eq$  unfolding  $satisfies-tableau-def$  by auto
    obtain  $y q$  where  $eq: eq = (y, q)$  by force
    from  $mem[unfolded eq] dist(1)[unfolded t]$  have  $xy: x \neq y$  by force
    from  $mem[unfolded eq] dist(2)[unfolded t]$  have  $xq: x \notin vars q$ 
      unfolding  $lvars-def rvars-def$  by auto
    from  $w$  have  $\langle tv w \rangle \models_e eq$  unfolding  $tv$   $eq$  satisfies-eq-iff using  $xy xq$ 
      by (auto intro!: valuate-depend)
  }
  moreover
  have  $\langle tv w \rangle \models_e (x, p)$  unfolding  $satisfies-eq-iff$   $tv$  using  $xp$ 
    by (auto intro!: valuate-depend)
  ultimately
  show  $\langle tv w \rangle \models_t t$  unfolding  $t$  satisfies-tableau-def by auto
qed

definition  $pivot-tableau-eq :: tableau \Rightarrow eq \Rightarrow tableau \Rightarrow var \Rightarrow tableau \times eq \times tableau$  where
   $pivot-tableau-eq t1 eq t2 x \equiv let eq' = pivot-eq eq x; m = map (\lambda e. subst-var-eq x (rhs eq') e) in (m t1, eq', m t2)$ 

lemma  $pivot-tableau-eq$ : assumes  $t: t = t1 @ eq \# t2 t' = t1' @ eq' \# t2'$ 
```

```

and  $x: x \in rvars-eq eq$  and  $norm: \Delta t$  and  $pte: pivot-tableau-eq t1 eq t2 x = (t1',eq',t2')$ 
shows  $\Delta t' lhs eq' = x (v :: 'a :: lrv valuation) \models_t t' \longleftrightarrow v \models_t t$ 
proof -
  let  $?s = \lambda t. State t undefined undefined undefined undefined$ 
  let  $?y = lhs eq$ 
  have  $yl: ?y \in lvars t$  unfolding  $t lvars-def$  by auto
  from  $norm$  have  $eq-t12: ?y \notin lhs ' (set t1 \cup set t2)$ 
    unfolding  $normalized-tableau-def t lvars-def$  by auto
  have  $eq: eq-for-lvar-code t ?y = eq$ 
    by (metis (mono-tags, lifting) EqForLVarDefault.eq-for-lvar Un-insert-right
 $eq-t12$ 
      image-iff insert-iff list.set(2) set-append t(1) yl)
  have  $*: (?y, b) \in set t1 \implies ?y \in lhs ' (set t1)$  for  $b t1$ 
    by (metis image-eqI lhs.simps)
  have  $pivot: pivot-tableau-code ?y x t = t'$ 
    unfolding Pivot'Default.pivot-tableau-def Let-def eq using pte[symmetric]
    unfolding  $t$  pivot-tableau-eq-def Let-def using eq-t12 by (auto dest!: *)
  note  $thms = Pivot'Default.pivot-vars' Pivot'Default.pivot-tableau$ 
  note  $thms = thms[unfolded Pivot'Default.pivot-def, of ?s t, simplified,$ 
    OF norm yl, unfolded eq, OF x, unfolded pivot]
  from  $thms(1)$   $thms(2)[of v]$  show  $\Delta t' v \models_t t' \longleftrightarrow v \models_t t$  by auto
  show  $lhs eq' = x$  using pte[symmetric]
    unfolding  $t$  pivot-tableau-eq-def Let-def pivot-eq-def by auto
qed

function preprocess-opt :: var set  $\Rightarrow$  tableau  $\Rightarrow$  tableau  $\Rightarrow$  tableau  $\times ((var, 'a :: lrv) mapping \Rightarrow (var, 'a) mapping)$  where
  preprocess-opt X t1 [] = (t1,id)
  | preprocess-opt X t1 ((x,p) # t2) = (if  $x \notin X$  then
    case preprocess-opt X t1 t2 of (t,tv)  $\Rightarrow$  (t, ( $\lambda v. upd x (p \setminus \langle v \rangle) v$ ) o tv)
    else case find ( $\lambda x. x \notin X$ ) (Abstract-Linear-Poly.vars-list p) of
      None  $\Rightarrow$  preprocess-opt X ((x,p) # t1) t2
      | Some y  $\Rightarrow$  case pivot-tableau-eq t1 (x,p) t2 y of
        (tt1,(z,q),tt2)  $\Rightarrow$  case preprocess-opt X tt1 tt2 of (t,tv)  $\Rightarrow$  (t, ( $\lambda v. upd z (q \setminus \langle v \rangle) v$ ) o tv))
      by pat-completeness auto
  termination by (relation measure ( $\lambda (X,t1,t2). length t2$ ), auto simp: pivot-tableau-eq-def Let-def)
  lemma preprocess-opt: assumes  $X = atom-var$  ‘ snd ‘ set as
    preprocess-opt X t1 t2 = (t',tv)  $\Delta t t = rev t1 @ t2$ 
  shows  $\Delta t'$ 
    ( $\langle w \rangle :: 'a :: lrv valuation$ )  $\models_t t' \implies \langle tv w \rangle \models_t t$ 
    ( $I, \langle w \rangle$ )  $\models_{ias} set as \implies (I, \langle tv w \rangle) \models_{ias} set as$ 
     $v \models_t t \implies (v :: 'a valuation) \models_t t'$ 
    using assms
  proof (atomize(full), induct X t1 t2 arbitrary: t tv w rule: preprocess-opt.induct)

```

```

case (1 X t1 t tv)
then show ?case by (auto simp: normalized-tableau-def lvars-def rvars-def satisfies-tableau-def
simp flip: rev-map)
next
case (2 X t1 x p t2 t tv w)
note IH = 2(1–3)
note X = 2(4)
note res = 2(5)
have norm: △ t by fact
have t: t = rev t1 @ (x, p) # t2 by fact
show ?case
proof (cases x ∈ X)
case False
with res obtain tv' where res: preprocess-opt X t1 t2 = (t', tv') and
tv: tv = (λv. Mapping.update x (p {⟨v⟩}) v) o tv'
by (auto split: prod.splits)
note delete = delete-lhs-var[OF norm t refl refl False[unfolded X]]
note IH = IH(1)[OF False X res delete(1) refl]
from delete(2)[of tv' w] delete(3)[of I tv' w] delete(4)[of v] IH[of w]
show ?thesis unfolding tv o-def
by auto
next
case True
then have ¬ x ∉ X by simp
note IH = IH(2–3)[OF this]
show ?thesis
proof (cases find (λx. x ∉ X) (Abstract-Linear-Poly.vars-list p))
case None
with res True have pre: preprocess-opt X ((x, p) # t1) t2 = (t', tv) by auto
from t have t: t = rev ((x, p) # t1) @ t2 by simp
from IH(1)[OF None X pre norm t]
show ?thesis .
next
case (Some z)
from Some[unfolded find-Some-iff] have zX: z ∉ X and z ∈ set (Abstract-Linear-Poly.vars-list p)
unfolding set-conv-nth by auto
then have z: z ∈ rvars-eq (x, p) by (simp add: set-vars-list)
obtain tt1 z' q tt2 where pte: pivot-tableau-eq t1 (x, p) t2 z = (tt1,(z',q),tt2)
by (cases pivot-tableau-eq t1 (x, p) t2 z, auto)
then have pte-rev: pivot-tableau-eq (rev t1) (x, p) t2 z = (rev tt1,(z',q),tt2)
unfolding pivot-tableau-eq-def Let-def by (auto simp: rev-map)
note eq = pivot-tableau-eq[OF t refl z norm pte-rev]
then have z': z' = z by auto
note eq = eq(1,3)[unfolded z']
note pte = pte[unfolded z']
note pte-rev = pte-rev[unfolded z']
note delete = delete-lhs-var[OF eq(1) refl refl refl zX[unfolded X]]

```

```

from res[unfolded preprocess-opt.simps Some option.simps pte] True
obtain tv' where res: preprocess-opt X tt1 tt2 = (t', tv') and
  tv: tv = ( $\lambda v.$  Mapping.update z (q {<v>}) v) o tv'
  by (auto split: prod.splits)
  note IH = IH(2)[OF Some, unfolded pte, OF refl refl refl X res delete(1)
refl]
  from IH[of w] delete(2)[of tv' w] delete(3)[of I tv' w] delete(4)[of v] show
?thesis
  unfolding tv o-def eq(2) by auto
  qed
  qed
qed

```

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)  $\triangle$  t
shows  $\triangle$  t'
  ( $\langle w \rangle :: 'a :: lrv\ valuation$ )  $\models_t t' \implies \langle tv\ w \rangle \models_t t$ 
  (I,  $\langle w \rangle$ )  $\models_{ias} \text{set as} \implies (I, \langle tv\ w \rangle) \models_{ias} \text{set as}$ 
  v  $\models_t t \implies (v :: 'a\ valuation) \models_t t'$ 
using preprocess-opt[OF refl assms(1)[unfolded preprocess-part-2-def] assms(2)]
by auto

```

```

definition preprocess :: ('i,QDelta) i-ns-constraint list  $\Rightarrow$  -  $\times$  -  $\times$  (-  $\Rightarrow$  (var,QDelta)mapping)
 $\times$  'i list where
  preprocess l = (case preprocess-part-1 (map (map-prod id normalize-ns-constraint)
l) of
  (t,as,ui)  $\Rightarrow$  case preprocess-part-2 as t of (t,tv)  $\Rightarrow$  (t,as,tv,ui))

```

```

lemma preprocess:
assumes id: preprocess cs = (t, as, trans-v, ui)
shows  $\triangle$  t
  fst ` set as  $\cup$  set ui  $\subseteq$  fst ` set cs
  distinct-indices-ns (set cs)  $\implies$  distinct-indices-atoms (set as)
  I  $\cap$  set ui = {}  $\implies$  (I,  $\langle v \rangle$ )  $\models_{ias} \text{set as} \implies$ 
     $\langle v \rangle \models_t t \implies (I, \langle trans-v\ v \rangle) \models_{inss} \text{set cs}$ 
    i  $\in$  set ui  $\implies$   $\nexists v. (\{i\}, v) \models_{inss} \text{set cs}$ 
     $\exists v. (I, v) \models_{inss} \text{set cs} \implies \exists v'. (I, v') \models_{ias} \text{set as} \wedge v' \models_t t$ 
proof -
  define ncs where ncs = map (map-prod id normalize-ns-constraint) cs
  have ncs: fst ` set ncs = fst ` set cs  $\wedge$  Iv. Iv  $\models_{inss} \text{set ncs} \leftrightarrow$  Iv  $\models_{inss} \text{set cs}$ 
    unfolding ncs-def by force auto
  from id obtain t1 where part1: preprocess-part-1 ncs = (t1,as,ui)
    unfolding preprocess-def by (auto simp: ncs-def split: prod.splits)
  from id[unfolded preprocess-def part1 split ncs-def[symmetric]]
  have part-2: preprocess-part-2 as t1 = (t,trans-v)
    by (auto split: prod.splits)
  have norm:  $\triangle$  t1 using normalized-tableau-preprocess' part1
    by (auto simp: preprocess-part-1-def Let-def)

```

```

note part-2 = preprocess-part-2[OF part-2 norm]
show  $\Delta t$  by fact
have unsat:  $(I, \langle v \rangle) \models_{ias} set as \Rightarrow \langle v \rangle \models_t t1 \Rightarrow I \cap set ui = \{\} \Rightarrow (I, \langle v \rangle)$ 
 $\models_{inss} set ncs$  for  $v$ 
using part1 [unfolded preprocess-part-1-def Let-def, simplified] i-preprocess'-sat[of  $I$ ] by blast
with part-2(2,3) show  $I \cap set ui = \{\} \Rightarrow (I, \langle v \rangle) \models_{ias} set as \Rightarrow \langle v \rangle \models_t t$ 
 $\Rightarrow (I, \langle trans-v v \rangle) \models_{inss} set cs$ 
by (auto simp: ncs)
from part1 [unfolded preprocess-part-1-def Let-def] obtain var where
as: as = Atoms (preprocess' ncs var) and ui: ui = UnsatIndices (preprocess' ncs var) by auto
note min-defs = distinct-indices-atoms-def distinct-indices-ns-def
have min1: (distinct-indices-ns (set ncs)  $\rightarrow$  ( $\forall k a. (k, a) \in set as \rightarrow (\exists v p. a = qdelta\text{-constraint-to-atom } p v \wedge (k, p) \in set ncs$ 
 $\wedge (\neg is\text{-monom } (poly p) \rightarrow Poly\text{-Mapping } (preprocess' ncs var) (poly p) = Some v)))$ )
 $\wedge fst`set as \cup set ui \subseteq fst`set ncs$ 
unfolding as ui
proof (induct ncs var rule: preprocess'.induct)
case (2 i h t v)
hence sub:  $fst`set (Atoms (preprocess' t v)) \cup set (UnsatIndices (preprocess' t v)) \subseteq fst`set t$  by auto
show ?case
proof (intro conjI impI allI, goal-cases)
show  $fst`set (Atoms (preprocess' ((i, h) \# t) v)) \cup set (UnsatIndices (preprocess' ((i, h) \# t) v)) \subseteq fst`set ((i, h) \# t)$ 
using sub by (auto simp: Let-def split: option.splits)
next
case (1 k a)
hence min': distinct-indices-ns (set t) unfolding min-defs list.simps by blast
note IH = 2[THEN conjunct1, rule-format, OF min']
show ?case
proof (cases (k, a)  $\in$  set (Atoms (preprocess' t v)))
case True
from IH[OF this] show ?thesis
by (force simp: Let-def split: option.splits if-split)
next
case new: False
with 1(2) have ki:  $k = i$  by (auto simp: Let-def split: if-splits option.splits)
show ?thesis
proof (cases is-monom (poly h))
case True
thus ?thesis using new 1(2) by (auto simp: Let-def True intro!: exI)
next
case no-monom: False
thus ?thesis using new 1(2) by (auto simp: Let-def no-monom split: option.splits if-splits intro!: exI)
qed

```

```

qed
qed
qed (auto simp: min-defs)
then show fst ` set as ∪ set ui ⊆ fst ` set cs by (auto simp: ncs)
{
  assume mini: distinct-indices-ns (set cs)
  have mini: distinct-indices-ns (set ncs) unfolding distinct-indices-ns-def
  proof (intro impI allI, goal-cases)
    case (1 n1 n2 i)
    from 1(1) obtain c1 where c1: (i,c1) ∈ set cs and n1: n1 = normalize-ns-constraint c1
      unfolding ncs-def by auto
    from 1(2) obtain c2 where c2: (i,c2) ∈ set cs and n2: n2 = normalize-ns-constraint c2
      unfolding ncs-def by auto
    from mini[unfolded distinct-indices-ns-def, rule-format, OF c1 c2]
    show ?case unfolding n1 n2
      by (cases c1; cases c2; auto simp: normalize-ns-constraint.simps Let-def)
  qed
  note min = min1[THEN conjunct1, rule-format, OF this]
  show distinct-indices-atoms (set as)
    unfolding distinct-indices-atoms-def
  proof (intro allI impI)
    fix i a b
    assume a: (i,a) ∈ set as and b: (i,b) ∈ set as
    from min[OF a] obtain v p where aa: a = qdelta-constraint-to-atom p v (i, p) ∈ set ncs
      ¬ is-monom (poly p) ==> Poly-Mapping (preprocess' ncs var) (poly p) = Some v
        by auto
      from min[OF b] obtain w q where bb: b = qdelta-constraint-to-atom q w (i, q) ∈ set ncs
        ¬ is-monom (poly q) ==> Poly-Mapping (preprocess' ncs var) (poly q) = Some w
        by auto
      from min1[unfolded distinct-indices-ns-def, rule-format, OF aa(2) bb(2)]
      have *: poly p = poly q ns-constraint-const p = ns-constraint-const q by auto
      show atom-var a = atom-var b ∧ atom-const a = atom-const b
      proof (cases is-monom (poly q))
        case True
        thus ?thesis unfolding aa(1) bb(1) using * by (cases p; cases q, auto)
      next
        case False
        thus ?thesis unfolding aa(1) bb(1) using * aa(3) bb(3) by (cases p; cases q, auto)
      qed
    qed
  }
  show i ∈ set ui ==> #v. ({i}, v) ⊨inss set cs
}

```

```

using preprocess'-unsat-indices[of i ncs] part1 unfolding preprocess-part-1-def
Let-def
  by (auto simp: ncs)
  assume  $\exists w. (I,w) \models_{inss} set cs$ 
  then obtain w where  $(I,w) \models_{inss} set cs$  by blast
  hence  $(I,w) \models_{inss} set ncs$  unfolding ncs .
  from preprocess'-unsat[OF this - start-fresh-variable-fresh, of ncs]
  have  $\exists v'. (I,v') \models_{ias} set as \wedge v' \models_t t1$ 
    using part1
    unfolding preprocess-part-1-def Let-def by auto
    then show  $\exists v'. (I,v') \models_{ias} set as \wedge v' \models_t t$ 
      using part-2(4) by auto
qed

```

```

interpretation PreprocessDefault: Preprocess preprocess
  by (unfold-locales; rule preprocess, auto)

```

```

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
  by unfold-locales

```

primrec

```

constraint-to-qdelta-constraint:: constraint  $\Rightarrow QDelta$  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  $\Rightarrow ('i, QDelta)$  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  $\Rightarrow ('i, QDelta)$  i-ns-constraint list where
  to-ns l  $\equiv$  concat (map i-constraint-to-qdelta-constraint l)

```

primrec

```

δ0-val :: QDelta ns-constraint  $\Rightarrow QDelta$  valuation  $\Rightarrow rat$  where
  δ0-val (LEQ-ns lll rrr) vl = δ0 lll{vl} rrr
  | δ0-val (GEQ-ns lll rrr) vl = δ0 rrr lll{vl}

```

```

primrec
   $\delta_0\text{-val-min} :: QDelta\ ns\text{-constraint list} \Rightarrow QDelta\ valuation \Rightarrow rat\ \mathbf{where}$ 
   $\delta_0\text{-val-min} []\ vl = 1$ 
   $\mid \delta_0\text{-val-min} (h\#t)\ vl = min (\delta_0\text{-val-min} t\ vl) (\delta_0\text{-val} h\ vl)$ 

abbreviation vars-list-constraints where
  vars-list-constraints cs  $\equiv$  remdups (concat (map Abstract-Linear-Poly.vars-list (map poly cs)))

definition
  from-ns ::(var, QDelta) mapping  $\Rightarrow QDelta\ ns\text{-constraint list} \Rightarrow (var, rat)\ mapping\ \mathbf{where}$ 
  from-ns vl cs  $\equiv$  let  $\delta = \delta_0\text{-val-min} cs \langle vl \rangle$  in
    Mapping.tabulate (vars-list-constraints cs) ( $\lambda\ var.\ val (\langle vl \rangle\ var)\ \delta$ )

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 unfold-locales
{
  fix ics :: 'i i-constraint list and v' and I
  let ?to-ns = to-ns ics
  let ?flat = set ?to-ns
  assume sat:  $(I, \langle v' \rangle) \models_{inss} ?flat$ 
  define cs where cs = map snd (filter ( $\lambda\ ic.\ fst\ ic \in I$ ) ics)
  define to-ns' where to-ns: to-ns' =  $(\lambda\ l.\ concat (map constraint-to-qdelta-constraint$  l))
  show  $(I, \langle from-ns\ v' (flat-list ?to-ns) \rangle) \models_{ics} set\ ics$  unfolding i-satisfies-cs.simps
  proof
    let ?listf = map ( $\lambda C.\ case\ C\ of\ (LEQ-ns\ l\ r) \Rightarrow (l\{\langle v' \rangle\}, r)$ 
       $\mid (GEQ-ns\ l\ r) \Rightarrow (r, l\{\langle v' \rangle\})$ 
    )
    let ?to-ns =  $\lambda\ ics.\ to-ns' (map\ snd\ (filter\ (\lambda ic.\ fst\ ic \in I)\ ics))$ 
    let ?list = ?listf (to-ns' cs)
    let ?f-list = flat-list (to-ns ics)
    let ?f-list = ?listf ?f-list
    obtain i-list where i-list: ?list = i-list by force
    obtain f-list where f-list: ?f-list = f-list by force
    have if-list: set i-list  $\subseteq$  set f-list unfolding
      i-list[symmetric] f-list[symmetric] to-ns-def to-ns-set-map set-concat cs-def
      by (intro image-mono, force)
    have  $\bigwedge\ qd1\ qd2.\ (qd1, qd2) \in set\ ?list \implies qd1 \leq qd2$ 
    proof-
      fix qd1 qd2
      assume  $(qd1, qd2) \in set\ ?list$ 
      then show  $qd1 \leq qd2$ 
        using sat unfolding cs-def
      proof(induct ics)
        case Nil

```

```

then show ?case
  by (simp add: to-ns)
next
  case (Cons h t)
  obtain i c where h: h = (i,c) by force
  from Cons(2) consider (ic) (qd1,qd2) ∈ set (?listf (?to-ns [(i,c)]))
  | (t) (qd1,qd2) ∈ set (?listf (?to-ns t))
  unfolding to-ns h set-map set-concat by fastforce
  then show ?case
  proof cases
    case t
    from Cons(1)[OF this] Cons(3) show ?thesis unfolding to-ns-def by
  auto
  next
    case ic
    note ic = ic[unfolded to-ns, simplified]
    from ic have i: (i ∈ I) = True by (cases i ∈ I, auto)
    note ic = ic[unfolded i if-True, simplified]
    from Cons(3)[unfolded h] i have ⟨v'⟩ ⊨nss set (to-ns' [c])
      unfolding i-satisfies-ns-constraints.simps unfolding to-ns to-ns-def
    by force
      with ic show ?thesis by (induct c) (auto simp add: to-ns)
    qed
    qed
    qed
  then have l1: ε > 0  $\implies$  ε ≤ (δ-min ?list)  $\implies$   $\forall$  qd1 qd2. (qd1, qd2) ∈ set
  ?list  $\longrightarrow$  val qd1 ε ≤ val qd2 ε for ε
  unfolding i-list
  by (simp add: delta-gt-zero delta-min[of i-list])
  have δ-min ?flist ≤ δ-min ?list unfolding f-list i-list
  by (rule delta-min-mono[OF if-list])
  from l1[OF delta-gt-zero this]
  have l1:  $\forall$  qd1 qd2. (qd1, qd2) ∈ set ?list  $\longrightarrow$  val qd1 (δ-min f-list) ≤ val qd2
  (δ-min f-list)
  unfolding f-list .
  have δ0-val-min (flat-list (to-ns ics)) ⟨v'⟩ = δ-min f-list unfolding f-list[symmetric]
  proof(induct ics)
    case Nil
    show ?case
      by (simp add: to-ns-def)
  next
    case (Cons h t)
    then show ?case
      by (cases h; cases snd h) (auto simp add: to-ns-def)
  qed
  then have l2: from-ns v' ?f-list = Mapping.tabulate (vars-list-constraints
  ?f-list) (λ var. val ⟨v'⟩ var) (δ-min f-list)
  by (auto simp add: from-ns-def)
  fix c

```

```

assume  $c \in \text{restrict-to } I \ (\text{set } ics)$ 
then obtain  $i$  where  $i : i \in I \ \text{and} \ mem : (i, c) \in \text{set } ics$  by auto
from  $mem$  show  $\langle \text{from-ns } v' ?f\text{-list} \rangle \models_c c$ 
proof (induct c)
  case ( $LT lll rrr$ )
    then have  $(lll\{\langle v' \rangle\}, (QDelta.QDelta rrr (-1))) \in \text{set } ?list$  using  $i$  un-
folding  $cs\text{-def}$ 
    by (force simp add: to-ns)
    then have  $\text{val}(lll\{\langle v' \rangle\}) (\delta\text{-min } f\text{-list}) \leq \text{val}(QDelta.QDelta rrr (-1))$ 
  ( $\delta\text{-min } f\text{-list}$ )
    using  $l1$ 
    by simp
  moreover
  have  $lll\{(\lambda x. \text{val}(\langle v' \rangle x)) (\delta\text{-min } f\text{-list})\} =$ 
     $lll\{\langle \text{from-ns } v' ?f\text{-list} \rangle\}$ 
  proof (rule valuate-depend, rule)
    fix  $x$ 
    assume  $x \in \text{vars } lll$ 
    then show  $\text{val}(\langle v' \rangle x) (\delta\text{-min } f\text{-list}) = \langle \text{from-ns } v' ?f\text{-list} \rangle x$ 
      using  $l2$ 
      using  $LT$ 
    by (auto simp add: comp-def lookup-tabulate restrict-map-def set-vars-list
          to-ns-def map2fun-def')
  qed
  ultimately
  have  $lll\{\langle \text{from-ns } v' ?f\text{-list} \rangle\} \leq (\text{val}(QDelta.QDelta rrr (-1)) (\delta\text{-min } f\text{-list}))$ 
    by (auto simp add: valuate-rat-validate)
  then show  $?case$ 
    using  $\text{delta-gt-zero}[of f\text{-list}]$ 
    by (simp add: val-def)
  next
    case ( $GT lll rrr$ )
    then have  $((QDelta.QDelta rrr 1), lll\{\langle v' \rangle\}) \in \text{set } ?list$  using  $i$  un-
folding  $cs\text{-def}$ 
    by (force simp add: to-ns)
    then have  $\text{val}(lll\{\langle v' \rangle\}) (\delta\text{-min } f\text{-list}) \geq \text{val}(QDelta.QDelta rrr 1) (\delta\text{-min }$ 
 $f\text{-list})$ 
    using  $l1$ 
    by simp
  moreover
  have  $lll\{(\lambda x. \text{val}(\langle v' \rangle x)) (\delta\text{-min } f\text{-list})\} =$ 
     $lll\{\langle \text{from-ns } v' ?f\text{-list} \rangle\}$ 
  proof (rule valuate-depend, rule)
    fix  $x$ 
    assume  $x \in \text{vars } lll$ 
    then show  $\text{val}(\langle v' \rangle x) (\delta\text{-min } f\text{-list}) = \langle \text{from-ns } v' ?f\text{-list} \rangle x$ 
      using  $l2$ 
      using  $GT$ 
    by (auto simp add: lookup-tabulate comp-def restrict-map-def set-vars-list)

```

```

to-ns-def map2fun-def')
qed
ultimately
have  $lll\{\langle from-ns v' ?f-list \rangle\} \geq val (QDelta.QDelta rrr 1) (\delta\text{-min } f\text{-list})$ 
  using l2
  by (simp add: valuate-rat-valuate)
then show ?case
  using delta-gt-zero[of f-list]
  by (simp add: val-def)
next
  case (LEQ lll rrr)
  then have  $(lll\{\langle v' \rangle\}, (QDelta.QDelta rrr 0)) \in set ?list$  using i unfolding
cs-def
  by (force simp add: to-ns)
  then have  $val (lll\{\langle v' \rangle\}) (\delta\text{-min } f\text{-list}) \leq val (QDelta.QDelta rrr 0) (\delta\text{-min } f\text{-list})$ 
    using l1
    by simp
moreover
have  $lll\{(\lambda x. val (\langle v' \rangle x) (\delta\text{-min } f\text{-list}))\} =$ 
 $lll\{\langle from-ns v' ?f-list \rangle\}$ 
proof (rule valuate-depend, rule)
  fix x
  assume  $x \in vars lll$ 
  then show  $val (\langle v' \rangle x) (\delta\text{-min } f\text{-list}) = \langle from-ns v' ?f-list \rangle x$ 
    using l2
    using LEQ
    by (auto simp add: lookup-tabulate comp-def restrict-map-def set-vars-list
to-ns-def map2fun-def')
qed
ultimately
have  $lll\{\langle from-ns v' ?f-list \rangle\} \leq val (QDelta.QDelta rrr 0) (\delta\text{-min } f\text{-list})$ 
  using l2
  by (simp add: valuate-rat-valuate)
then show ?case
  by (simp add: val-def)
next
  case (GEQ lll rrr)
  then have  $((QDelta.QDelta rrr 0), lll\{\langle v' \rangle\}) \in set ?list$  using i unfolding
cs-def
  by (force simp add: to-ns)
  then have  $val (lll\{\langle v' \rangle\}) (\delta\text{-min } f\text{-list}) \geq val (QDelta.QDelta rrr 0) (\delta\text{-min } f\text{-list})$ 
    using l1
    by simp
moreover
have  $lll\{(\lambda x. val (\langle v' \rangle x) (\delta\text{-min } f\text{-list}))\} =$ 
 $lll\{\langle from-ns v' ?f-list \rangle\}$ 
proof (rule valuate-depend, rule)

```

```

fix x
assume x ∈ vars lll
then show val ((v' x) (δ-min f-list)) = ⟨from-ns v' ?f-list⟩ x
  using l2
  using GEQ
  by (auto simp add: lookup-tabulate comp-def restrict-map-def set-vars-list
to-ns-def map2fun-def')
qed
ultimately
have lll{⟨from-ns v' ?f-list⟩} ≥ val (QDelta.QDelta rrr 0) (δ-min f-list)
  using l2
  by (simp add: valuate-rat-valuate)
then show ?case
  by (simp add: val-def)
next
case (EQ lll rrr)
then have ((QDelta.QDelta rrr 0), lll{v'}) ∈ set ?list and
  (lll{v'}, (QDelta.QDelta rrr 0)) ∈ set ?list using i unfolding cs-def
  by (force simp add: to-ns)+
then have val (lll{v'}) (δ-min f-list) ≥ val (QDelta.QDelta rrr 0) (δ-min
f-list) and
  val (lll{v'}) (δ-min f-list) ≤ val (QDelta.QDelta rrr 0) (δ-min f-list)
  using l1
  by simp-all
moreover
have lll{⟨λx. val ((v' x) (δ-min f-list))⟩} =
  lll{⟨from-ns v' ?f-list⟩}
proof (rule valuate-depend, rule)
fix x
assume x ∈ vars lll
then show val ((v' x) (δ-min f-list)) = ⟨from-ns v' ?f-list⟩ x
  using l2
  using EQ
  by (auto simp add: lookup-tabulate comp-def restrict-map-def set-vars-list
to-ns-def map2fun-def')
qed
ultimately
have lll{⟨from-ns v' ?f-list⟩} ≥ val (QDelta.QDelta rrr 0) (δ-min f-list)
and
lll{⟨from-ns v' ?f-list⟩} ≤ val (QDelta.QDelta rrr 0) (δ-min f-list)
  using l1
  by (auto simp add: valuate-rat-valuate)
then show ?case
  by (simp add: val-def)
qed
qed
} note sat = this
fix cs :: ('i × constraint) list
have set-to-ns: set (to-ns cs) = { (i,n) | i n c. (i,c) ∈ set cs ∧ n ∈ set

```

```

(constraint-to-qdelta-constraint c) }
  unfolding to-ns-def by auto
  show indices: fst ` set (to-ns cs) = fst ` set cs
  proof
    show fst ` set (to-ns cs) ⊆ fst ` set cs
      unfolding set-to-ns by force
    {
      fix i
      assume i ∈ fst ` set cs
      then obtain c where (i,c) ∈ set cs by force
        hence i ∈ fst ` set (to-ns cs) unfolding set-to-ns by (cases c; force)
    }
    thus fst ` set cs ⊆ fst ` set (to-ns cs) by blast
  qed
  {
    assume dist: distinct-indices cs
    show distinct-indices-ns (set (to-ns cs)) unfolding distinct-indices-ns-def
    proof (intro allI impI conjI notI)
      fix n1 n2 i
      assume (i,n1) ∈ set (to-ns cs) (i,n2) ∈ set (to-ns cs)
      then obtain c1 c2 where i: (i,c1) ∈ set cs (i,c2) ∈ set cs
      and n: n1 ∈ set (constraint-to-qdelta-constraint c1) n2 ∈ set (constraint-to-qdelta-constraint
c2)
        unfolding set-to-ns by auto
      from dist
      have distinct (map fst cs) unfolding distinct-indices-def by auto
      with i have c12: c1 = c2 by (metis eq-key-imp-eq-value)
      note n = n[unfolded c12]
      show poly n1 = poly n2 using n by (cases c2, auto)
      show ns-constraint-const n1 = ns-constraint-const n2 using n by (cases c2,
auto)
    qed
  } note mini = this
  fix I mode
  assume unsat: minimal-unsat-core-ns I (set (to-ns cs))
  note unsat = unsat[unfolded minimal-unsat-core-ns-def indices]
  hence indices: I ⊆ fst ` set cs by auto
  show minimal-unsat-core I cs
    unfolding minimal-unsat-core-def
  proof (intro conjI indices impI allI, clarify)
    fix v
    assume v: (I,v) ⊨ics set cs
    let ?v = λvar. QDelta.QDelta (v var) 0
    have (I,?v) ⊨ins (set (to-ns cs)) using v
    proof(induct cs)
      case (Cons ic cs)
      obtain i c where ic: ic = (i,c) by force
      from Cons(2-) ic
      have rec: (I,v) ⊨ics set cs and c: i ∈ I ==> v ⊨c c by auto
    
```

```

{
fix jn
assume i: i ∈ I and jn ∈ set (to-ns [(i,c)])
then have jn ∈ set (i-constraint-to-qdelta-constraint (i,c))
  unfolding to-ns-def by auto
then obtain n where n: n ∈ set (constraint-to-qdelta-constraint c)
  and jn: jn = (i,n) by force
from c[OF i] have c: v ⊨c c by force
from c n jn have ?v ⊨ns snd jn
  by (cases c) (auto simp add: less-eq-QDelta-def to-ns-def valuate-valuate-rat
valuate-minus zero-QDelta-def)
} note main = this
from Cons(1)[OF rec] have IH: (I,?v) ⊨inss set (to-ns cs) .
show ?case unfolding i-satisfies-ns-constraints.simps
proof (intro ballI)
fix x
assume x ∈ snd ` (set (to-ns (ic # cs)) ∩ I × UNIV)
then consider (1) x ∈ snd ` (set (to-ns cs) ∩ I × UNIV)
| (2) x ∈ snd ` (set (to-ns [(i,c)]) ∩ I × UNIV)
  unfolding ic to-ns-def by auto
then show ?v ⊨ns x
proof cases
  case 1
  then show ?thesis using IH by auto
next
  case 2
  then obtain jn where x: snd jn = x and jn ∈ set (to-ns [(i,c)]) ∩ I ×
UNIV
    by auto
  with main[of jn] show ?thesis unfolding to-ns-def by auto
qed
qed
qed (simp add: to-ns-def)
with unsat show False unfolding minimal-unsat-core-ns-def by simp blast
next
fix J
assume *: distinct-indices cs J ⊂ I
hence distinct-indices-ns (set (to-ns cs))
  using mini by auto
with * unsat obtain v where model: (J, v) ⊨inss set (to-ns cs) by blast
define w where w = Mapping.Mapping (λ x. Some (v x))
have v = ⟨w⟩ unfolding w-def map2fun-def
  by (intro ext, transfer, auto)
with model have model: (J, ⟨w⟩) ⊨inss set (to-ns cs) by auto
from sat[OF this]
show ∃ v. (J, v) ⊨ics set cs by blast
qed
qed

```

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 \Rightarrow 'i list + (var, rat) mapping **where**
 simplex-index = solve-exec-code

lemma simplex-index:
 $\text{simplex-index } cs = \text{Unsat } I \Rightarrow \text{set } I \subseteq \text{fst} \cdot \text{set } cs \wedge \neg (\exists v. (\text{set } I, v) \models_{ics} \text{set } cs) \wedge$
 $(\text{distinct-indices } cs \rightarrow (\forall J \subset \text{set } I. (\exists v. (J, v) \models_{ics} \text{set } cs)))$ — minimal
 unsat core
 $\text{simplex-index } cs = \text{Sat } v \Rightarrow \langle v \rangle \models_{cs} (\text{snd} \cdot \text{set } cs)$ — satisfying assingment
proof (unfold simplex-index-def)
assume solve-exec-code cs = Unsat I
from SolveExec'Default.simplex-unsat0[OF this]
have core: minimal-unsat-core (set I) cs **by** auto
then show set I \subseteq fst · set cs \wedge $\neg (\exists v. (\text{set } I, v) \models_{ics} \text{set } cs) \wedge$
 $(\text{distinct-indices } cs \rightarrow (\forall J \subset \text{set } I. \exists v. (J, v) \models_{ics} \text{set } cs))$
unfolding minimal-unsat-core-def **by** auto
next
assume solve-exec-code cs = Sat v
from SolveExec'Default.simplex-sat0[OF this]
show $\langle v \rangle \models_{cs} (\text{snd} \cdot \text{set } cs)$.
qed

Simplex version where indices will be created

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

lemma simplex:
 $\text{simplex } cs = \text{Unsat } I \Rightarrow \neg (\exists v. v \models_{cs} \text{set } cs)$ — unsat of original constraints
 $\text{simplex } cs = \text{Unsat } I \Rightarrow \text{set } I \subseteq \{0..<\text{length } cs\} \wedge \neg (\exists v. v \models_{cs} \{cs ! i \mid i. i \in \text{set } I\})$
 $\wedge (\forall J \subset \text{set } I. \exists v. v \models_{cs} \{cs ! i \mid i. i \in J\})$ — minimal unsat core
 $\text{simplex } cs = \text{Sat } v \Rightarrow \langle v \rangle \models_{cs} \text{set } cs$ — satisfying assignment
proof (unfold simplex-def)
let ?cs = zip [0..<length cs] cs
assume simplex-index ?cs = Unsat I
from simplex-index(1)[OF this]
have index: set I $\subseteq \{0 .. < \text{length } cs\}$ **and**
core: $\nexists v. v \models_{cs} (\text{snd} \cdot (\text{set } ?cs \cap \text{set } I \times \text{UNIV}))$
 $(\text{distinct-indices } (\text{zip } [0..<\text{length } cs] cs) \rightarrow (\forall J \subset \text{set } I. \exists v. v \models_{cs} (\text{snd} \cdot (\text{set } ?cs \cap J \times \text{UNIV}))))$
by (auto simp flip: set-map)
note core(2)
also have distinct-indices (zip [0..<length cs] cs)
unfolding distinct-indices-def set-zip **by** (auto simp: set-conv-nth)

```

also have ( $\forall J \subset set I. \exists v. v \models_{cs} (snd ` (set ?cs \cap J \times UNIV)) =$ 
 $(\forall J \subset set I. \exists v. v \models_{cs} \{ cs ! i \mid i. i \in J\})$  using index
by (intro all-cong1 imp-cong ex-cong1 arg-cong[of - -  $\lambda x. - \models_{cs} x$ ] refl, force
simp: set-zip)
finally have core': ( $\forall J \subset set I. \exists v. v \models_{cs} \{ cs ! i \mid i. i \in J\}$ ) .
note unsat = unsat-mono[OF core(1)]
show  $\neg (\exists v. v \models_{cs} set cs)$ 
by (rule unsat, auto simp: set-zip)
show set I  $\subseteq \{0..<length cs\} \wedge \neg (\exists v. v \models_{cs} \{cs ! i \mid i. i \in set I\})$ 
 $\wedge (\forall J \subset set I. \exists v. v \models_{cs} \{cs ! i \mid i. i \in J\})$ 
by (intro conjI index core', rule unsat, auto simp: set-zip)
next
assume simplex-index (zip [0..<length cs] cs) = Sat v
from simplex-index(2)[OF this]
show  $\langle v \rangle \models_{cs} set cs$  by (auto simp flip: set-map)
qed

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 -  $\Rightarrow$  True | Unsat -  $\Rightarrow$  False
by eval

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 -  $\Rightarrow$  False | Unsat I  $\Rightarrow$  set I = {1,3,4} — Constraints 1,3,4 are unsat
core
by eval

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  $\Rightarrow$  's and
  assert-s :: ('i,'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
  precond-s :: tableau  $\Rightarrow$  bool 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
assumes
  assert-s-ok: invariant-s t as s  $\Rightarrow$  assert-s a s = Inr s'  $\Rightarrow$ 
    invariant-s t (insert a as) s' and
  assert-s-unsat: invariant-s t as s  $\Rightarrow$  assert-s a s = Unsat I  $\Rightarrow$ 
    minimal-unsat-core-tabl-atoms (set I) t (insert a as) and
  check-s-ok: invariant-s t as s  $\Rightarrow$  check-s s = (s', None)  $\Rightarrow$ 
    checked-s t as s' and
  check-s-unsat: invariant-s t as s  $\Rightarrow$  check-s s = (s', Some I)  $\Rightarrow$ 
    weak-invariant-s t as s'  $\wedge$  minimal-unsat-core-tabl-atoms (set I) t as and
  init-s: precond-s t  $\Rightarrow$  checked-s t {} (init-s t) and
  solution-s: checked-s t as s  $\Rightarrow$  solution-s s = v  $\Rightarrow$   $\langle v \rangle \models_t t \wedge \langle v \rangle \models_{as} \text{Simplex.flat}$  as and
  backtrack-s: checked-s t as s  $\Rightarrow$  checkpoint-s s = c
     $\Rightarrow$  weak-invariant-s t bs s'  $\Rightarrow$  backtrack-s c s' = s''  $\Rightarrow$  as  $\subseteq$  bs  $\Rightarrow$  invariant-s t as s'' and
  weak-invariant-s: invariant-s t as s  $\Rightarrow$  weak-invariant-s t as s and
  checked-invariant-s: checked-s t as s  $\Rightarrow$  invariant-s t as s
begin

fun assert-all-s :: ('i,'a) i-atom list  $\Rightarrow$  's  $\Rightarrow$  'i list + 's where
  assert-all-s [] s = Inr s
  | assert-all-s (a # as) s = (case assert-s a s of Unsat I  $\Rightarrow$  Unsat I
    | Inr s'  $\Rightarrow$  assert-all-s as s')

lemma assert-all-s-ok: invariant-s t as s  $\Rightarrow$  assert-all-s bs s = Inr s'  $\Rightarrow$ 
  invariant-s t (set bs  $\cup$  as) s'
proof (induct bs arbitrary: s as)
  case (Cons b bs s as)
    from Cons(3) obtain s'' where ass: assert-s b s = Inr s'' and rec: assert-all-s
    bs s'' = Inr s'
    by (auto split: sum.splits)
    from Cons(1)[OF assert-s-ok[OF Cons(2) ass] rec]
    show ?case by auto
qed auto

```

```

lemma assert-all-s-unsat: invariant-s t as s  $\implies$  assert-all-s bs s = Unsat I  $\implies$ 
minimal-unsat-core-tabl-atoms (set I) t (as  $\cup$  set bs)
proof (induct bs arbitrary: s as)
  case (Cons b bs s as)
    show ?case
    proof (cases assert-s b s)
      case unsat: (Inl J)
      with Cons have J: J = I by auto
      from assert-s-unsat[OF Cons(2) unsat] J
      have min: minimal-unsat-core-tabl-atoms (set I) t (insert b as) by auto
      show ?thesis
      by (rule minimal-unsat-core-tabl-atoms-mono[OF - min], auto)
    next
      case (Inr s')
      from Cons(1)[OF assert-s-ok[OF Cons(2) Inr]] Cons(3) Inr show ?thesis by
      auto
      qed
    qed simp
  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  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state and
  init :: tableau  $\Rightarrow$  ('i,'a) state and
  check :: ('i,'a) state  $\Rightarrow$  ('i,'a) state
begin

  definition weak-invariant-s where
    weak-invariant-s t (as :: ('i,'a)i-atom set) s =
    ( $\models_{nolhs}$  s  $\wedge$ 
      $\triangle$  ( $\mathcal{T}$  s)  $\wedge$ 
      $\nabla$  s  $\wedge$ 
      $\Diamond$  s  $\wedge$ 
      $(\forall v :: (var \Rightarrow 'a). v \models_t \mathcal{T} s \longleftrightarrow v \models_t t)$   $\wedge$ 
     index-valid as s  $\wedge$ 
     Simplex.flat as  $\doteq \mathcal{B}$  s  $\wedge$ 
     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  $\wedge$   $\neg \mathcal{U}$  s)

  definition checked-s where
    checked-s t as s = (invariant-s t as s  $\wedge$   $\models$  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 :: -  $\Rightarrow$  ('i, 'a) state  $\Rightarrow$  ('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  $\neg \mathcal{U} s \models_{nols} s \triangle (\mathcal{T} s) \nabla s \diamond s$ 
    Simplex.flat as  $\doteq \mathcal{B} s$  as  $\models_i \mathcal{BI} s$  index-valid as s
     $(\forall v :: (var \Rightarrow 'a). v \models_t \mathcal{T} s \longleftrightarrow v \models_t t)$ 
  using assms unfolding invariant-defs by auto

lemma weak-invariant-sD: assumes weak-invariant-s t as s
  shows  $\models_{nols} s \triangle (\mathcal{T} s) \nabla s \diamond s$ 
    Simplex.flat as  $\doteq \mathcal{B} s$  as  $\models_i \mathcal{BI} s$  index-valid as s
     $(\forall v :: (var \Rightarrow 'a). v \models_t \mathcal{T} s \longleftrightarrow v \models_t t)$ 
  using assms unfolding invariant-defs by auto

lemma minimal-unsat-state-core-translation: assumes
  unsat: minimal-unsat-state-core (s :: ('i, 'a::lrv)state) and
  tabl:  $\forall (v :: 'a \text{ valuation}). v \models_t \mathcal{T} s = v \models_t t$  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
    unfolding minimal-unsat-core-tabl-atoms-def
  proof (intro conjI impI notI allI; (elim exE conjE)?)
  from unsat[unfolded minimal-unsat-state-core-def]
  have unsat: unsat-state-core s
    and minimal: distinct-indices-state s  $\implies$  subsets-sat-core s
    by auto
  from unsat[unfolded unsat-state-core-def I[symmetric]]
  have Is: set I  $\subseteq$  indices-state s and unsat:  $(\exists v. (set I, v) \models_{is} s)$  by auto
  from Is index show set I  $\subseteq$  fst ' as
    using index-valid-indices-state by blast
  {
    fix v
    assume t:  $v \models_t t$  and as: (set I, v)  $\models_{ias}$  as
    from t tabl have t:  $v \models_t \mathcal{T} s$  by auto
    then have (set I, v)  $\models_{is} s$  using as imp
    using atoms-imply-bounds-index.simps satisfies-state-index.simps by blast
  }

```

```

    with unsat show False by blast
}
{
fix J
assume dist: distinct-indices-atoms as
  and J: J ⊂ set I
from J Is have J': J ⊆ indices-state s by auto
from dist index have distinct-indices-state s by (metis index-valid-distinct-indices)
with minimal have subsets-sat-core s .
from this[unfolded subsets-sat-core-def I[symmetric], rule-format, OF J]
obtain v where (J, v) ⊨_ise s by blast
from satisfying-state-valuation-to-atom-tabl[OF J' this index dist] tabl
show ∃ v. v ⊨_t t ∧ (J, v) ⊨_iaes as by blast
}
qed

sublocale Incremental-Atom-Ops
  init assert-s check-s V checkpoint-s backtrack-s △ weak-invariant-s invariant-s
  checked-s
proof (unfold-locales, goal-cases)
  case (1 t as s a s')
    from 1(2)[unfolded assert-s-def Let-def]
    have U: ¬ U (assert-bound a s) and s': s' = assert-bound a s by (auto split:
    if-splits)
    note * = invariant-sD[OF 1(1)]
    from assert-bound-nolhs-tableau-id[OF *(1–5)]
    have T: T s' = T s unfolding s' by auto
    from *(3,9)
    have △ (T s') ∀ v :: var ⇒ 'a. v ⊨_t T s' = v ⊨_t t unfolding T by blast+
    moreover from assert-bound-nolhs-sat[OF *(1–5) U]
    have ⊨_nolhs s' ◁ s' unfolding s' by auto
    moreover from assert-bound-nolhs-atoms-equiv-bounds[OF *(1–6), of a]
    have Simplex.flat (insert a as) ≡ B s' unfolding s' by auto
    moreover from assert-bound-nolhs-atoms-imply-bounds-index[OF *(1–5,7)]
    have insert a as ⊨_i BI s' unfolding s' .
    moreover from assert-bound-nolhs-tableau-valuated[OF *(1–5)]
    have ∇ s' unfolding s' .
    moreover from assert-bound-nolhs-index-valid[OF *(1–5,8)]
    have index-valid (insert a as) s' unfolding s' by auto
    moreover from U s'
    have ¬ U s' by auto
    ultimately show ?case unfolding invariant-defs by auto
next
  case (2 t as s a I)
    from 2(2)[unfolded assert-s-def Let-def]
    obtain s' where s': s' = assert-bound a s and U: U (assert-bound a s)
      and I: I = the (U_c s')
      by (auto split: if-splits)
    note * = invariant-sD[OF 2(1)]

```

```

from assert-bound-nolhs-tableau-id[OF *(1-5)]
have T:  $\mathcal{T} s' = \mathcal{T} s$  unfolding  $s'$  by auto
from *(3,9)
have tabl:  $\forall v :: var \Rightarrow 'a. v \models_t \mathcal{T} s' = v \models_t t$  unfolding  $T$  by blast+
from assert-bound-nolhs-unsat[OF *(1-5,8) U]  $s'$ 
have unsat: minimal-unsat-state-core  $s'$  by auto
from assert-bound-nolhs-index-valid[OF *(1-5,8)]
have index: index-valid (insert a as)  $s'$  unfolding  $s'$  by auto
from assert-bound-nolhs-atoms-imply-bounds-index[OF *(1-5,7)]
have imp: insert a as  $\models_i \mathcal{BI} s'$  unfolding  $s'$ .
from minimal-unsat-state-core-translation[OF unsat tabl index imp I]
show ?case .
next
case (3 t as s  $s'$ )
from 3(2)[unfolded check-s-def Let-def]
have U:  $\neg \mathcal{U} (\text{check } s)$  and  $s': s' = \text{check } s$  by (auto split: if-splits)
note * = invariant-sD[OF 3(1)]
note ** = *(1,2,5,3,4)
from check-tableau-equiv[OF **] *(9)
have  $\forall v :: - \Rightarrow 'a. v \models_t \mathcal{T} s' = v \models_t t$  unfolding  $s'$  by auto
moreover from check-tableau-index-valid[OF **] *(8)
have index-valid as  $s'$  unfolding  $s'$  by auto
moreover from check-tableau-normalized[OF **]
have  $\Delta (\mathcal{T} s')$  unfolding  $s'$ .
moreover from check-tableau-valuated[OF **]
have  $\nabla s'$  unfolding  $s'$ .
moreover from check-sat[OF ** U]
have  $\models s'$  unfolding  $s'$ .
moreover from satisfies-satisfies-no-lhs[OF this] satisfies-consistent[of  $s'$ ] this
have  $\models_{nolhs} s' \diamond s'$  by blast+
moreover from check-bounds-id[OF **] *(6)
have Simplex.flat as  $\models \mathcal{B} s'$  unfolding  $s'$  by (auto simp: boundsu-def boundsl-def)
moreover from check-bounds-id[OF **] *(7)
have as  $\models_i \mathcal{BI} s'$  unfolding  $s'$  by (auto simp: boundsu-def boundsl-def indexu-def
indexl-def)
moreover from U
have  $\neg \mathcal{U} s'$  unfolding  $s'$ .
ultimately show ?case unfolding invariant-defs by auto
next
case (4 t as s  $s' I$ )
from 4(2)[unfolded check-s-def Let-def]
have s':  $s' = \text{check } s$  and U:  $\mathcal{U} (\text{check } s)$ 
and I:  $I = \text{the } (\mathcal{U}_c s')$ 
by (auto split: if-splits)
note * = invariant-sD[OF 4(1)]
note ** = *(1,2,5,3,4)
from check-unsat[OF ** U]
have unsat: minimal-unsat-state-core  $s'$  unfolding  $s'$  by auto
from check-tableau-equiv[OF **] *(9)

```

```

have tabl:  $\forall v :: - \Rightarrow 'a. v \models_t \mathcal{T} s' = v \models_t t$  unfolding  $s'$  by auto
from check-tableau-index-valid[OF **] *(8)
have index: index-valid as  $s'$  unfolding  $s'$  by auto
from check-bounds-id[OF **] *(7)
have imp: as  $\models_i \mathcal{BI} s'$  unfolding  $s'$  by (auto simp: boundsu-def boundsl-def
indexu-def indexl-def)
from check-bounds-id[OF **] *(6)
have bequiv: Simplex.flat as  $\doteq \mathcal{B} s'$  unfolding  $s'$  by (auto simp: boundsu-def
boundsl-def)
have weak-invariant-s  $t$  as  $s'$  unfolding invariant-defs
using
  check-tableau-normalized[OF **]
  check-tableau-valuated[OF **]
  check-tableau[OF **]
  unfolding  $s'[\text{symmetric}]$ 
  by (intro conjI index imp tabl bequiv, auto)
with minimal-unsat-state-core-translation[OF unsat tabl index imp I]
show ?case by auto
next
case *: (5  $t$ )
show ?case unfolding invariant-defs
using
  init-tableau-normalized[OF *]
  init-index-valid[of -  $t$ ]
  init-atoms-imply-bounds-index[of  $t$ ]
  init-satisfies[of  $t$ ]
  init-atoms-equiv-bounds[of  $t$ ]
  init-tableau-id[of  $t$ ]
  init-unsat-flag[of  $t$ ]
  init-tableau-valuated[of  $t$ ]
  satisfies-consistent[of init  $t$ ] satisfies-satisfies-no-lhs[of init  $t$ ]
  by auto
next
case (6  $t$  as  $s v$ )
then show ?case unfolding invariant-defs
by (meson atoms-equiv-bounds.simps curr-val-satisfies-state-def satisfies-state-def)
next
case (7  $t$  as  $s c bs s' s''$ )
from 7(1)[unfolded checked-s-def]
have inv-s: invariant-s  $t$  as  $s$  and  $s: \models s$  by auto
from 7(2) have  $c: c = \mathcal{B}_i s$  unfolding checkpoint-s-def by auto
have  $s'': \mathcal{T} s'' = \mathcal{T} s' \mathcal{V} s'' = \mathcal{V} s' \mathcal{B}_i s'' = \mathcal{B}_i s \mathcal{U} s'' = \text{False}$   $\mathcal{U}_c s'' = \text{None}$ 
  unfolding 7(4)[symmetric]  $c$ 
  by (atomize(full), cases  $s'$ , auto)
then have BI:  $\mathcal{B} s'' = \mathcal{B} s \mathcal{I} s'' = \mathcal{I} s$  by (auto simp: boundsu-def boundsl-def
indexu-def indexl-def)
note * = invariant-sD[OF inv-s]
note ** = weak-invariant-sD[OF 7(3)]
have  $\neg \mathcal{U} s''$  unfolding  $s''$  by auto

```

```

moreover from **(2)
have  $\triangle (\mathcal{T} s'')$  unfolding  $s''$  .
moreover from **(3)
have  $\nabla s''$  unfolding tableau-valuated-def  $s''$  .
moreover from **(8)
have  $\forall v :: - \Rightarrow 'a. v \models_t \mathcal{T} s'' = v \models_t t$  unfolding  $s''$  .
moreover from *(6)
have Simplex.flat as  $\doteq \mathcal{B} s''$  unfolding BI .
moreover from *(7)
have  $as \models_i \mathcal{BI} s''$  unfolding BI .
moreover from *(8)
have index-valid as  $s''$  unfolding index-valid-def using  $s''$  by auto
moreover from **(3)
have  $\nabla s''$  unfolding tableau-valuated-def  $s''$  .
moreover from satisfies-consistent[of  $s$ ]  $s$ 
have  $\diamond s''$  unfolding bounds-consistent-def using BI by auto
moreover
from 7(5) *(6) **(5)
have  $vB: v \models_b \mathcal{B} s' \implies v \models_b \mathcal{B} s''$  for  $v$ 
    unfolding atoms-equiv-bounds.simps satisfies-atom-set-def BI
    by force
from **(1)
have  $t: \langle \mathcal{V} s' \rangle \models_t \mathcal{T} s'$  and  $b: \langle \mathcal{V} s' \rangle \models_b \mathcal{B} s' \parallel - lvars(\mathcal{T} s')$ 
    unfolding curr-val-satisfies-no-lhs-def by auto
let  $?v = \lambda x. if x \in lvars(\mathcal{T} s') then case \mathcal{B}_l s' x of None \Rightarrow the(\mathcal{B}_u s' x) |$ 
Some  $b \Rightarrow b$  else  $\langle \mathcal{V} s' \rangle x$ 
have  $?v \models_b \mathcal{B} s'$  unfolding satisfies-bounds.simps
proof (intro allI)
fix  $x :: var$ 
show in-bounds  $x ?v (\mathcal{B} s')$ 
proof (cases  $x \in lvars(\mathcal{T} s')$ )
case True
with **(4)[unfolded bounds-consistent-def, rule-format, of  $x$ ]
show ?thesis by (cases  $\mathcal{B}_l s' x$ ; cases  $\mathcal{B}_u s' x$ , auto simp: bound-compare-defs)
next
case False
with  $b$ 
show ?thesis unfolding satisfies-bounds-set.simps by auto
qed
qed
from  $vB[Of this]$  have  $v: ?v \models_b \mathcal{B} s''$  .
have  $\langle \mathcal{V} s' \rangle \models_b \mathcal{B} s'' \parallel - lvars(\mathcal{T} s')$  unfolding satisfies-bounds-set.simps
proof clarify
fix  $x$ 
assume  $x \notin lvars(\mathcal{T} s')$ 
with  $v$ [unfolded satisfies-bounds.simps, rule-format, of  $x$ ]
show in-bounds  $x \langle \mathcal{V} s' \rangle (\mathcal{B} s'')$  by auto
qed
with  $t$  have  $\models_{nolhs} s''$  unfolding curr-val-satisfies-no-lhs-def  $s''$ 

```

```

    by auto
ultimately show ?case unfolding invariant-defs by blast
qed (auto simp: invariant-defs)

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⟩) ⊨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 (induct ias)
  case Nil
    show ?case unfolding list-map-to-fun-def by auto
  next
  case (Cons ja ias)
    obtain j a where ja: ja = (j,a) by force
    show ?case
    proof (cases j = i)
      case False
        then have id: list-map-to-fun (create-map (ja # ias)) i = list-map-to-fun
        (create-map ias) i
          unfolding ja list-map-to-fun-def
          by (auto simp: Let-def split: option.splits)
        show ?thesis unfolding id Cons unfolding ja using False by auto
      next
      case True
        with ja have ja: ja = (i,a) by auto
        have id: list-map-to-fun (create-map (ja # ias)) i = ja # list-map-to-fun
        (create-map ias) i
          unfolding ja list-map-to-fun-def
          by (auto simp: Let-def split: option.splits)
        show ?thesis unfolding id using Cons unfolding ja by auto
    qed
  qed

fun prod-wrap :: ('c ⇒ 's ⇒ 's × ('i list option))
  ⇒ 'c × 's ⇒ ('c × 's) × ('i list option) where
prod-wrap f (asi,s) = (case f asi s of (s', info) ⇒ ((asi,s'), info))

lemma prod-wrap-def': prod-wrap f (asi,s) = map-prod (Pair asi) id (f asi s)
  unfolding prod-wrap.simps by (auto split: prod.splits)

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 ⇒ 's and
  assert-s :: ('i :: linorder, '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
  weak-invariant-s :: tableau ⇒ ('i, 'a) i-atom set ⇒ 's ⇒ 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 ( $\lambda$  asitv. check-s)

definition assert-nsc where assert-nsc i = ( $\lambda$  ((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)
unfolding i-satisfies-atom-set.simps
by (rule arg-cong[of - -  $\lambda$  x. v  $\models_{as}$  x], auto)

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 (unfold-locales, goal-cases)
  case (1 nsc J S j S')
    obtain asi tv s ui where S: S = ((asi,tv,ui),s) by (cases S, auto)
    obtain t as tv' ui' where prep[simp]: preprocess nsc = (t, as, tv', ui') by (cases preprocess nsc)
      note pre = 1[unfolded S assert-nsc-def]
      from pre(2) obtain s' where
        ok: assert-all-s (list-map-to-fun asi j) s = Inr s' and S': S' = ((asi,tv,ui),s')
      and j: j ∈ set ui
        by (auto split: sum.splits if-splits)
        from pre(1)[simplified]
        have inv: invariant-s t (set as ∩ J × UNIV) s
          and asi: set (list-map-to-fun asi j) = set as ∩ {j} × UNIV invariant-as-asi
          (set as) asi tv tv' ui ui' J ∩ set ui = {} by auto
          from assert-all-s-ok[OF inv ok, unfolded asi] asi(2-) j
          show ?case unfolding invariant-nsc.simps S' prep split
            by (metis Int-insert-left Sigma-Un-distrib1 inf-sup-distrib1 insert-is-Un)
    next
      case (2 nsc J S j I)
        obtain asi s tv ui where S: S = ((asi,tv,ui),s) by (cases S, auto)
        obtain t as tv' ui' where prep[simp]: preprocess nsc = (t, as, tv', ui') by (cases preprocess nsc)
        note pre = 2[unfolded S assert-nsc-def split]
        show ?case
          proof (cases j ∈ set ui)
            case False
              with pre(2) have unsat: assert-all-s (list-map-to-fun asi j) s = Unsat I
                by (auto split: sum.splits)
              from pre(1)
              have inv: invariant-s t (set as ∩ J × UNIV) s
                and asi: set (list-map-to-fun asi j) = set as ∩ {j} × UNIV by auto
                from assert-all-s-unsat[OF inv unsat, unfolded asi]
                have minimal-unsat-core-tabl-atoms (set I) t (set as ∩ J × UNIV ∪ set as ∩ {j} × UNIV) .
                also have set as ∩ J × UNIV ∪ set as ∩ {j} × UNIV = set as ∩ insert j J × UNIV by blast
                finally have unsat: minimal-unsat-core-tabl-atoms (set I) t (set as ∩ insert j J × UNIV) .
                hence I: set I ⊆ insert j J unfolding minimal-unsat-core-tabl-atoms-def by force
                  with False pre have empty: set I ∩ set ui' = {} by auto
                  have minimal-unsat-core-tabl-atoms (set I) t (set as)
                    by (rule minimal-unsat-core-tabl-atoms-mono[OF - unsat], auto)
                  from preprocess-minimal-unsat-core[OF prep this empty]
                  have minimal-unsat-core-ns (set I) (set nsc) .
                  then show ?thesis using I by blast
    next
      case True
      with pre(2) have I: I = [j] by auto

```

```

from pre(1)[unfolded invariant-nsc.simps prep split invariant-as-asi.simps]
have set ui = set ui' by simp
with True have j: j ∈ set ui' by auto
from preprocess-unsat-index[OF prep j]
show ?thesis unfolding I by auto
qed
next
case (? nsc J S S')
then show ?case using check-s-ok unfolding check-nsc-def
  by (cases S, auto split: prod.splits, blast)
next
case (? nsc J S S' I)
obtain asi s tv ui where S: S = ((asi, tv, ui), s) by (cases S, auto)
obtain t as tv' ui' where prep[simp]: preprocess nsc = (t, as, tv', ui') by (cases
  preprocess nsc)
from 4(2)[unfolded S check-nsc-def, simplified]
obtain s' where unsat: check-s s = (s', Some I) and S': S' = ((asi, tv, ui), s')
  by (cases check-s s, auto)
note pre = 4[unfolded S check-nsc-def unsat, simplified]
from pre have
  inv: invariant-s t (set as ∩ J × UNIV) s
  by auto
from check-s-unsat[OF inv unsat]
have weak: weak-invariant-s t (set as ∩ J × UNIV) s'
  and unsat: minimal-unsat-core-tabl-atoms (set I) t (set as ∩ J × UNIV) by
auto
hence I: set I ⊆ J unfolding minimal-unsat-core-tabl-atoms-def by force
with pre have empty: set I ∩ set ui' = {} by auto
have minimal-unsat-core-tabl-atoms (set I) t (set as)
  by (rule minimal-unsat-core-tabl-atoms-mono[OF - unsat], auto)
from preprocess-minimal-unsat-core[OF prep this empty]
have minimal-unsat-core-ns (set I) (set nsc) .
then show ?case using I weak unfolding S' using pre by auto
next
case (? nsc)
obtain t as tv' ui' where prep[simp]: preprocess nsc = (t, as, tv', ui') by (cases
  preprocess nsc)
show ?case unfolding init-nsc-def
  using init-s preprocess-tableau-normalized[OF prep]
  by (auto simp: list-map-to-fun-create-map)
next
case (? nsc J S v)
obtain asi s tv ui where S: S = ((asi, tv, ui), s) by (cases S, auto)
obtain t as tv' ui' where prep[simp]: preprocess nsc = (t, as, tv', ui') by (cases
  preprocess nsc)
have (J, (solution-s s)) ⊨ias set as (solution-s s) ⊨t t
  using 6 S solution-s[of t - s] by auto
from i-preprocess-sat[OF prep - this]
show ?case using 6 S by auto

```

```

next
  case ( $\gamma$  nsc J S c K S' S'')
    obtain t as tvp uip where prep[simp]: preprocess nsc = (t, as, tvp, uip) by (cases
      preprocess nsc)
      obtain asi s tv ui where S: S = ((asi, tv, ui), s) by (cases S, auto)
      obtain asi' s' tv' ui' where S': S' = ((asi', tv', ui'), s') by (cases S', auto)
      obtain asi'' s'' tv'' ui'' where S'': S'' = ((asi'', tv'', ui''), s'') by (cases S'', auto)
      from backtrack-s[of t - s c - s' s'']
      show ?case using  $\gamma$  S S' S'' by auto
next
  case ( $\delta$  nsc J S)
    then show ?case using weak-invariant-s by (cases S, auto)
next
  case ( $\epsilon$  nsc J S)
    then show ?case using checked-invariant-s by (cases S, auto)
qed

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  $\Rightarrow$  assert-cs j s = Inr s'  $\Rightarrow$ 
    invariant-cs cs (insert j J) s' and
  assert-cs-unsat: invariant-cs cs J s  $\Rightarrow$  assert-cs j s = Unsat I  $\Rightarrow$ 
    set I  $\subseteq$  insert j J  $\wedge$  minimal-unsat-core (set I) cs and
  check-cs-ok: invariant-cs cs J s  $\Rightarrow$  check-cs s = (s', None)  $\Rightarrow$ 
    checked-cs cs J s' and
  check-cs-unsat: invariant-cs cs J s  $\Rightarrow$  check-cs s = (s', Some I)  $\Rightarrow$ 
    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  $\Rightarrow$  solution-cs s = v  $\Rightarrow$  (J, v)  $\models_{ics}$  set cs and
  backtrack-cs: checked-cs cs J s  $\Rightarrow$  checkpoint-cs s = c
     $\Rightarrow$  weak-invariant-cs cs K s'  $\Rightarrow$  backtrack-cs c s' = s''  $\Rightarrow$  J  $\subseteq$  K  $\Rightarrow$ 
  invariant-cs cs J s'' and
  weak-invariant-cs: invariant-cs cs J s  $\Rightarrow$  weak-invariant-cs cs J s and
  checked-invariant-cs: checked-cs cs J s  $\Rightarrow$  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 check-
  point-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  $\Rightarrow$  's and
    assert-nsc :: 'i  $\Rightarrow$  's  $\Rightarrow$  'i list + 's and
    check-nsc :: 's  $\Rightarrow$  's  $\times$  'i list option and
    solution-nsc :: 's  $\Rightarrow$  (var, 'a) mapping and
    checkpoint-nsc :: 's  $\Rightarrow$  'c and
    backtrack-nsc :: 'c  $\Rightarrow$  's  $\Rightarrow$  's and
    weak-invariant-nsc :: ('i,'a) i-ns-constraint list  $\Rightarrow$  'i set  $\Rightarrow$  's  $\Rightarrow$  bool and
    invariant-nsc :: ('i,'a) i-ns-constraint list  $\Rightarrow$  'i set  $\Rightarrow$  's  $\Rightarrow$  bool and
    checked-nsc :: ('i,'a) i-ns-constraint list  $\Rightarrow$  'i set  $\Rightarrow$  's  $\Rightarrow$  bool and
    to-ns :: 'i i-constraint list  $\Rightarrow$  ('i,'a) i-ns-constraint list and
    from-ns :: (var, 'a) mapping  $\Rightarrow$  'a ns-constraint list  $\Rightarrow$  (var, rat) mapping
  begin

    fun assert-cs where assert-cs i (cs,s) = (case assert-nsc i s of
      Unsat I  $\Rightarrow$  Unsat I
      | Inr s'  $\Rightarrow$  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 ( $\lambda$  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)  $\wedge$  weak-invariant-nsc
        (to-ns cs) J s)
    fun invariant-cs where
      invariant-cs cs J (ds,s) = (ds = map snd (to-ns cs)  $\wedge$  invariant-nsc (to-ns cs) J
        s)
    fun checked-cs where
      checked-cs cs J (ds,s) = (ds = map snd (to-ns cs)  $\wedge$  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 (unfold-locales, goal-cases)
  case (1 cs J S j S')
    then obtain s where S: S = (map snd (to-ns cs),s) by (cases S, auto)
    note pre = 1[unfolded S assert-cs.simps]
    from pre(2) obtain s' where
      ok: assert-nsc j s = Inr s' and S': S' = (map snd (to-ns cs),s')
      by (auto split: sum.splits)
    from pre(1)
    have inv: invariant-nsc (to-ns cs) J s by simp
    from assert-nsc-ok[OF inv ok]
    show ?case unfolding invariant-cs.simps S' split by auto
  next
    case (2 cs J S j I)
    then obtain s where S: S = (map snd (to-ns cs), s) by (cases S, auto)
    note pre = 2[unfolded S assert-cs.simps]
    from pre(2) have unsat: assert-nsc j s = Unsat I
    by (auto split: sum.splits)
    from pre(1) have inv: invariant-nsc (to-ns cs) J s by auto
    from assert-nsc-unsat[OF inv unsat]
    have set I ⊆ insert j J minimal-unsat-core-ns (set I) (set (to-ns cs))
    by auto
    from to-ns-unsat[OF this(2)] this(1)
    show ?case by blast
  next
    case (3 cs J S S')
    then show ?case using check-nsc-ok unfolding check-cs-def
    by (cases S, auto split: prod.splits)
  next
    case (4 cs J S S' I)
    then obtain s where S: S = (map snd (to-ns cs),s) by (cases S, auto)
    note pre = 4[unfolded S check-cs-def]
    from pre(2) obtain s' where unsat: check-nsc s = (s',Some I)
    and S': S' = (map snd (to-ns cs),s')
    by (auto split: prod.splits)
    from pre(1) have inv: invariant-nsc (to-ns cs) J s by auto
    from check-nsc-unsat[OF inv unsat]
    have set I ⊆ J weak-invariant-nsc (to-ns cs) J s'
    minimal-unsat-core-ns (set I) (set (to-ns cs))
    unfolding minimal-unsat-core-ns-def by auto
    from to-ns-unsat[OF this(3)] this(1,2)
    show ?case unfolding S' using S by auto
  next
    case (5 cs)
    show ?case unfolding init-cs-def Let-def using init-nsc by auto
  next
    case (6 cs J S v)
    then obtain s where S: S = (map snd (to-ns cs),s) by (cases S, auto)
    obtain w where w: solution-nsc s = w by auto

```

```

note pre = 6[unfolded S solution-cs.simps w Let-def]
from pre have
  inv: checked-nsc (to-ns cs) J s and
    v: v = ⟨from-ns w (map snd (to-ns cs))⟩ by auto
  from solution-nsc[OF inv w] have w: (J, ⟨w⟩) ⊨inss set (to-ns cs) .
  from i-to-ns-sat[OF w]
  show ?case unfolding v .
next
  case (7 cs J S c K S' S'')
  then show ?case using backtrack-nsc[of to-ns cs J]
    by (cases S, cases S', cases S'', auto)
next
  case (8 cs J S)
  then show ?case using weak-invariant-nsc by (cases S, auto)
next
  case (9 cs J S)
  then show ?case using checked-invariant-nsc by (cases S, auto)
qed

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

..

lemma Incremental-State-Ops-Simplex-Default-assert-all-s[simp]:

Incremental-State-Ops-Simplex-Default.assert-all-s = assert-all-s

by (metis assert-all-s-def assert-s-def)

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*

..

type-synonym '*i simplex-state*' = *QDelta ns-constraint list*

$\times (('i, ('i \times QDelta atom) list) mapping \times ((var, QDelta) mapping \Rightarrow (var, QDelta) mapping))$

$\times 'i list)$

$\times ('i, QDelta) state$

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 –

interpret *Incremental-NS-Constraint-Ops* *init-nsc* *assert-nsc* *check-nsc* *solution-nsc* *checkpoint-nsc*

backtrack-nsc *weak-invariant-nsc* *invariant-nsc* *checked-nsc*

using *Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.ns-constraints-ops*

```

show 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
 $\dots$ 
qed

```

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 internal 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)  $\Rightarrow$  ((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  $\in$  set ui then Inl [i] else
   case assert-all-s (list-map-to-fun asi i) s of
     Inl y  $\Rightarrow$  Inl y | Inr s'  $\Rightarrow$  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)  $\Rightarrow$ 
  (Simplex-State (cs, asi-tv, s'), res))

fun solution-simplex where
  solution-simplex (Simplex-State (cs, (asi, tv, ui), s)) = ⟨from-ns (tv (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)
by (simp add: Let-def Incremental-Simplex.init-cs-def Incremental-Atom-Ops-For-NS-Constraint-Ops-Default)

```

```

lemma assert-simplex': assert-simplex i (Simplex-State s) = map-sum id Simplex-State (assert-simplex' i s)
  by (cases s, cases fst (snd s), auto
    simp add: Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.assert-nsc-def
    split: sum.splits)

lemma check-simplex': check-simplex (Simplex-State s) = map-prod Simplex-State id (check-simplex' s)
  by (cases s, simp add: Incremental-Simplex.check-cs-def
    Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.check-nsc-def split: prod.splits)

lemma solution-simplex': solution-simplex (Simplex-State s) = solution-simplex' s
  by (cases s, auto)

lemma checkpoint-simplex': checkpoint-simplex (Simplex-State s) = Simplex-Checkpoint (checkpoint-simplex' s)
  by (cases s, auto split: sum.splits)

lemma backtrack-simplex': backtrack-simplex (Simplex-Checkpoint c) (Simplex-State s) = Simplex-State (backtrack-simplex' c s)
  by (cases s, auto split: sum.splits)

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)
  using Incremental-Simplex.init-cs by (simp add: init-simplex')

lemma assert-simplex-ok:
  invariant-simplex cs J s  $\implies$  assert-simplex j s = Inr s'  $\implies$  invariant-simplex cs (insert j J) s'
  proof (cases s)

```

```

case s: (Simplex-State ss)
  show invariant-simplex cs J s  $\Rightarrow$  assert-simplex j s = Inr s'  $\Rightarrow$  invariant-simplex cs (insert j J) s'
    unfolding s invariant-simplex.simps assert-simplex' using Incremental-Simplex.assert-cs-ok[of cs J ss j]
      by (cases assert-simplex' j ss, auto)
qed

lemma assert-simplex-unsat:
  invariant-simplex cs J s  $\Rightarrow$  assert-simplex j s = Inl I  $\Rightarrow$ 
    set I  $\subseteq$  insert j J  $\wedge$  minimal-unsat-core (set I) cs
  proof (cases s)
    case s: (Simplex-State ss)
      show invariant-simplex cs J s  $\Rightarrow$  assert-simplex j s = Inl I  $\Rightarrow$ 
        set I  $\subseteq$  insert j J  $\wedge$  minimal-unsat-core (set I) cs
        unfolding s invariant-simplex.simps assert-simplex'
          using Incremental-Simplex.assert-cs-unsat[of cs J ss j]
            by (cases assert-simplex' j ss, auto)
  qed

lemma check-simplex-ok:
  invariant-simplex cs J s  $\Rightarrow$  check-simplex s = (s',None)  $\Rightarrow$  checked-simplex cs J s'
  proof (cases s)
    case s: (Simplex-State ss)
      show invariant-simplex cs J s  $\Rightarrow$  check-simplex s = (s',None)  $\Rightarrow$  checked-simplex cs J s'
        unfolding s invariant-simplex.simps check-simplex.simps check-simplex' using
          Incremental-Simplex.check-cs-ok[of cs J ss]
        by (cases check-simplex' ss, auto)
  qed

lemma check-simplex-unsat:
  invariant-simplex cs J s  $\Rightarrow$  check-simplex s = (s',Some I)  $\Rightarrow$ 
    weak-invariant-simplex cs J s'  $\wedge$  set I  $\subseteq$  J  $\wedge$  minimal-unsat-core (set I) cs
  proof (cases s)
    case s: (Simplex-State ss)
      show invariant-simplex cs J s  $\Rightarrow$  check-simplex s = (s',Some I)  $\Rightarrow$ 
        weak-invariant-simplex cs J s'  $\wedge$  set I  $\subseteq$  J  $\wedge$  minimal-unsat-core (set I) cs
        unfolding s invariant-simplex.simps check-simplex.simps check-simplex'
          using Incremental-Simplex.check-cs-unsat[of cs J ss - I]
        by (cases check-simplex' ss, auto)
  qed

lemma solution-simplex:
  checked-simplex cs J s  $\Rightarrow$  solution-simplex s = v  $\Rightarrow$  (J, v)  $\models_{ics}$  set cs
  using Incremental-Simplex.solution-cs[of cs J]
  by (cases s, auto simp: solution-simplex')

```

```

lemma backtrack-simplex:
  checked-simplex cs J s ==>
  checkpoint-simplex s = c ==>
  weak-invariant-simplex cs K s' ==>
  backtrack-simplex c s' = s'' ==>
  J ⊆ K ==>
  invariant-simplex cs J s''

proof -
  obtain ss where ss: s = Simplex-State ss by (cases s, auto)
  obtain ss' where ss': s' = Simplex-State ss' by (cases s', auto)
  obtain ss'' where ss'': s'' = Simplex-State ss'' by (cases s'', auto)
  obtain cc where cc: c = Simplex-Checkpoint cc by (cases c, auto)
  show checked-simplex cs J s ==>
    checkpoint-simplex s = c ==>
    weak-invariant-simplex cs K s' ==>
    backtrack-simplex c s' = s'' ==>
    J ⊆ K ==>
    invariant-simplex cs J s''

  unfolding ss ss' ss'' cc checked-simplex.simps invariant-simplex.simps check-
  point-simplex' backtrack-simplex'
  using Incremental-Simplex.backtrack-cs[of cs J ss cc K ss' ss''] by simp
  qed

```

```

lemma weak-invariant-simplex:
  invariant-simplex cs J s ==> weak-invariant-simplex cs J s
  using Incremental-Simplex.weak-invariant-cs[of cs J] by (cases s, auto)

```

```

lemma checked-invariant-simplex:
  checked-simplex cs J s ==> invariant-simplex cs J s
  using Incremental-Simplex.checked-invariant-cs[of cs J] by (cases s, auto)

```

```

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 (induct K arbitrary: s J)
  case (Cons k K s J)
  from Cons(3) obtain s'' where ass: assert-simplex k s = Inr s'' and rec:

```

```

assert-all-simplex K s'' = Inr s'
  by (auto split: sum.splits)
from Cons(1)[OF assert-simplex-ok[OF Cons(2) ass] rec]
  show ?case by auto
qed auto

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 (induct K arbitrary: s J)
  case (Cons k K s J)
  show ?case
  proof (cases assert-simplex k s)
    case unsat: (Inl J')
    with Cons have J': J' = I by auto
    from assert-simplex-unsat[OF Cons(2) unsat]
    have set J' ⊆ insert k J minimal-unsat-core (set J') cs by auto
    then show ?thesis unfolding J' i-satisfies-cs.simps
      by auto
  next
  case (Inr s')
    from Cons(1)[OF assert-simplex-ok[OF Cons(2) Inr]] Cons(3) Inr show
    ?thesis by auto
  qed
qed simp

```

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 ( $\lambda x.$  ("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.