

# An Incremental Simplex Algorithm with Unsatisfiable Core Generation\*

Filip Marić      Mirko Spasić      René Thiemann

February 23, 2021

## 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Auxiliary Results</b>	<b>3</b>
<b>3</b>	<b>Linearly Ordered Rational Vectors</b>	<b>6</b>
<b>4</b>	<b>Linear Polynomials and Constraints</b>	<b>9</b>
<b>5</b>	<b>Rational Numbers Extended with Infinitesimal Element</b>	<b>19</b>
<b>6</b>	<b>The Simplex Algorithm</b>	<b>22</b>
6.1	Procedure Specification . . . . .	24
6.2	Handling Strict Inequalities . . . . .	25
6.3	Preprocessing . . . . .	27
6.4	Incrementally Asserting Atoms . . . . .	32
6.5	Asserting Single Atoms . . . . .	45

---

\*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 . . . . .	51
6.7	Check implementation . . . . .	63
6.8	Symmetries . . . . .	74
6.9	Concrete implementation . . . . .	75
<b>7</b>	<b>The Incremental Simplex Algorithm</b>	<b>87</b>
7.1	Lowest Layer: Fixed Tableau and Incremental Atoms . . . . .	87
7.2	Intermediate Layer: Incremental Non-Strict Constraints . . . . .	89
7.3	Highest Layer: Incremental Constraints . . . . .	92
7.4	Concrete Implementation . . . . .	93
7.4.1	Connecting all the locales . . . . .	93
7.4.2	An implementation which encapsulates the state . . . . .	96
7.4.3	Soundness of the incremental simplex implementation . . . . .	96
7.5	Test Executability and Example for Incremental Interface . . . . .	99

## 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.Permutation
  HOL-Library.Mapping
begin
```

**lemma** *map-reindex*:

```
assumes  $\forall i < \text{length } l. g (l ! i) = f i$ 
shows  $\text{map } f [0..<\text{length } l] = \text{map } g l$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *map-parametrize-idx*:

```
 $\text{map } f l = \text{map } (\lambda i. f (l ! i)) [0..<\text{length } l]$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *last-tl*:

```
assumes  $\text{length } l > 1$ 
shows  $\text{last } (\text{tl } l) = \text{last } l$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *hd-tl*:

```
assumes  $\text{length } l > 1$ 
shows  $\text{hd } (\text{tl } l) = l ! 1$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *butlast-empty-conv-length*:

```
shows  $(\text{butlast } l = []) = (\text{length } l \leq 1)$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *butlast-nth*:

```
assumes  $n + 1 < \text{length } l$ 
shows  $\text{butlast } l ! n = l ! n$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *last-take-conv-nth*:

```
assumes  $0 < n \wedge n \leq \text{length } l$ 
shows  $\text{last } (\text{take } n l) = l ! (n - 1)$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *tl-nth*:

```
assumes  $l \neq []$ 
shows  $\text{tl } l ! n = l ! (n + 1)$ 
 $\langle \text{proof} \rangle$ 
```

**lemma** *interval-3split*:

**assumes**  $i < n$

**shows**  $[0..<n] = [0..<i] @ [i] @ [i+1..<n]$

*<proof>*

**abbreviation**  $list\text{-}min\ l \equiv foldl\ min\ (hd\ l)\ (tl\ l)$

**lemma** *list-min-Min[simp]*:  $l \neq [] \implies list\text{-}min\ l = Min\ (set\ l)$

*<proof>*

**definition** *min-satisfying* ::  $((a::linorder) \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ option$  **where**

$min\text{-}satisfying\ P\ l \equiv$

$let\ xs = filter\ P\ l\ in$

$if\ xs = []\ then\ None\ else\ Some\ (list\text{-}min\ xs)$

**lemma** *min-satisfying-None*:

$min\text{-}satisfying\ P\ l = None \longrightarrow$

$(\forall x \in set\ l. \neg P\ x)$

*<proof>*

**lemma** *min-satisfying-Some*:

$min\text{-}satisfying\ P\ l = Some\ x \longrightarrow$

$x \in set\ l \wedge P\ x \wedge (\forall x' \in set\ l. x' < x \longrightarrow \neg P\ x')$

*<proof>*

**lemma** *min-element*:

**fixes**  $k :: nat$

**assumes**  $\exists (m::nat). P\ m$

**shows**  $\exists mm. P\ mm \wedge (\forall m'. m' < mm \longrightarrow \neg P\ m')$

*<proof>*

**lemma** *finite-fun-args*:

**assumes**  $finite\ A\ \forall a \in A. finite\ (B\ a)$

**shows**  $finite\ \{f. (\forall a. if\ a \in A\ then\ f\ a \in B\ a\ else\ f\ a = f0\ a)\}$  (**is**  $finite\ (?F\ A)$ )

*<proof>*

**lemma** *foldl-mapping-update*:  
**assumes**  $X \in \text{set } l \text{ distinct } (\text{map } f \ l)$   
**shows**  $\text{Mapping.lookup } (\text{foldl } (\lambda m \ a. \ \text{Mapping.update } (f \ a) \ (g \ a) \ m) \ i \ l) \ (f \ X) = \text{Some } (g \ X)$   
 $\langle \text{proof} \rangle$

**end**

**theory** *Rel-Chain*  
**imports**  
*Simplex-Auxiliary*  
**begin**

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

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

**lemma** *rel-chain-drop*:  
 $\text{rel-chain } l \ R \ ==> \ \text{rel-chain } (\text{drop } n \ l) \ R$   
 $\langle \text{proof} \rangle$

**lemma** *rel-chain-take*:  
 $\text{rel-chain } l \ R \ ==> \ \text{rel-chain } (\text{take } n \ l) \ R$   
 $\langle \text{proof} \rangle$

**lemma** *rel-chain-butlast*:  
 $\text{rel-chain } l \ R \ ==> \ \text{rel-chain } (\text{butlast } l) \ R$   
 $\langle \text{proof} \rangle$

**lemma** *rel-chain-tl*:  
 $\text{rel-chain } l \ R \ ==> \ \text{rel-chain } (\text{tl } l) \ R$   
 $\langle \text{proof} \rangle$

**lemma** *rel-chain-append*:  
**assumes**  $\text{rel-chain } l \ R \ \text{rel-chain } l' \ R \ (\text{last } l, \ \text{hd } l') \in R$   
**shows**  $\text{rel-chain } (l \ @ \ l') \ R$   
 $\langle \text{proof} \rangle$

**lemma** *rel-chain-appendD*:  
**assumes**  $\text{rel-chain } (l \ @ \ l') \ R$   
**shows**  $\text{rel-chain } l \ R \ \text{rel-chain } l' \ R \ l \neq [] \wedge l' \neq [] \longrightarrow (\text{last } l, \ \text{hd } l') \in R$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancl-rel-chain*:

$(x, y) \in R^* \iff (\exists l. l \neq [] \wedge \text{hd } l = x \wedge \text{last } l = y \wedge \text{rel-chain } l R)$   
(**is** ?lhs = ?rhs)  
<proof>

**lemma** *trancl-rel-chain*:

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

**lemma** *rel-chain-elems-rtrancl*:

**assumes** *rel-chain*  $l R$   $i \leq j < \text{length } l$   
**shows**  $(l ! i, l ! j) \in R^*$   
<proof>

**lemma** *reorder-cyclic-list*:

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

**end**

### 3 Linearly Ordered Rational Vectors

**theory** *Simplex-Algebra*

**imports**

*HOL.Rat*

*HOL.Real-Vector-Spaces*

**begin**

**class** *scaleRat* =

**fixes** *scaleRat* ::  $\text{rat} \Rightarrow 'a \Rightarrow 'a$  (**infixr** \*R 75)

**begin**

**abbreviation**

*divideRat* ::  $'a \Rightarrow \text{rat} \Rightarrow 'a$  (**infixl** '/R 70)

**where**

$x /R r == \text{scaleRat } (\text{inverse } r) x$

**end**

**class** *rational-vector* = *scaleRat* + *ab-group-add* +

**assumes** *scaleRat-right-distrib*:  $\text{scaleRat } a (x + y) = \text{scaleRat } a x + \text{scaleRat } a y$

**and** *scaleRat-left-distrib*:  $\text{scaleRat } (a + b) x = \text{scaleRat } a x + \text{scaleRat } b x$

**and** *scaleRat-scaleRat*:  $\text{scaleRat } a (\text{scaleRat } b x) = \text{scaleRat } (a * b) x$

**and** *scaleRat-one*:  $\text{scaleRat } 1 \ x = x$

**interpretation** *rational-vector*:

*vector-space* *scaleRat* ::  $\text{rat} \Rightarrow 'a \Rightarrow 'a::\text{rational-vector}$   
(*proof*)

**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*:  $\llbracket a \leq b; k > 0 \rrbracket \implies k *R a \leq k *R b$   
(*proof*)

**lemma** *scaleRat-leq2*:  $\llbracket a \leq b; k < 0 \rrbracket \implies k *R a \geq k *R b$   
(*proof*)

**lemma** *zero-scaleRat*

[*simp*]:  $0 *R v = \text{zero}$   
(*proof*)

**lemma** *scaleRat-zero*

[*simp*]:  $a *R (0::'a) = 0$   
(*proof*)

**lemma** *scaleRat-uminus* [*simp*]:

$-1 *R x = - (x :: 'a)$   
(*proof*)

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

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

**lemma** *minus-leq*:

$(a::'a) \leq b \iff a - b \leq 0$   
(*proof*)

**lemma** *minus-geq*:  $(a::'a) \leq b \iff 0 \leq b - a$

(*proof*)

**lemma** *divide-lt*:

$\llbracket c *R (a::'a) < b; (c::\text{rat}) > 0 \rrbracket \implies a < (1/c) *R b$   
(*proof*)

**lemma** *divide-gt*:

$\llbracket c *R (a::'a) > b; (c::rat) > 0 \rrbracket \implies a > (1/c) *R b$   
*<proof>*

**lemma** *divide-leq*:

$\llbracket c *R (a::'a) \leq b; (c::rat) > 0 \rrbracket \implies a \leq (1/c) *R b$   
*<proof>*

**lemma** *divide-geq*:

$\llbracket c *R (a::'a) \geq b; (c::rat) > 0 \rrbracket \implies a \geq (1/c) *R b$   
*<proof>*

**lemma** *divide-lt1*:

$\llbracket c *R (a::'a) < b; (c::rat) < 0 \rrbracket \implies a > (1/c) *R b$   
*<proof>*

**lemma** *divide-gt1*:

$\llbracket c *R (a::'a) > b; (c::rat) < 0 \rrbracket \implies a < (1/c) *R b$   
*<proof>*

**lemma** *divide-leq1*:

$\llbracket c *R (a::'a) \leq b; (c::rat) < 0 \rrbracket \implies a \geq (1/c) *R b$   
*<proof>*

**lemma** *divide-geq1*:

$\llbracket c *R (a::'a) \geq b; (c::rat) < 0 \rrbracket \implies a \leq (1/c) *R b$   
*<proof>*

**end**

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

**subclass** (**in** *linordered-rational-vector*) *ordered-ab-semigroup-add*  
*<proof>*

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

**begin**

**definition** *scaleRat-rat* :: *rat*  $\Rightarrow$  *rat*  $\Rightarrow$  *rat* **where**

*[simp]*:  $x *R y = x * y$

**instance** *<proof>*

**end**

**instantiation** *rat* :: *ordered-rational-vector*

**begin**

**instance** *<proof>*

**end**

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



```

begin
instance <proof>
end

instantiation rat :: lrv
begin
instance <proof>
end

lemma uminus-less-lrv[simp]: fixes a b :: 'a :: lrv
  shows  $- a < - b \longleftrightarrow b < a$ 
  <proof>

end

```

## 4 Linear Polynomials and Constraints

```

theory Abstract-Linear-Poly
  imports
    Simplex-Algebra
begin

type-synonym var = nat

  (Infinite) linear polynomials as functions from vars to coeffs

definition fun-zero :: var  $\Rightarrow$  '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-valuate :: (var  $\Rightarrow$  rat)  $\Rightarrow$  'a valuation  $\Rightarrow$  ('a::rational-vector) where
  [simp]: fun-valuate 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 <proof>

```

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

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

linear-poly type – rat coeffs

**typedef** *linear-poly* = {  $c :: \text{var} \Rightarrow \text{rat}$ .  $\text{inv } c$  }  
 $\langle \text{proof} \rangle$

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  $\sum x \mid p \ x \neq (0::'b). p \ x * v \ x$ . Executable representation of polynomials uses RBT mappings instead of functions.

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

Vector space operations on polynomials

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

**lift-definition** *zero-linear-poly* :: *linear-poly* **is** *fun-zero*  $\langle \text{proof} \rangle$

**lift-definition** *plus-linear-poly* :: *linear-poly*  $\Rightarrow$  *linear-poly*  $\Rightarrow$  *linear-poly* **is** *fun-plus*  
 $\langle \text{proof} \rangle$

**lift-definition** *scaleRat-linear-poly* :: *rat*  $\Rightarrow$  *linear-poly*  $\Rightarrow$  *linear-poly* **is** *fun-scale*  
 $\langle \text{proof} \rangle$

**definition** *uminus-linear-poly* :: *linear-poly*  $\Rightarrow$  *linear-poly* **where**  
*uminus-linear-poly*  $lp = -1 *R \ lp$

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

**instance**  
 $\langle \text{proof} \rangle$

**end**

Coefficient

**lift-definition** *coeff* :: *linear-poly*  $\Rightarrow$  *var*  $\Rightarrow$  *rat* **is** *fun-coeff*  $\langle$ *proof* $\rangle$

**lemma** *coeff-plus* [*simp*] : *coeff* (*lp1* + *lp2*) *var* = *coeff* *lp1* *var* + *coeff* *lp2* *var*  
 $\langle$ *proof* $\rangle$

**lemma** *coeff-scaleRat* [*simp*]: *coeff* (*k* \**R* *lp1*) *var* = *k* \* *coeff* *lp1* *var*  
 $\langle$ *proof* $\rangle$

**lemma** *coeff-uminus* [*simp*]: *coeff* (-*lp*) *var* = - *coeff* *lp* *var*  
 $\langle$ *proof* $\rangle$

**lemma** *coeff-minus* [*simp*]: *coeff* (*lp1* - *lp2*) *var* = *coeff* *lp1* *var* - *coeff* *lp2* *var*  
 $\langle$ *proof* $\rangle$

Set of variables

**lift-definition** *vars* :: *linear-poly*  $\Rightarrow$  *var set* **is** *fun-vars*  $\langle$ *proof* $\rangle$

**lemma** *coeff-zero*: *coeff* *p* *x*  $\neq$  0  $\longleftrightarrow$  *x*  $\in$  *vars* *p*  
 $\langle$ *proof* $\rangle$

**lemma** *finite-vars*: *finite* (*vars* *p*)  
 $\langle$ *proof* $\rangle$

List of variables

**lift-definition** *vars-list* :: *linear-poly*  $\Rightarrow$  *var list* **is** *fun-vars-list*  $\langle$ *proof* $\rangle$

**lemma** *set-vars-list*: *set* (*vars-list* *lp*) = *vars* *lp*  
 $\langle$ *proof* $\rangle$

Construct single variable polynomial

**lift-definition** *Var* :: *var*  $\Rightarrow$  *linear-poly* **is** *fun-var*  $\langle$ *proof* $\rangle$

Value of a polynomial in a given valuation

**lift-definition** *valuate* :: *linear-poly*  $\Rightarrow$  '*a* *valuation*  $\Rightarrow$  ('*a*::*rational-vector*) **is**  
*fun-valuate*  $\langle$ *proof* $\rangle$

**syntax**

-*valuate* :: *linear-poly*  $\Rightarrow$  '*a* *valuation*  $\Rightarrow$  '*a* (-  $\{\}$  -  $\}$ )

**translations**

*p* $\{\}$ *v* $\}$  == *CONST* *valuate* *p* *v*

**lemma** *valuate-zero*: (0  $\{\}$ *v* $\}$ ) = 0  
 $\langle$ *proof* $\rangle$

**lemma**

*valuate-diff*: (*p*  $\{\}$ *v1* $\}$ ) - (*p*  $\{\}$ *v2* $\}$ ) = (*p*  $\{\}$   $\lambda$  *x*. *v1* *x* - *v2* *x*  $\}$ )  
 $\langle$ *proof* $\rangle$

**lemma** *valuate-opposite-val*:

**shows**  $p \llbracket \lambda x. - v x \rrbracket = - (p \llbracket v \rrbracket)$   
*<proof>*

**lemma** *valuate-nonneg*:

**fixes**  $v :: 'a::\text{linordered-rational-vector valuation}$

**assumes**  $\forall x \in \text{vars } p. (\text{coeff } p x > 0 \longrightarrow v x \geq 0) \wedge (\text{coeff } p x < 0 \longrightarrow v x \leq 0)$

**shows**  $p \llbracket v \rrbracket \geq 0$   
*<proof>*

**lemma** *valuate-nonpos*:

**fixes**  $v :: 'a::\text{linordered-rational-vector valuation}$

**assumes**  $\forall x \in \text{vars } p. (\text{coeff } p x > 0 \longrightarrow v x \leq 0) \wedge (\text{coeff } p x < 0 \longrightarrow v x \geq 0)$

**shows**  $p \llbracket v \rrbracket \leq 0$   
*<proof>*

**lemma** *valuate-uminus*:  $(-p) \llbracket v \rrbracket = - (p \llbracket v \rrbracket)$

*<proof>*

**lemma** *valuate-add-lemma*:

**fixes**  $v :: 'a \Rightarrow 'b::\text{rational-vector}$

**assumes**  $\text{finite } \{v. f1 v \neq 0\} \text{ finite } \{v. f2 v \neq 0\}$

**shows**

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

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

*<proof>*

**lemma** *valuate-add*:  $(p1 + p2) \llbracket v \rrbracket = (p1 \llbracket v \rrbracket) + (p2 \llbracket v \rrbracket)$

*<proof>*

**lemma** *valuate-minus*:  $(p1 - p2) \llbracket v \rrbracket = (p1 \llbracket v \rrbracket) - (p2 \llbracket v \rrbracket)$

*<proof>*

**lemma** *valuate-scaleRat*:

$(c *R lp) \llbracket v \rrbracket = c *R (lp \llbracket v \rrbracket)$

*<proof>*

**lemma** *valuate-Var*:  $(\text{Var } x) \llbracket v \rrbracket = v x$

*<proof>*

**lemma** *valuate-sum*:  $((\sum x \in A. f x) \llbracket v \rrbracket) = (\sum x \in A. ((f x) \llbracket v \rrbracket))$

*<proof>*

**lemma** *distinct-vars-list*:

*distinct (vars-list p)*

*<proof>*

**lemma** *zero-coeff-zero*:  $p = 0 \iff (\forall v. \text{coeff } p \ v = 0)$   
*<proof>*

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

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

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

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

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

**lemma** *vars-uminus [simp]*:  $\text{vars } (-p) = \text{vars } p$   
*<proof>*

**lemma** *vars-plus [simp]*:  $\text{vars } (p1 + p2) \subseteq \text{vars } p1 \cup \text{vars } p2$   
*<proof>*

**lemma** *vars-minus [simp]*:  $\text{vars } (p1 - p2) \subseteq \text{vars } p1 \cup \text{vars } p2$   
*<proof>*

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

**lemma** *vars-scaleRat1*:  $\text{vars } (c *R \ p) \subseteq \text{vars } p$   
*<proof>*

**lemma** *vars-scaleRat*:  $c \neq 0 \implies \text{vars}(c *R \ p) = \text{vars } p$   
*<proof>*

**lemma** *vars-Var [simp]*:  $\text{vars } (\text{Var } x) = \{x\}$   
*<proof>*

**lemma** *coeff-Var1 [simp]*:  $\text{coeff } (\text{Var } x) \ x = 1$   
*<proof>*

**lemma** *coeff-Var2*:  $x \neq y \implies \text{coeff } (\text{Var } x) \ y = 0$

*<proof>*

**lemma** *valuate-depend*:

**assumes**  $\forall x \in \text{vars } p. v \ x = v' \ x$

**shows**  $(p \ \{\!|v|\!\}) = (p \ \{\!|v'|\!\})$

*<proof>*

**lemma** *valuate-update-x-lemma*:

**fixes**  $v1 \ v2 :: 'a::\text{rational-vector valuation}$

**assumes**

$\forall y. f \ y \neq 0 \longrightarrow y \neq x \longrightarrow v1 \ y = v2 \ y$

$\text{finite } \{v. f \ v \neq 0\}$

**shows**

$(\sum x \in \{v. f \ v \neq 0\}. f \ x *R \ v1 \ x) + f \ x *R \ (v2 \ x - v1 \ x) = (\sum x \in \{v. f \ v \neq 0\}. f \ x *R \ v2 \ x)$

*<proof>*

**lemma** *valuate-update-x*:

**fixes**  $v1 \ v2 :: 'a::\text{rational-vector valuation}$

**assumes**  $\forall y \in \text{vars } lp. y \neq x \longrightarrow v1 \ y = v2 \ y$

**shows**  $lp \ \{\!|v1|\!\} + \text{coeff } lp \ x *R \ (v2 \ x - v1 \ x) = (lp \ \{\!|v2|\!\})$

*<proof>*

**lemma** *vars-zero*:  $\text{vars } 0 = \{\}$

*<proof>*

**lemma** *vars-empty-zero*:  $\text{vars } lp = \{\} \longleftrightarrow lp = 0$

*<proof>*

**definition** *max-var*::  $\text{linear-poly} \Rightarrow \text{var}$  **where**

$\text{max-var } lp \equiv \text{if } lp = 0 \text{ then } 0 \text{ else } \text{Max } (\text{vars } lp)$

**lemma** *max-var-max*:

**assumes**  $a \in \text{vars } lp$

**shows**  $\text{max-var } lp \geq a$

*<proof>*

**lemma** *max-var-code*[code]:

$\text{max-var } lp = (\text{let } vl = \text{vars-list } lp$

$\text{in if } vl = [] \text{ then } 0 \text{ else } \text{foldl } \text{max } (\text{hd } vl) \ (\text{tl } vl))$

*<proof>*

**definition** *monom-var*::  $\text{linear-poly} \Rightarrow \text{var}$  **where**

$\text{monom-var } l = \text{max-var } l$

**definition** *monom-coeff*::  $\text{linear-poly} \Rightarrow \text{rat}$  **where**

$\text{monom-coeff } l = \text{coeff } l \ (\text{monom-var } l)$

**definition** *is-monom* ::  $\text{linear-poly} \Rightarrow \text{bool}$  **where**

$is\text{-monom } l \longleftrightarrow length (vars\text{-list } l) = 1$

**lemma** *is-monom-vars-not-empty*:

$is\text{-monom } l \implies vars\ l \neq \{\}$

*<proof>*

**lemma** *monom-var-in-vars*:

$is\text{-monom } l \implies monom\text{-var } l \in vars\ l$

*<proof>*

**lemma** *zero-is-no-monom[simp]*:  $\neg is\text{-monom } 0$

*<proof>*

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

$is\text{-monom } l \implies monom\text{-coeff } l \neq 0$

*<proof>*

**lemma** *list-two-elements*:

$\llbracket y \in set\ l; x \in set\ l; length\ l = Suc\ 0; y \neq x \rrbracket \implies False$

*<proof>*

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

**assumes** *is-monom l*

**shows**  $vars\ l = \{monom\text{-var } l\}$

*<proof>*

**lemma** *monom-valuate*:

**assumes** *is-monom m*

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

*<proof>*

**lemma** *coeff-zero-simp [simp]*:

$coeff\ 0\ v = 0$

*<proof>*

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

*<proof>*

**lemma** *poly-eqI*:

**assumes**  $\bigwedge v. coeff\ p\ v = coeff\ q\ v$

**shows**  $p = q$

*<proof>*

**lemma** *coeff-sum-list*:

**assumes** *distinct xs*

**shows**  $coeff\ (\sum x \leftarrow xs. f\ x *R\ lp\text{-monom } 1\ x)\ v = (if\ v \in set\ xs\ then\ f\ v\ else\ 0)$

*<proof>*

**lemma** *linear-poly-sum*:

$p \{ v \} = (\sum_{x \in \text{vars } p} \text{coeff } p \ x * R \ v \ x)$   
 <proof>

**lemma** *all-valuate-zero*: **assumes**  $\bigwedge(v::'a::\text{lrval valuation}). p \{ v \} = 0$   
**shows**  $p = 0$   
 <proof>

**lemma** *linear-poly-eqI*: **assumes**  $\bigwedge(v::'a::\text{lrval valuation}). (p \{ v \}) = (q \{ v \})$   
**shows**  $p = q$   
 <proof>

**lemma** *monom-poly-assemble*:  
**assumes** *is-monom*  $p$   
**shows**  $\text{monom-coeff } p * R \ \text{lp-monom } 1 (\text{monom-var } p) = p$   
 <proof>

**lemma** *coeff-sum*:  $\text{coeff } (\text{sum } (f :: - \Rightarrow \text{linear-poly}) \text{ is}) \ x = \text{sum } (\lambda \ i. \ \text{coeff } (f \ i) \ x) \ \text{is}$   
 <proof>

**end**

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

**definition** *get-var-coeff* ::  $(\text{var}, \text{rat}) \text{ fmap} \Rightarrow \text{var} \Rightarrow \text{rat}$  **where**  
 $\text{get-var-coeff } \text{lp } v == \text{case } \text{fmlookup } \text{lp } v \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } c \Rightarrow c$

**definition** *set-var-coeff* ::  $\text{var} \Rightarrow \text{rat} \Rightarrow (\text{var}, \text{rat}) \text{ fmap} \Rightarrow (\text{var}, \text{rat}) \text{ fmap}$  **where**  
 $\text{set-var-coeff } v \ c \ \text{lp} ==$   
 $\text{if } c = 0 \text{ then } \text{fmdrop } v \ \text{lp} \text{ else } \text{fmupd } v \ c \ \text{lp}$

**lift-definition** *LinearPoly* ::  $(\text{var}, \text{rat}) \text{ fmap} \Rightarrow \text{linear-poly}$  **is** *get-var-coeff*  
 <proof>

**definition** *ordered-keys* ::  $( 'a :: \text{linorder}, 'b) \text{ fmap} \Rightarrow 'a \text{ list}$  **where**  
 $\text{ordered-keys } m = \text{sorted-list-of-set } (\text{fset } (\text{fmdom } m))$

**context includes** *fmap.lifting lifting-syntax*  
**begin**

**lemma** [*transfer-rule*]:  $(( (= ) ==> (=) ) ==> \text{pcr-linear-poly} ==> (=) (=)$   
*pcr-linear-poly*



*<proof>*

**lemma** [*transfer-rule*]: (*pcr-fmap* (=) (=) ==> *pcr-linear-poly*) ( $\lambda f x. \text{case } f \ x \text{ of } \text{None} \Rightarrow 0 \mid \text{Some } x \Rightarrow x$ ) *LinearPoly*  
*<proof>*

**lift-definition** *linear-poly-map* :: *linear-poly*  $\Rightarrow$  (*var*, *rat*) *fmap* **is**  
 $\lambda lp \ x. \text{if } lp \ x = 0 \text{ then } \text{None} \text{ else } \text{Some } (lp \ x)$  *<proof>*

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

Zero

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

**lemma** [*code abstract*]:  
*linear-poly-map* 0 = *zero* *<proof>*

Addition

**definition** *add-monom* :: *rat*  $\Rightarrow$  *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* ( $\lambda lp \ v. \text{add-monom } (get-var-coeff \ lp1 \ v) \ v \ lp$ ) *lp2* (*ordered-keys* *lp1*)

**lemma** *lookup-add-monom*:  
*get-var-coeff* *lp* *v* + *c*  $\neq 0 \implies$   
*fmlookup* (*add-monom* *c* *v* *lp*) *v* = *Some* (*get-var-coeff* *lp* *v* + *c*)  
*get-var-coeff* *lp* *v* + *c* = 0  $\implies$   
*fmlookup* (*add-monom* *c* *v* *lp*) *v* = *None*  
*x*  $\neq v \implies$  *fmlookup* (*add-monom* *c* *v* *lp*) *x* = *fmlookup* *lp* *x*  
*<proof>*

**lemma** *fmlookup-fold-not-mem*: *x*  $\notin$  *set* *k1*  $\implies$   
*fmlookup* (*foldl* ( $\lambda lp \ v. \text{add-monom } (get-var-coeff \ P1 \ v) \ v \ lp$ ) *P2* *k1*) *x*  
= *fmlookup* *P2* *x*  
*<proof>*

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

Scaling

**definition** *scale* :: *rat*  $\Rightarrow$  (*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>*

**lemma** *coeff-code* [*code*]:  
 $coeff\ lp = get-var-coeff (linear-poly-map\ lp)$   
*<proof>*

**lemma** *Var-code*[*code abstract*]:  
 $linear-poly-map (Var\ x) = set-var-coeff\ x\ 1\ fmempty$   
*<proof>*

**lemma** *vars-code*[*code*]:  $vars\ lp = fset (fndom (linear-poly-map\ lp))$   
*<proof>*

**lemma** *vars-list-code*[*code*]:  $vars-list\ lp = ordered-keys (linear-poly-map\ lp)$   
*<proof>*

**lemma** *valuate-code*[*code*]:  $valuate\ lp\ val = ($   
   $let\ lpm = linear-poly-map\ lp$   
   $in\ sum-list (map (\lambda\ x. (the (fmlookup\ lpm\ x)) *R (val\ x)) (vars-list\ lp)))$   
*<proof>*

**end**

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

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

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

**instance**  
*<proof>*  
**end**

**end**

## 5 Rational Numbers Extended with Infinitesimal Element

```

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

datatype QDelta = QDelta rat rat

primrec qdfst :: QDelta  $\Rightarrow$  rat where
  qdfst (QDelta a b) = a

primrec qdsnd :: QDelta  $\Rightarrow$  rat where
  qdsnd (QDelta a b) = b

lemma [simp]: QDelta (qdfst qd) (qdsnd qd) = qd
   $\langle$ proof $\rangle$ 

lemma [simp]:  $\llbracket$ QDelta.qdsnd x = QDelta.qdsnd y; QDelta.qdfst y = QDelta.qdfst
x $\rrbracket \Longrightarrow$  x = y
   $\langle$ proof $\rangle$ 

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
   $\langle$ proof $\rangle$ 

```

**end**

**instantiation** *QDelta* :: *linorder*

**begin**

**definition** *less-eq-QDelta* :: *QDelta*  $\Rightarrow$  *QDelta*  $\Rightarrow$  *bool*

**where**

$qd1 \leq qd2 \iff (qdfst\ qd1 < qdfst\ qd2) \vee (qdfst\ qd1 = qdfst\ qd2 \wedge qdsnd\ qd1 \leq qdsnd\ qd2)$

**definition** *less-QDelta* :: *QDelta*  $\Rightarrow$  *QDelta*  $\Rightarrow$  *bool*

**where**

$qd1 < qd2 \iff (qdfst\ qd1 < qdfst\ qd2) \vee (qdfst\ qd1 = qdfst\ qd2 \wedge qdsnd\ qd1 < qdsnd\ qd2)$

**instance**  $\langle proof \rangle$

**end**

**instantiation** *QDelta*:: *linordered-rational-vector*

**begin**

**instance**  $\langle proof \rangle$

**end**

**instantiation** *QDelta* :: *lrv*

**begin**

**definition** *one-QDelta* **where**

*one-QDelta* = *QDelta* 1 0

**instance**  $\langle proof \rangle$

**end**

**definition**  $\delta 0$  :: *QDelta*  $\Rightarrow$  *QDelta*  $\Rightarrow$  *rat*

**where**

$\delta 0\ qd1\ qd2 ==$

*let*  $c1 = qdfst\ qd1$ ;  $c2 = qdfst\ qd2$ ;  $k1 = qdsnd\ qd1$ ;  $k2 = qdsnd\ qd2$  *in*

*(if*  $(c1 < c2 \wedge k1 > k2)$  *then*

$(c2 - c1) / (k1 - k2)$

*else*

1

*)*

**definition** *val* :: *QDelta*  $\Rightarrow$  *rat*  $\Rightarrow$  *rat*

**where** *val* *qd*  $\delta = (qdfst\ qd) + \delta * (qdsnd\ qd)$

**lemma** *val-plus*:

*val* (*qd1* + *qd2*)  $\delta = \text{val}\ qd1\ \delta + \text{val}\ qd2\ \delta$

$\langle proof \rangle$

**lemma** *val-scaleRat*:

*val* ( $c *R\ qd$ )  $\delta = c * \text{val}\ qd\ \delta$

$\langle proof \rangle$

**lemma** *qdfst-setsum*:

$finite\ A \implies qdfst\ (\sum\ x \in A.\ f\ x) = (\sum\ x \in A.\ qdfst\ (f\ x))$   
 $\langle proof \rangle$

**lemma** *qdsnd-setsum*:

$finite\ A \implies qdsnd\ (\sum\ x \in A.\ f\ x) = (\sum\ x \in A.\ qdsnd\ (f\ x))$   
 $\langle proof \rangle$

**lemma** *valuate-valuate-rat*:

$lp\ \{\{\lambda v.\ (QDelta\ (vl\ v)\ 0)\}\} = QDelta\ (lp\ \{vl\})\ 0$   
 $\langle proof \rangle$

**lemma** *valuate-rat-valuate*:

$lp\ \{\{\lambda v.\ val\ (vl\ v)\ \delta\}\} = val\ (lp\ \{vl\})\ \delta$   
 $\langle proof \rangle$

**lemma** *delta0*:

**assumes**  $qd1 \leq qd2$   
**shows**  $\forall\ \varepsilon.\ \varepsilon > 0 \wedge \varepsilon \leq (\delta0\ qd1\ qd2) \longrightarrow val\ qd1\ \varepsilon \leq val\ qd2\ \varepsilon$   
 $\langle proof \rangle$

**primrec**

$\delta\text{-min} :: (QDelta \times QDelta)\ list \Rightarrow rat$  **where**  
 $\delta\text{-min}\ [] = 1$  |  
 $\delta\text{-min}\ (h \# t) = min\ (\delta\text{-min}\ t)\ (\delta0\ (fst\ h)\ (snd\ h))$

**lemma** *delta-gt-zero*:

$\delta\text{-min}\ l > 0$   
 $\langle proof \rangle$

**lemma** *delta-le-one*:

$\delta\text{-min}\ l \leq 1$   
 $\langle proof \rangle$

**lemma** *delta-min-append*:

$\delta\text{-min}\ (as\ @\ bs) = min\ (\delta\text{-min}\ as)\ (\delta\text{-min}\ bs)$   
 $\langle proof \rangle$

**lemma** *delta-min-mono*:  $set\ as \subseteq set\ bs \implies \delta\text{-min}\ bs \leq \delta\text{-min}\ as$

$\langle proof \rangle$

**lemma** *delta-min*:

**assumes**  $\forall\ qd1\ qd2.\ (qd1,\ qd2) \in set\ qd \longrightarrow qd1 \leq qd2$   
**shows**  $\forall\ \varepsilon.\ \varepsilon > 0 \wedge \varepsilon \leq \delta\text{-min}\ qd \longrightarrow (\forall\ qd1\ qd2.\ (qd1,\ qd2) \in set\ qd \longrightarrow val\ qd1\ \varepsilon \leq val\ qd2\ \varepsilon)$   
 $\langle proof \rangle$

**lemma** *QDelta-0-0*: *QDelta 0 0 = 0* *<proof>*  
**lemma** *qdsnd-0*: *qdsnd 0 = 0* *<proof>*  
**lemma** *qdfst-0*: *qdfst 0 = 0* *<proof>*

**end**

## 6 The Simplex Algorithm

**theory** *Simplex*

**imports**

*Linear-Poly-Maps*  
*QDelta*  
*Rel-Chain*  
*Simplex-Algebra*  
*HOL-Library.RBT-Mapping*  
*HOL-Library.Permutation*  
*HOL-Library.Code-Target-Numeral*

**begin**

Linear constraints are of the form  $p \bowtie c$  or  $p_1 \bowtie p_2$ , where  $p$ ,  $p_1$ , and  $p_2$ , are linear polynomials,  $c$  is a rational constant and  $\bowtie \in \{<, >, \leq, \geq, =\}$ . Their abstract syntax is given by the *constraint* type, and semantics is given by the relation  $\models_c$ , defined straightforwardly by primitive recursion over the *constraint* type. 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*  
| *LTPP linear-poly linear-poly*  
| *GTPP linear-poly linear-poly*  
| *LEQPP linear-poly linear-poly*  
| *GEQPP linear-poly linear-poly*  
| *EQPP linear-poly linear-poly*

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

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

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

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

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

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

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

**primrec**

*satisfies-constraint* :: 'a :: lrv valuation ⇒ constraint ⇒ bool (infixl  $\models_c$  100)

**where**

$v \models_c (LT\ l\ r) \longleftrightarrow (l\{v\}) < r * R\ 1$   
 $v \models_c (GT\ l\ r) \longleftrightarrow (l\{v\}) > r * R\ 1$   
 $v \models_c (LEQ\ l\ r) \longleftrightarrow (l\{v\}) \leq r * R\ 1$   
 $v \models_c (GEQ\ l\ r) \longleftrightarrow (l\{v\}) \geq r * R\ 1$   
 $v \models_c (EQ\ l\ r) \longleftrightarrow (l\{v\}) = r * R\ 1$   
 $v \models_c (LTPP\ l1\ l2) \longleftrightarrow (l1\{v\}) < (l2\{v\})$   
 $v \models_c (GTPP\ l1\ l2) \longleftrightarrow (l1\{v\}) > (l2\{v\})$   
 $v \models_c (LEQPP\ l1\ l2) \longleftrightarrow (l1\{v\}) \leq (l2\{v\})$   
 $v \models_c (GEQPP\ l1\ l2) \longleftrightarrow (l1\{v\}) \geq (l2\{v\})$   
 $v \models_c (EQPP\ l1\ l2) \longleftrightarrow (l1\{v\}) = (l2\{v\})$

**abbreviation** *satisfies-constraints* :: rat valuation ⇒ constraint set ⇒ bool (infixl  $\models_{cs}$  100) **where**

$v \models_{cs} cs \equiv \forall c \in cs. v \models_c c$

**lemma** *unsat-mono*: **assumes**  $\neg (\exists v. v \models_{cs} cs)$

**and**  $cs \subseteq ds$

**shows**  $\neg (\exists v. v \models_{cs} ds)$

*<proof>*

**fun** *i-satisfies-cs* (infixl  $\models_{ics}$  100) **where**

$(I, v) \models_{ics} cs \longleftrightarrow v \models_{cs} \text{restrict-to } I\ cs$

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

*distinct-indices as* = (*distinct* (*map fst as*))

**lemma** *distinct-indicesD*: *distinct-indices as* ⇒  $(i, x) \in \text{set } as \implies (i, y) \in \text{set } as \implies x = y$

*<proof>*

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

first  $c_2$ -constraint. Consequently, the index-set  $\{c_1, c_2\}$  will be returned, but this set is not minimal since  $\{c_2\}$  is already unsatisfiable.

**definition** *minimal-unsat-core* :: 'i set  $\Rightarrow$  'i i-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  $\equiv$  Inl

**abbreviation** (input) *Sat where* Sat  $\equiv$  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  $\Rightarrow$  'i list + rat valuation

— If the status *Sat* is returned, then returned valuation satisfies all constraints.

**assumes** *simplex-sat*: *solve cs* = *Sat v*  $\implies$  v  $\models_{cs}$  flat (set cs)

— If the status *Unsat* is returned, then constraints are unsatisfiable, i.e., an unsatisfiable core is returned.

**assumes** *simplex-unsat*: *solve cs* = *Unsat I*  $\implies$  *minimal-unsat-core* (set I) cs

**abbreviation** (input) *look where* look  $\equiv$  Mapping.lookup

**abbreviation** (input) *upd where* upd  $\equiv$  Mapping.update

**lemma** *look-upd*: look (upd k v m) = (look m)(k  $\mapsto$  v)

*<proof>*

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

**definition** *map2fun*:: (var, 'a :: zero) mapping  $\Rightarrow$  var  $\Rightarrow$  'a **where**

*map2fun* v  $\equiv$   $\lambda$ x. case look v x of None  $\Rightarrow$  0 | Some y  $\Rightarrow$  y

**syntax**

-*map2fun* :: (var, 'a) mapping  $\Rightarrow$  var  $\Rightarrow$  'a ((-))

**translations**

*<v>* == CONST *map2fun* v

**lemma** *map2fun-def'*:

*<v>* x  $\equiv$  case Mapping.lookup v x of None  $\Rightarrow$  0 | Some y  $\Rightarrow$  y

*<proof>*

Note that the above specification requires returning a valuation (defined as a HOL function), which is not efficiently executable. In order to enable more efficient data structures for representing valuations, a refinement of this specification is needed and the function *solve* is replaced by the function *solve-exec* returning optional (var, rat) mapping instead of var  $\Rightarrow$  rat function. This way, efficient data structures for representing mappings can be easily plugged-in during code generation [2]. A conversion from the *map-*



ping datatype to HOL function is denoted by  $\langle - \rangle$  and given by:  $\langle v \rangle x \equiv \text{case Mapping.lookup } v \text{ of None} \Rightarrow 0::'a \mid \text{Some } y \Rightarrow y$ .

**locale** *SolveExec* =

**fixes** *solve-exec* ::  $'i$  *i-constraint list*  $\Rightarrow 'i$  *list* + (*var, rat*) *mapping*

**assumes** *simplex-sat0*: *solve-exec cs* = *Sat v*  $\Longrightarrow \langle v \rangle \models_{cs}$  *flat (set cs)*

**assumes** *simplex-unsat0*: *solve-exec cs* = *Unsat I*  $\Longrightarrow$  *minimal-unsat-core (set I) cs*

**begin**

**definition** *solve* **where**

*solve cs*  $\equiv$  *case solve-exec cs of Sat v*  $\Rightarrow$  *Sat*  $\langle v \rangle$   $\mid$  *Unsat c*  $\Rightarrow$  *Unsat c*

**end**

**sublocale** *SolveExec* < *Solve solve*

$\langle \text{proof} \rangle$

## 6.2 Handling Strict Inequalities

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

**type-synonym**  $(i, 'a)$  *i-ns-constraint* =  $'i \times 'a$  *ns-constraint*

**primrec** *satisfiable-ns-constraint* ::  $'a::lrv$  *valuation*  $\Rightarrow 'a$  *ns-constraint*  $\Rightarrow$  *bool*

(**infixl**  $\models_{ns}$  100) **where**

$v \models_{ns}$  *LEQ-ns l r*  $\longleftrightarrow l\{v\} \leq r$

$\mid v \models_{ns}$  *GEQ-ns l r*  $\longleftrightarrow l\{v\} \geq r$

**abbreviation** *satisfies-ns-constraints* ::  $'a::lrv$  *valuation*  $\Rightarrow 'a$  *ns-constraint set*  $\Rightarrow$

*bool* (**infixl**  $\models_{nss}$  100) **where**

$v \models_{nss} cs \equiv \forall c \in cs. v \models_{ns} c$

**fun** *i-satisfies-ns-constraints* ::  $'i$  *set*  $\times 'a::lrv$  *valuation*  $\Rightarrow (i, 'a)$  *i-ns-constraint*

*set*  $\Rightarrow$  *bool* (**infixl**  $\models_{inss}$  100) **where**

$(I, v) \models_{inss} cs \longleftrightarrow v \models_{nss}$  *restrict-to I cs*

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

**primrec** *poly* :: 'a ns-constraint  $\Rightarrow$  linear-poly **where**  
 $poly (LEQ\text{-}ns\ p\ a) = p$   
 $| poly (GEQ\text{-}ns\ p\ a) = p$

**primrec** *ns-constraint-const* :: 'a ns-constraint  $\Rightarrow$  'a **where**  
 $ns\text{-}constraint\text{-}const (LEQ\text{-}ns\ p\ a) = a$   
 $| ns\text{-}constraint\text{-}const (GEQ\text{-}ns\ p\ a) = a$

**definition** *distinct-indices-ns* :: ('i, 'a :: lrv) i-ns-constraint set  $\Rightarrow$  bool **where**  
 $distinct\text{-}indices\text{-}ns\ ns = ((\forall\ n1\ n2\ i. (i, n1) \in ns \longrightarrow (i, n2) \in ns \longrightarrow$   
 $poly\ n1 = poly\ n2 \wedge ns\text{-}constraint\text{-}const\ n1 = ns\text{-}constraint\text{-}const\ n2))$

**definition** *minimal-unsat-core-ns* :: 'i set  $\Rightarrow$  ('i, 'a :: lrv) i-ns-constraint set  $\Rightarrow$  bool **where**  
 $minimal\text{-}unsat\text{-}core\text{-}ns\ I\ cs = ((I \subseteq fst\ ' cs) \wedge (\neg (\exists\ v. (I, v) \models_{inss} cs))$   
 $\wedge (distinct\text{-}indices\text{-}ns\ cs \longrightarrow (\forall\ J \subset I. \exists\ v. (J, v) \models_{inss} cs)))$

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

**locale** *To-ns* =

— Convert a constraint to an equisatisfiable non-strict constraint list. The conversion must work for arbitrary subsets of constraints – selected by some index set I – in order to carry over unsat-cores and in order to support incremental simplex solving.

**fixes** *to-ns* :: 'i i-constraint list  $\Rightarrow$  ('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  $\Rightarrow$  'a ns-constraint list  $\Rightarrow$  (var, rat) mapping

**assumes** *to-ns-unsat*:  $minimal\text{-}unsat\text{-}core\text{-}ns\ I\ (set\ (to\text{-}ns\ cs)) \implies minimal\text{-}unsat\text{-}core\ I\ cs$

**assumes** *i-to-ns-sat*:  $(I, \langle v^\wedge \rangle) \models_{inss} set\ (to\text{-}ns\ cs) \implies (I, \langle from\text{-}ns\ v' (flat\text{-}list\ (to\text{-}ns\ cs)) \rangle) \models_{ics} set\ cs$

**assumes** *to-ns-indices*:  $fst\ ' set\ (to\text{-}ns\ cs) = fst\ ' set\ cs$

**assumes** *distinct-cond*:  $distinct\text{-}indices\ cs \implies distinct\text{-}indices\text{-}ns\ (set\ (to\text{-}ns\ cs))$

**begin**

**lemma** *to-ns-sat*:  $\langle v^\wedge \rangle \models_{nss} flat\ (set\ (to\text{-}ns\ cs)) \implies \langle from\text{-}ns\ v' (flat\text{-}list\ (to\text{-}ns\ cs)) \rangle \models_{cs} flat\ (set\ cs)$

$\langle proof \rangle$

**end**

**locale** *Solve-exec-ns* =

**fixes** *solve-exec-ns* :: ('i, 'a :: lrv) i-ns-constraint list  $\Rightarrow$  'i list + (var, 'a) mapping

**assumes** *simplex-ns-sat*:  $solve\text{-}exec\text{-}ns\ cs = Sat\ v \implies \langle v \rangle \models_{nss} flat\ (set\ cs)$

**assumes** *simplex-ns-unsat*:  $solve\text{-}exec\text{-}ns\ cs = Unsat\ I \implies minimal\text{-}unsat\text{-}core\text{-}ns$

(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*  
⟨proof⟩

### 6.3 Preprocessing

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

**type-synonym** *eq* = var  $\times$  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*:  $v \models_e (x, p) \equiv v x = p\{v\}$

$\langle proof \rangle$

**type-synonym**  $tableau = eq\ list$

**definition**  $satisfies-tableau :: 'a::rational-vector\ valuation \Rightarrow tableau \Rightarrow bool$  (**infixl**  $\models_t$  100) **where**

$v \models_t t \equiv \forall e \in set\ t. v \models_e e$

**definition**  $lvars :: tableau \Rightarrow var\ set$  **where**

$lvars\ t = set\ (map\ lhs\ t)$

**definition**  $rvars :: tableau \Rightarrow var\ set$  **where**

$rvars\ t = \bigcup (set\ (map\ rvars-eq\ t))$

**abbreviation**  $tvvars$  **where**  $tvvars\ t \equiv lvars\ t \cup rvars\ t$

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

**definition**  $normalized-tableau :: tableau \Rightarrow bool$  ( $\Delta$ ) **where**

$normalized-tableau\ t \equiv distinct\ (map\ lhs\ t) \wedge lvars\ t \cap rvars\ t = \{\} \wedge 0 \notin rhs\ set\ t$

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  $v \models_e (x, p) \equiv v\ x = p \ \{\!| \ v \ |\!\}$ . 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  $\Delta\ t$ ) iff no variable occurs on the lhs of two equations in a tableau and if sets of lhs and rhs variables are distinct.

**lemma**  $normalized-tableau-unique-eq-for-lvar$ :

**assumes**  $\Delta\ t$

**shows**  $\forall x \in lvars\ t. \exists! p. (x, p) \in set\ t$

$\langle proof \rangle$

**lemma**  $recalc-tableau-lvars$ :

**assumes**  $\Delta\ t$

**shows**  $\forall v. \exists v'. (\forall x \in rvars\ t. v\ x = v'\ x) \wedge v' \models_t t$

$\langle proof \rangle$

**lemma**  $tableau-perm$ :

**assumes**  $lvars\ t1 = lvars\ t2\ rvars\ t1 = rvars\ t2$   
 $\Delta\ t1\ \Delta\ t2\ \wedge\ v::'a::lrv\ \text{valuation}.\ v \models_t t1 \longleftrightarrow v \models_t t2$   
**shows**  $t1 <\sim\sim> t2$   
 $\langle\text{proof}\rangle$

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

**datatype**  $'a\ \text{atom} = \text{Leq}\ \text{var}\ 'a \quad | \quad \text{Geq}\ \text{var}\ 'a$

**primrec**  $\text{atom-var}::'a\ \text{atom} \Rightarrow \text{var}$  **where**  
 $\text{atom-var}\ (\text{Leq}\ \text{var}\ a) = \text{var}$   
 $| \text{atom-var}\ (\text{Geq}\ \text{var}\ a) = \text{var}$

**primrec**  $\text{atom-const}::'a\ \text{atom} \Rightarrow 'a$  **where**  
 $\text{atom-const}\ (\text{Leq}\ \text{var}\ a) = a$   
 $| \text{atom-const}\ (\text{Geq}\ \text{var}\ a) = a$

**primrec**  $\text{satisfies-atom}::'a::\text{linorder}\ \text{valuation} \Rightarrow 'a\ \text{atom} \Rightarrow \text{bool}$  (**infixl**  $\models_a\ 100$ )  
**where**  
 $v \models_a \text{Leq}\ x\ c \longleftrightarrow v\ x \leq c \quad | \quad v \models_a \text{Geq}\ x\ c \longleftrightarrow v\ x \geq c$

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

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

**lemma**  $\text{satisfies-atom}'\text{-stronger}:\ v \models_{ae}\ a \Longrightarrow v \models_a\ a$   $\langle\text{proof}\rangle$

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

**lemma**  $\text{satisfies-atom-set}'\text{-stronger}:\ v \models_{aes}\ as \Longrightarrow v \models_{as}\ as$   
 $\langle\text{proof}\rangle$

There is also the indexed variant of an atom

**type-synonym**  $(i,'a)\ i\text{-atom} = 'i \times 'a\ \text{atom}$

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

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

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

**lemma** *satisfies-atom-restrict-to-Cons*:  $v \models_{as} \text{restrict-to } I \text{ (set } as) \implies (i \in I \implies v \models_a a)$   
 $\implies v \models_{as} \text{restrict-to } I \text{ (set } ((i,a) \# as))$   
 ⟨proof⟩

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

**definition** *distinct-indices-atoms* ::  $('i,'a) \text{ i-atom set} \Rightarrow \text{bool}$  **where**  
*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*:: $('i,'a::lrv) \text{ i-ns-constraint list} \Rightarrow \text{tableau} \times ('i,'a) \text{ i-atom list}$   
 $\times ((\text{var},'a) \text{ mapping} \Rightarrow (\text{var},'a) \text{ mapping}) \times 'i \text{ list}$   
**assumes**

— The returned tableau is always normalized.

*preprocess-tableau-normalized*:  $\text{preprocess } cs = (t, as, \text{trans-}v, U) \implies \Delta t$  **and**

— Tableau and atoms are equisatisfiable with starting non-strict constraints.

*i-preprocess-sat*:  $\bigwedge v. \text{preprocess } cs = (t, as, \text{trans-}v, U) \implies I \cap \text{set } U = \{\} \implies (I, \langle v \rangle) \models_{ias} \text{set } as \implies \langle v \rangle \models_t t \implies (I, \langle \text{trans-}v \ v \rangle) \models_{inss} \text{set } cs$  **and**

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

— distinct indices on ns-constraints ensures distinct indices in atoms

*preprocess-distinct*:  $\text{preprocess } cs = (t, as, \text{trans-}v, U) \implies \text{distinct-indices-ns (set } cs) \implies \text{distinct-indices-atoms (set } as)$  **and**

— unsat indices

*preprocess-unsat-indices*:  $\text{preprocess } cs = (t, as, \text{trans-}v, U) \implies i \in \text{set } U \implies \neg (\exists v. (\{i\}, v) \models_{inss} \text{set } cs)$  **and**

— preprocessing cannot introduce new indices

*preprocess-index*:  $\text{preprocess } cs = (t, as, \text{trans-}v, U) \implies \text{fst } ' \text{ set } as \cup \text{set } U \subseteq \text{fst } ' \text{ set } cs$

**begin**

**lemma** *preprocess-sat*:  $\text{preprocess } cs = (t, as, \text{trans-}v, U) \implies U = [] \implies \langle v \rangle \models_{as} \text{flat (set } as) \implies \langle v \rangle \models_t t \implies \langle \text{trans-}v \ v \rangle \models_{nss} \text{flat (set } cs)$   
 ⟨proof⟩

**end**

**definition** *minimal-unsat-core-tabl-atoms* ::  $'i \text{ set} \Rightarrow \text{tableau} \Rightarrow ('i,'a::lrv) \text{ i-atom}$

*set*  $\Rightarrow$  *bool* **where**

*minimal-unsat-core-tabl-atoms* *I t as* = ( *I*  $\subseteq$  *fst* ' *as*  $\wedge$  ( $\neg$  ( $\exists v. v \models_t t \wedge (I, v) \models_{ias} as$ ))  $\wedge$   
(*distinct-indices-atoms* *as*  $\longrightarrow$  ( $\forall J \subset I. \exists v. v \models_t t \wedge (J, v) \models_{iaes} as$ )))

**lemma** *minimal-unsat-core-tabl-atomsD*: **assumes** *minimal-unsat-core-tabl-atoms* *I t as*

**shows** *I*  $\subseteq$  *fst* ' *as*

$\neg$  ( $\exists v. v \models_t t \wedge (I, v) \models_{ias} as$ )

*distinct-indices-atoms* *as*  $\Longrightarrow J \subset I \Longrightarrow \exists v. v \models_t t \wedge (J, v) \models_{iaes} as$

*<proof>*

**locale** *AssertAll* =

**fixes** *assert-all* :: *tableau*  $\Rightarrow$  ('*i*, '*a*::*lrv*) *i-atom list*  $\Rightarrow$  '*i list* + (*var*, '*a*)*mapping*

**assumes** *assert-all-sat*:  $\Delta t \Longrightarrow \text{assert-all } t \text{ as} = \text{Sat } v \Longrightarrow \langle v \rangle \models_t t \wedge \langle v \rangle \models_{as}$

*flat* (*set as*)

**assumes** *assert-all-unsat*:  $\Delta t \Longrightarrow \text{assert-all } t \text{ as} = \text{Unsat } I \Longrightarrow \text{minimal-unsat-core-tabl-atoms}$   
(*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>*

**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*⟩  
**end**

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

## 6.4 Incrementally Asserting Atoms

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

**type-synonym** *'a bounds = var → 'a*

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

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

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

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

**abbreviation** (*input*) *le where*

$$le \text{ } lt \text{ } x \text{ } y \equiv lt \text{ } x \text{ } y \vee x = y$$

**definition** *geub* ( $\triangleright_{ub}$ ) **where**

$$\triangleright_{ub} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow False \mid \text{Some } b' \Rightarrow le \text{ } lt \text{ } b' \text{ } c$$

**definition** *gtub* ( $\triangleright_{ub}$ ) **where**

$$\triangleright_{ub} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow False \mid \text{Some } b' \Rightarrow lt \text{ } b' \text{ } c$$

**definition** *leub* ( $\triangleleft_{ub}$ ) **where**

$$\triangleleft_{ub} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow True \mid \text{Some } b' \Rightarrow le \text{ } lt \text{ } c \text{ } b'$$

**definition** *ltub* ( $\triangleleft_{ub}$ ) **where**

$$\triangleleft_{ub} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow True \mid \text{Some } b' \Rightarrow lt \text{ } c \text{ } b'$$

**definition** *lelb* ( $\triangleleft_{lb}$ ) **where**

$$\triangleleft_{lb} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow False \mid \text{Some } b' \Rightarrow le \text{ } lt \text{ } c \text{ } b'$$

**definition** *ltlb* ( $\triangleleft_{lb}$ ) **where**

$$\triangleleft_{lb} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow False \mid \text{Some } b' \Rightarrow lt \text{ } c \text{ } b'$$

**definition** *gelb* ( $\triangleright_{lb}$ ) **where**

$$\triangleright_{lb} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow True \mid \text{Some } b' \Rightarrow le \text{ } lt \text{ } b' \text{ } c$$

**definition** *gtlb* ( $\triangleright_{lb}$ ) **where**

$$\triangleright_{lb} \text{ } lt \text{ } c \text{ } b \equiv \text{case } b \text{ of } None \Rightarrow True \mid \text{Some } b' \Rightarrow lt \text{ } b' \text{ } c$$

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

$$c \geq_{ub} b = \triangleright_{ub} (<) \text{ } c \text{ } b$$

**definition** *gt-ubound* :: *'a::linorder*  $\Rightarrow$  *'a option*  $\Rightarrow$  *bool* (**infixl**  $>_{ub}$  100) **where**



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

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

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

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

**lemma** *compare-strict-nonstrict*:  
 $x <_{ub} b \implies x \leq_{ub} b$   
 $x >_{ub} b \implies x \geq_{ub} b$   
 $x <_{lb} b \implies x \leq_{lb} b$   
 $x >_{lb} b \implies x \geq_{lb} b$   
 ⟨proof⟩

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

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

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

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

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

**lemma** *satisfies-bounds-iff*:

$$v \models_b (lb, ub) \longleftrightarrow (\forall x. v x \geq_{lb} lb x \wedge v x \leq_{ub} ub x)$$

*<proof>*

**lemma** *not-in-bounds*:

$$\neg (in\_bounds\ x\ v\ (lb, ub)) = (v x <_{lb} lb x \vee v x >_{ub} ub x)$$

*<proof>*

**fun** *atoms-equiv-bounds* :: *'a::linorder atom set*  $\Rightarrow$  *'a bounds*  $\times$  *'a bounds*  $\Rightarrow$  *bool*

**(infixl**  $\doteq$  100) **where**

$$as \doteq (lb, ub) \longleftrightarrow (\forall v. v \models_{as} as \longleftrightarrow v \models_b (lb, ub))$$

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

**lemma** *atoms-equiv-bounds-simps*:

$$as \doteq (lb, ub) \equiv \forall v. v \models_{as} as \longleftrightarrow v \models_b (lb, ub)$$

*<proof>*

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

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

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

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

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

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

**datatype** (*'i, 'a*) *state* = *State*

( $\mathcal{T}$ : *tableau*)

( $\mathcal{B}_l$ : (*'i, 'a*) *bounds-index*)

( $\mathcal{B}_u$ : (*'i, 'a*) *bounds-index*)

( $\mathcal{V}$ : (*var*, *'a*) *mapping*)

( $\mathcal{U}$ : *bool*)

( $\mathcal{U}_c$ : *'i list option*)

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

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

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

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

**abbreviation**  $BoundsIndicesMap (\mathcal{B}_i) \text{ where } \mathcal{B}_i s \equiv (\mathcal{B}_{il} s, \mathcal{B}_{iu} s)$

**abbreviation**  $Bounds :: ('i, 'a) \text{ state} \Rightarrow 'a \text{ bounds} \times 'a \text{ bounds } (\mathcal{B}) \text{ where } \mathcal{B} s \equiv$   
 $(\mathcal{B}_l s, \mathcal{B}_u s)$

**abbreviation**  $Indices :: ('i, 'a) \text{ state} \Rightarrow 'i \text{ bound-index} \times 'i \text{ bound-index } (\mathcal{I}) \text{ where}$   
 $\mathcal{I} s \equiv (\mathcal{I}_l s, \mathcal{I}_u s)$

**abbreviation**  $BoundsIndices :: ('i, 'a) \text{ state} \Rightarrow ('a \text{ bounds} \times 'a \text{ bounds}) \times ('i$   
 $\text{bound-index} \times 'i \text{ bound-index}) (\mathcal{BI})$

**where**  $\mathcal{BI} s \equiv (\mathcal{B} s, \mathcal{I} s)$

**fun**  $satisfies\_bounds\_index :: 'i \text{ set} \times 'a::\text{lrval} \text{ valuation} \Rightarrow ('a \text{ bounds} \times 'a \text{ bounds})$   
 $\times$

$('i \text{ bound-index} \times 'i \text{ bound-index}) \Rightarrow \text{bool} \text{ (infixl } \models_{ib} 100) \text{ where}$

$(I, v) \models_{ib} ((BL, BU), (IL, IU)) \iff ($

$(\forall x c. BL x = \text{Some } c \longrightarrow IL x \in I \longrightarrow v x \geq c)$

$\wedge (\forall x c. BU x = \text{Some } c \longrightarrow IU x \in I \longrightarrow v x \leq c))$

**declare**  $satisfies\_bounds\_index.simps[simp del]$

**fun**  $satisfies\_bounds\_index' :: 'i \text{ set} \times 'a::\text{lrval} \text{ valuation} \Rightarrow ('a \text{ bounds} \times 'a \text{ bounds})$   
 $\times$

$('i \text{ bound-index} \times 'i \text{ bound-index}) \Rightarrow \text{bool} \text{ (infixl } \models_{ibe} 100) \text{ where}$

$(I, v) \models_{ibe} ((BL, BU), (IL, IU)) \iff ($

$(\forall x c. BL x = \text{Some } c \longrightarrow IL x \in I \longrightarrow v x = c)$

$\wedge (\forall x c. BU x = \text{Some } c \longrightarrow IU x \in I \longrightarrow v x = c))$

**declare**  $satisfies\_bounds\_index'.simps[simp del]$

**fun**  $atoms\_imply\_bounds\_index :: ('i, 'a::\text{lrval}) \text{ i-atom set} \Rightarrow ('a \text{ bounds} \times 'a \text{ bounds})$   
 $\times ('i \text{ bound-index} \times 'i \text{ bound-index})$

$\Rightarrow \text{bool} \text{ (infixl } \models_i 100) \text{ where}$

$as \models_i bi \iff (\forall I v. (I, v) \models_{ias} as \longrightarrow (I, v) \models_{ib} bi)$

**declare**  $atoms\_imply\_bounds\_index.simps[simp del]$

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

**lemma**  $atoms\_imply\_bounds\_index\text{-mono}: as \subseteq as' \Longrightarrow as \models_i bi \Longrightarrow as' \models_i bi$   
 $\langle \text{proof} \rangle$

**definition** *satisfies-state* :: 'a::lrv valuation  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  bool (infixl  $\models_s$  100) **where**

$$v \models_s s \equiv v \models_b \mathcal{B} s \wedge v \models_t \mathcal{T} s$$

**definition** *curr-val-satisfies-state* :: ('i,'a::lrv) state  $\Rightarrow$  bool ( $\models$ ) **where**

$$\models s \equiv \langle \mathcal{V} s \rangle \models_s s$$

**fun** *satisfies-state-index* :: 'i set  $\times$  'a::lrv valuation  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  bool (infixl  $\models_{is}$  100) **where**

$$(I,v) \models_{is} s \iff (v \models_t \mathcal{T} s \wedge (I,v) \models_{ib} \mathcal{BI} s)$$

**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**

$$(I,v) \models_{ise} s \iff (v \models_t \mathcal{T} s \wedge (I,v) \models_{ibe} \mathcal{BI} s)$$

**declare** *satisfies-state-index'.simps*[simp del]

**definition** *indices-state* :: ('i,'a)state  $\Rightarrow$  'i set **where**

$$\text{indices-state } s = \{ i. \exists x b. \text{look } (\mathcal{B}_{il} s) x = \text{Some } (i,b) \vee \text{look } (\mathcal{B}_{iu} s) x = \text{Some } (i,b) \}$$

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**

$$\begin{aligned} \text{distinct-indices-state } s &= (\forall i x b x' b'. \\ &((\text{look } (\mathcal{B}_{il} s) x = \text{Some } (i,b) \vee \text{look } (\mathcal{B}_{iu} s) x = \text{Some } (i,b)) \longrightarrow \\ &(\text{look } (\mathcal{B}_{il} s) x' = \text{Some } (i,b') \vee \text{look } (\mathcal{B}_{iu} s) x' = \text{Some } (i,b')) \longrightarrow \\ &(x = x' \wedge b = b'))) \end{aligned}$$

**lemma** *distinct-indices-stateD*: **assumes** *distinct-indices-state*  $s$

**shows**  $\text{look } (\mathcal{B}_{il} s) x = \text{Some } (i,b) \vee \text{look } (\mathcal{B}_{iu} s) x = \text{Some } (i,b) \implies \text{look } (\mathcal{B}_{il} s) x' = \text{Some } (i,b') \vee \text{look } (\mathcal{B}_{iu} s) x' = \text{Some } (i,b')$   
 $\implies x = x' \wedge b = b'$

*<proof>*

**definition** *unsat-state-core* :: ('i,'a::lrv) state  $\Rightarrow$  bool **where**

$$\text{unsat-state-core } s = (\text{set } (\text{the } (\mathcal{U}_c s)) \subseteq \text{indices-state } s \wedge (\neg (\exists v. (\text{set } (\text{the } (\mathcal{U}_c s)), v) \models_{is} s)))$$

**definition** *subsets-sat-core* :: ('i,'a::lrv) state  $\Rightarrow$  bool **where**

$$\text{subsets-sat-core } s = ((\forall I. I \subset \text{set } (\text{the } (\mathcal{U}_c s)) \longrightarrow (\exists v. (I,v) \models_{ise} s)))$$

**definition** *minimal-unsat-state-core* :: ('i,'a::lrv) state  $\Rightarrow$  bool **where**

$$\text{minimal-unsat-state-core } s = (\text{unsat-state-core } s \wedge (\text{distinct-indices-state } s \longrightarrow \text{subsets-sat-core } s))$$

**lemma** *minimal-unsat-core-tabl-atoms-mono*: **assumes** *sub*:  $as \subseteq bs$

**and** *unsat*: *minimal-unsat-core-tabl-atoms*  $I t as$

**shows** *minimal-unsat-core-tabl-atoms*  $I t bs$

*<proof>*

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

**lemma** *unsat-state-core-unsat*:  $unsat\text{-state-core } s \implies \neg (\exists v. v \models_s s)$   
*<proof>*

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

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

**lemma** *index-valid-indices-state*:  $index\text{-valid } as s \implies indices\text{-state } s \subseteq fst ' as$   
*<proof>*

**lemma** *index-valid-mono*:  $as \subseteq bs \implies index\text{-valid } as s \implies index\text{-valid } bs s$   
*<proof>*

**lemma** *index-valid-distinct-indices*: **assumes**  $index\text{-valid } as s$   
**and** *distinct-indices-atoms*  $as$   
**shows** *distinct-indices-state*  $s$   
*<proof>*

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

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

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

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

**fun**  $\mathcal{V}$ -*update* **where**

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

**fun**  $\mathcal{T}$ -update **where**

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

**lemma** *update-simps*[*simp*]:

$\mathcal{B}_{iu}$  ( $\mathcal{B}_{iu}$ -update  $up$   $s$ ) =  $up$  ( $\mathcal{B}_{iu}$   $s$ )

$\mathcal{B}_{il}$  ( $\mathcal{B}_{iu}$ -update  $up$   $s$ ) =  $\mathcal{B}_{il}$   $s$

$\mathcal{T}$  ( $\mathcal{B}_{iu}$ -update  $up$   $s$ ) =  $\mathcal{T}$   $s$

$\mathcal{V}$  ( $\mathcal{B}_{iu}$ -update  $up$   $s$ ) =  $\mathcal{V}$   $s$

$\mathcal{U}$  ( $\mathcal{B}_{iu}$ -update  $up$   $s$ ) =  $\mathcal{U}$   $s$

$\mathcal{U}_c$  ( $\mathcal{B}_{iu}$ -update  $up$   $s$ ) =  $\mathcal{U}_c$   $s$

$\mathcal{B}_{il}$  ( $\mathcal{B}_{il}$ -update  $up$   $s$ ) =  $up$  ( $\mathcal{B}_{il}$   $s$ )

$\mathcal{B}_{iu}$  ( $\mathcal{B}_{il}$ -update  $up$   $s$ ) =  $\mathcal{B}_{iu}$   $s$

$\mathcal{T}$  ( $\mathcal{B}_{il}$ -update  $up$   $s$ ) =  $\mathcal{T}$   $s$

$\mathcal{V}$  ( $\mathcal{B}_{il}$ -update  $up$   $s$ ) =  $\mathcal{V}$   $s$

$\mathcal{U}$  ( $\mathcal{B}_{il}$ -update  $up$   $s$ ) =  $\mathcal{U}$   $s$

$\mathcal{U}_c$  ( $\mathcal{B}_{il}$ -update  $up$   $s$ ) =  $\mathcal{U}_c$   $s$

$\mathcal{V}$  ( $\mathcal{V}$ -update  $V$   $s$ ) =  $V$

$\mathcal{B}_{il}$  ( $\mathcal{V}$ -update  $V$   $s$ ) =  $\mathcal{B}_{il}$   $s$

$\mathcal{B}_{iu}$  ( $\mathcal{V}$ -update  $V$   $s$ ) =  $\mathcal{B}_{iu}$   $s$

$\mathcal{T}$  ( $\mathcal{V}$ -update  $V$   $s$ ) =  $\mathcal{T}$   $s$

$\mathcal{U}$  ( $\mathcal{V}$ -update  $V$   $s$ ) =  $\mathcal{U}$   $s$

$\mathcal{U}_c$  ( $\mathcal{V}$ -update  $V$   $s$ ) =  $\mathcal{U}_c$   $s$

$\mathcal{T}$  ( $\mathcal{T}$ -update  $T$   $s$ ) =  $T$

$\mathcal{B}_{il}$  ( $\mathcal{T}$ -update  $T$   $s$ ) =  $\mathcal{B}_{il}$   $s$

$\mathcal{B}_{iu}$  ( $\mathcal{T}$ -update  $T$   $s$ ) =  $\mathcal{B}_{iu}$   $s$

$\mathcal{V}$  ( $\mathcal{T}$ -update  $T$   $s$ ) =  $\mathcal{V}$   $s$

$\mathcal{U}$  ( $\mathcal{T}$ -update  $T$   $s$ ) =  $\mathcal{U}$   $s$

$\mathcal{U}_c$  ( $\mathcal{T}$ -update  $T$   $s$ ) =  $\mathcal{U}_c$   $s$

*<proof>*

**declare**

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

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

**fun** *set-unsat* :: *'i list*  $\Rightarrow$  (*'i,'a*) *state*  $\Rightarrow$  (*'i,'a*) *state* **where**

*set-unsat*  $I$  (State  $T$  BIL BIU  $V$   $U$  UC) = State  $T$  BIL BIU  $V$  True (Some (remdups  $I$ ))

**lemma** *set-unsat-simps*[*simp*]:  $\mathcal{B}_{il}$  (*set-unsat*  $I$   $s$ ) =  $\mathcal{B}_{il}$   $s$

$\mathcal{B}_{iu}$  (*set-unsat*  $I$   $s$ ) =  $\mathcal{B}_{iu}$   $s$

$\mathcal{T}$  (*set-unsat*  $I$   $s$ ) =  $\mathcal{T}$   $s$

$\mathcal{V}$  (*set-unsat*  $I$   $s$ ) =  $\mathcal{V}$   $s$

$\mathcal{U}$  (*set-unsat*  $I$   $s$ ) = True

$\mathcal{U}_c$  (*set-unsat*  $I$   $s$ ) = Some (remdups  $I$ )

*<proof>*

**datatype** (*'i,'a*) *Direction* = *Direction*

(lt: 'a::linorder  $\Rightarrow$  '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  $\Longrightarrow$  *assert-all-state* t as = s'  $\Longrightarrow$  (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  $\Longrightarrow$  *assert-all-state* t as = s'  $\Longrightarrow$   $\neg \mathcal{U}$  s'  $\Longrightarrow$   $\models$  s' **and**

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

— If  $\mathcal{U}$  is raised, then there is no valuation satisfying the tableau and the bounds in the final state (that are, in this case, equivalent to a subset of asserted atoms).



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

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

**end**

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

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

**locale** *Init'* =

**fixes** *init* :: *tableau*  $\Rightarrow$   $(i, a::\text{lrval})$  *state*

**assumes** *init'-tableau-normalized*:  $\Delta t \implies \Delta (\mathcal{T} (\text{init } t))$

**assumes** *init'-tableau-equiv*:  $\Delta t \implies (v::\text{'a valuation}) \models_t t = v \models_t \mathcal{T} (\text{init } t)$

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

**locale** *Init* = **fixes** *init*::*tableau*  $\Rightarrow$   $(i, a::\text{lrval})$  *state*

**assumes**

— Tableau in the initial state for  $t$  is  $t$ : *init-tableau-id*:  $\mathcal{T} (\text{init } t) = t$  **and**

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

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

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

— All tableau vars are valuated: *init-tableau-valuated*:  $\nabla (init\ t)$

**begin**

**lemma** *init-satisfies-bounds*:

$\langle \mathcal{V} (init\ t) \rangle \models_b \mathcal{B} (init\ t)$   
*<proof>*

**lemma** *init-satisfies*:

$\models (init\ t)$   
*<proof>*

**lemma** *init-atoms-equiv-bounds*:

$\{\} \doteq \mathcal{B} (init\ t)$   
*<proof>*

**lemma** *init-atoms-imply-bounds-index*:

$\{\} \models_i \mathcal{BI} (init\ t)$   
*<proof>*

**lemma** *init-tableau-normalized*:

$\Delta t \implies \Delta (\mathcal{T} (init\ t))$   
*<proof>*

**lemma** *init-index-valid: index-valid as (init t)*

*<proof>*

**lemma** *init-indices: indices-state (init t) = {}*

*<proof>*

**end**

**sublocale** *Init < Init' init*

*<proof>*

**abbreviation** *vars-list where*

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

**lemma** *tvars*  $t = \text{set } (\text{vars-list } t)$

*<proof>*

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  $\nabla s$  hold. The specification for *assert* directly follows from the specification of *assert-all-state* (except that it is additionally required that bounds reflect asserted atoms also when unsatisfiability is detected, and that it is required that *assert* keeps the tableau normalized and valuated).

**locale** *Assert* = **fixes** *assert::('i,'a::lrv) i-atom  $\Rightarrow$  ('i,'a) state  $\Rightarrow$  ('i,'a) state*

**assumes**

— Tableau remains equivalent to the previous one and normalized and valuated.

*assert-tableau*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \text{let } s' = \text{assert } a \text{ s in}$   
 $((v::'a \text{ valuation}) \models_t \mathcal{T} s \longleftrightarrow v \models_t \mathcal{T} s') \wedge \Delta (\mathcal{T} s') \wedge \nabla s'$  **and**

— If the  $\mathcal{U}$  flag is not raised, then the current valuation is updated so that it satisfies the current tableau and the current bounds.

*assert-sat*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \neg \mathcal{U} (\text{assert } a \text{ s}) \Longrightarrow \models (\text{assert } a \text{ s})$   
**and**

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

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

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

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

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

**begin**

**lemma** *assert-tableau-equiv*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow (v::'a \text{ valuation}) \models_t$   
 $\mathcal{T} s \longleftrightarrow v \models_t \mathcal{T} (\text{assert } a \text{ s})$

*<proof>*

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

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

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

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

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

**lemma** *AssertAllState'-precond*:  
 $\Delta t \Longrightarrow \Delta (\mathcal{T} (\text{assert-all-state } t \text{ as}))$   
 $\wedge (\nabla (\text{assert-all-state } t \text{ as}))$   
 $\wedge (\neg \mathcal{U} (\text{assert-all-state } t \text{ as}) \longrightarrow \models (\text{assert-all-state } t \text{ as}))$   
*<proof>*

**lemma** *AssertAllState'Induct*:  
**assumes**  
 $\Delta t$   
 $P \{ \} (\text{init } t)$   
 $\bigwedge as \ bs \ t. as \subseteq bs \Longrightarrow P \ as \ t \Longrightarrow P \ bs \ t$   
 $\bigwedge s \ a \ as. \llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s; P \ as \ s; \text{index-valid } as \ s \rrbracket \Longrightarrow P (\text{insert } a \ as) (\text{assert } a \ s)$   
**shows**  $P (\text{set } as) (\text{assert-all-state } t \ as)$   
*<proof>*

**lemma** *AssertAllState'-index-valid*:  $\Delta t \Longrightarrow \text{index-valid } (\text{set } as) (\text{assert-all-state } t \ as)$   
*<proof>*

**lemma** *AssertAllState'-sat-atoms-equiv-bounds*:  
 $\Delta t \Longrightarrow \neg \mathcal{U} (\text{assert-all-state } t \ as) \Longrightarrow \text{flat } (\text{set } as) \doteq \mathcal{B} (\text{assert-all-state } t \ as)$   
*<proof>*

**lemma** *AssertAllState'-unsat-atoms-implies-bounds*:

```

assumes  $\Delta t$ 
shows  $set\ as \models_i \mathcal{BI} (assert\text{-}all\text{-}state\ t\ as)$ 
 $\langle proof \rangle$ 

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  $AssertAllState' < AssertAllState\ assert\text{-}all\text{-}state$ 
 $\langle proof \rangle$ 

```

## 6.5 Asserting Single Atoms

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

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

```

fun  $satisfies\text{-}bounds\text{-}set :: 'a::linorder\ valuation \Rightarrow 'a\ bounds \times 'a\ bounds \Rightarrow var\ set \Rightarrow bool$  where

```

```

 $satisfies\text{-}bounds\text{-}set\ v\ (lb,\ ub)\ S \iff (\forall x \in S.\ in\text{-}bounds\ x\ v\ (lb,\ ub))$ 

```

```

declare  $satisfies\text{-}bounds\text{-}set.simps [simp\ del]$ 

```

```

syntax

```

```

 $\text{-}satisfies\text{-}bounds\text{-}set :: (var \Rightarrow 'a::linorder) \Rightarrow 'a\ bounds \times 'a\ bounds \Rightarrow var\ set \Rightarrow bool$ 
 $(- \models_b - \parallel -)$ 

```

```

translations

```

```

 $v \models_b b \parallel S == CONST\ satisfies\text{-}bounds\text{-}set\ v\ b\ S$ 

```

```

lemma  $satisfies\text{-}bounds\text{-}set\text{-}iff:$ 

```

```

 $v \models_b (lb,\ ub) \parallel S \equiv (\forall x \in S.\ v\ x \geq_{lb}\ lb\ x \wedge v\ x \leq_{ub}\ ub\ x)$ 

```

```

 $\langle proof \rangle$ 

```

```

definition  $curr\text{-}val\text{-}satisfies\text{-}no\text{-}lhs (\models_{no\text{-}lhs})$  where

```

```

 $\models_{no\text{-}lhs}\ s \equiv \langle \mathcal{V}\ s \rangle \models_t (\mathcal{T}\ s) \wedge (\langle \mathcal{V}\ s \rangle \models_b (\mathcal{B}\ s) \parallel (-\ lvars\ (\mathcal{T}\ s)))$ 

```

```

lemma  $satisfies\text{-}satisfies\text{-}no\text{-}lhs:$ 

```

$\models s \implies \models_{\text{no lhs}} s$   
 ⟨proof⟩

**definition** *bounds-consistent* :: ('i,'a::linorder) state  $\Rightarrow$  bool ( $\diamond$ ) **where**

$\diamond s \equiv$   
 $\forall x. \text{if } \mathcal{B}_l s x = \text{None} \vee \mathcal{B}_u s x = \text{None} \text{ then True else the } (\mathcal{B}_l s x) \leq \text{the } (\mathcal{B}_u s x)$

So, the *assert-bound* function must ensure that the given atom is included in the bounds, that the tableau remains satisfied by the valuation and that all variables except the lhs variables in the tableau are within their bounds. To formalize this, we introduce the notation  $v \models_b (lb, ub) \parallel S$ , and define  $v \models_b (lb, ub) \parallel S \equiv \forall x \in S. v x \geq_{lb} lb x \wedge v x \leq_{ub} ub x$ , and  $\models_{\text{no lhs}} s \equiv \langle \mathcal{V} s \rangle \models_t \mathcal{T} s \wedge \langle \mathcal{V} s \rangle \models_b \mathcal{B} s \parallel - \text{lvars } (\mathcal{T} s)$ . The *assert-bound* function raises the  $\mathcal{U}$  flag if and only if lower and upper bounds overlap. This is formalized as  $\diamond s \equiv \forall x. \text{if } \mathcal{B}_l s x = \text{None} \vee \mathcal{B}_u s x = \text{None} \text{ then True else the } (\mathcal{B}_l s x) \leq \text{the } (\mathcal{B}_u s x)$ .

**lemma** *satisfies-bounds-consistent*:

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

**lemma** *satisfies-consistent*:

$\models s \longrightarrow \diamond s$   
 ⟨proof⟩

**lemma** *bounds-consistent-geq-lb*:

$[\diamond s; \mathcal{B}_u s x_i = \text{Some } c]$   
 $\implies c \geq_{lb} \mathcal{B}_l s x_i$   
 ⟨proof⟩

**lemma** *bounds-consistent-leq-ub*:

$[\diamond s; \mathcal{B}_l s x_i = \text{Some } c]$   
 $\implies c \leq_{ub} \mathcal{B}_u s x_i$   
 ⟨proof⟩

**lemma** *bounds-consistent-gt-ub*:

$[\diamond s; c <_{lb} \mathcal{B}_l s x] \implies \neg c >_{ub} \mathcal{B}_u s x$   
 ⟨proof⟩

**lemma** *bounds-consistent-lt-lb*:

$[\diamond s; c >_{ub} \mathcal{B}_u s x] \implies \neg c <_{lb} \mathcal{B}_l s x$   
 ⟨proof⟩

Since the *assert-bound* is the first step in the *assert* function implementation, the preconditions for *assert-bound* are the same as preconditions for the *assert* function. The 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*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \text{assert-bound } a s = s' \Longrightarrow \mathcal{T} s' = \mathcal{T} s \wedge \nabla s' \text{ and}$

— 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*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \text{assert-bound } a s = s' \Longrightarrow \neg \mathcal{U} s' \Longrightarrow \models_{\text{noLhs}} s' \wedge \diamond s' \text{ and}$

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

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

*assert-bound-atoms-imply-bounds-index*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \text{ats} \models_i \mathcal{BI} s \Longrightarrow \text{insert } a \text{ ats} \models_i \mathcal{BI} (\text{assert-bound } a s) \text{ and}$

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

*assert-bound-unsat*:  $\llbracket \neg \mathcal{U} s; \models s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \text{index-valid } a s \Longrightarrow \text{assert-bound } a s = s' \Longrightarrow \mathcal{U} s' \Longrightarrow \text{minimal-unsat-state-core } s' \text{ and}$

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

**begin**

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

*<proof>*

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

*<proof>*

**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*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noLhs}} s; \Delta (\mathcal{T} s); \nabla s; \diamond s \rrbracket \Longrightarrow \mathcal{T} (\text{assert-bound } a s) = \mathcal{T} s$

**assumes** *assert-bound-nolhs-sat*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noLhs}} s; \Delta (\mathcal{T} s); \nabla s; \diamond s \rrbracket \Longrightarrow$

$\neg \mathcal{U} (\text{assert-bound } a s) \Longrightarrow \models_{\text{noLhs}} (\text{assert-bound } a s) \wedge \diamond (\text{assert-bound } a s)$

**assumes** *assert-bound-nolhs-atoms-equiv-bounds*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noLhs}} s; \Delta (\mathcal{T} s); \nabla s; \diamond s \rrbracket \Longrightarrow$

$\text{flat } \text{ats} \doteq \mathcal{B} s \Longrightarrow \text{flat } (\text{ats} \cup \{a\}) \doteq \mathcal{B} (\text{assert-bound } a s)$

**assumes** *assert-bound-nolhs-atoms-imply-bounds-index*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noLhs}} s; \Delta (\mathcal{T} s); \nabla s; \diamond s \rrbracket \Longrightarrow$

$s); \nabla s; \diamond s]] \Longrightarrow$   
 $ats \models_i \mathcal{BI} s \Longrightarrow insert\ a\ ats \models_i \mathcal{BI} (assert-bound\ a\ s)$   
**assumes** *assert-bound-nolhs-unsat*:  $[[\neg \mathcal{U} s; \models_{nolhs} s; \Delta (\mathcal{T} s); \nabla s; \diamond s]] \Longrightarrow$   
 $index-valid\ as\ s \Longrightarrow \mathcal{U} (assert-bound\ a\ s) \Longrightarrow minimal-unsat-state-core (assert-bound$   
 $a\ s)$   
**assumes** *assert-bound-nolhs-tableau-validated*:  $[[\neg \mathcal{U} s; \models_{nolhs} s; \Delta (\mathcal{T} s); \nabla s; \diamond$   
 $s]] \Longrightarrow$   
 $\nabla (assert-bound\ a\ s)$   
**assumes** *assert-bound-nolhs-index-valid*:  $[[\neg \mathcal{U} s; \models_{nolhs} s; \Delta (\mathcal{T} s); \nabla s; \diamond s]]$   
 $\Longrightarrow$   
 $index-valid\ as\ s \Longrightarrow index-valid (insert\ a\ as) (assert-bound\ a\ s)$

**sublocale** *AssertBoundNoLhs* < *AssertBound*  
*<proof>*

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

**locale** *Check* = **fixes** *check::('i,'a)::lrv) state  $\Rightarrow$  ('i,'a) state  
**assumes***

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

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

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

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

— The bounds remain unchanged.

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

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

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

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

**begin**



**lemma** *check-tableau-equiv*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noIhs}} s; \diamond s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow$   
 $(v::'a \text{ valuation}) \models_t \mathcal{T} s \longleftrightarrow v \models_t \mathcal{T} (\text{check } s)$   
 ⟨proof⟩

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

**lemma** *check-tableau-normalized*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noIhs}} s; \diamond s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \Delta (\mathcal{T} (\text{check } s))$   
 ⟨proof⟩

**lemma** *check-tableau-validated*:  $\llbracket \neg \mathcal{U} s; \models_{\text{noIhs}} s; \diamond s; \Delta (\mathcal{T} s); \nabla s \rrbracket \Longrightarrow \nabla (\text{check } s)$   
 ⟨proof⟩

**lemma** *check-indices-state*: **assumes**  $\neg \mathcal{U} s \Longrightarrow \models_{\text{noIhs}} s \neg \mathcal{U} s \Longrightarrow \diamond s \neg \mathcal{U} s$   
 $\Longrightarrow \Delta (\mathcal{T} s) \neg \mathcal{U} s \Longrightarrow \nabla s$   
**shows** *indices-state (check s) = indices-state s*  
 ⟨proof⟩

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

**end**

**locale** *Assert' = AssertBound assert-bound + Check check for*  
*assert-bound* ::  $('i, 'a::\text{Irv}) \text{ i-atom} \Rightarrow ('i, 'a) \text{ state} \Rightarrow ('i, 'a) \text{ state}$  **and**  
*check* ::  $('i, 'a::\text{Irv}) \text{ state} \Rightarrow ('i, 'a) \text{ state}$

**begin**

**definition** *assert* ::  $('i, 'a) \text{ i-atom} \Rightarrow ('i, 'a) \text{ state} \Rightarrow ('i, 'a) \text{ state}$  **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**

$\Delta (\mathcal{T} (\text{assert-bound } a \ s))$

$\neg \mathcal{U} (\text{assert-bound } a \ s) \Longrightarrow \models_{\text{noIhs}} (\text{assert-bound } a \ s) \wedge \diamond (\text{assert-bound } a \ s)$

$\nabla (\text{assert-bound } a \ s)$

⟨proof⟩

**end**

**sublocale** *Assert' < Assert assert*

*<proof>*

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

**locale** *AssertAllState''* = *Init init* + *AssertBoundNoLhs assert-bound* + *Check check for*

*init* :: *tableau*  $\Rightarrow$  (*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* postcondition must imply its precondition. This is not the case, since variables on the lhs may be out of their bounds. Therefore, we make a refinement and specify weaker preconditions (replace  $\models s$ , by  $\models_{\text{noLhs}} s$  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*  $\longrightarrow$   $\models_{\text{noLhs}} s \wedge \diamond s$

**shows** *let* *s'* = *assert-bound-loop* *ats* *s* *in*

$\Delta$  ( $\mathcal{T}$  *s'*)  $\wedge$   $\nabla$  *s'*  $\wedge$  ( $\neg$   $\mathcal{U}$  *s'*  $\longrightarrow$   $\models_{\text{noLhs}} s' \wedge \diamond s'$ )

*<proof>*

**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'*  $\longrightarrow$   $\models_{\text{noLhs}} s' \wedge \diamond s'$ )

*<proof>*

**lemma** *AssertAllState''Induct*:

**assumes**

$\Delta$  *t*

*P* {} (*init t*)

$\bigwedge$  *as bs t. as*  $\subseteq$  *bs*  $\implies$  *P as t*  $\implies$  *P bs t*

$\bigwedge$  *s a* *ats. [ $\neg$   $\mathcal{U}$  *s*;  $\langle \mathcal{V}$  *s*  $\rangle \models_t \mathcal{T}$  *s*;  $\models_{\text{noLhs}} s$ ;  $\Delta$  ( $\mathcal{T}$  *s*);  $\nabla$  *s*;  $\diamond$  *s*; *P* (*set* *ats*) *s*; *index-valid* (*set* *ats*) *s*]*

$\implies P$  (*insert a (set ats) (assert-bound a s)*)  
**shows**  $P$  (*set ats (assert-bound-loop ats (init t))*)  
 <proof>

**lemma** *AssertAllState''-tableau-id:*

$\Delta t \implies \mathcal{T}$  (*assert-bound-loop ats (init t)*) =  $\mathcal{T}$  (*init t*)  
 <proof>

**lemma** *AssertAllState''-sat:*

$\Delta t \implies$   
 $\neg \mathcal{U}$  (*assert-bound-loop ats (init t)*)  $\longrightarrow \models_{no\ l\ h\ s}$  (*assert-bound-loop ats (init t)*)  
 $\wedge \Diamond$  (*assert-bound-loop ats (init t)*)  
 <proof>

**lemma** *AssertAllState''-unsat:*

$\Delta t \implies \mathcal{U}$  (*assert-bound-loop ats (init t)*)  $\longrightarrow \text{minimal-unsat-state-core}$  (*assert-bound-loop ats (init t)*)  
 <proof>

**lemma** *AssertAllState''-sat-atoms-equiv-bounds:*

$\Delta t \implies \neg \mathcal{U}$  (*assert-bound-loop ats (init t)*)  $\longrightarrow \text{flat (set ats)} \doteq \mathcal{B}$  (*assert-bound-loop ats (init t)*)  
 <proof>

**lemma** *AssertAllState''-atoms-imply-bounds-index:*

**assumes**  $\Delta t$   
**shows** *set ats*  $\models_i \mathcal{BI}$  (*assert-bound-loop ats (init t)*)  
 <proof>

**lemma** *AssertAllState''-index-valid:*

$\Delta t \implies \text{index-valid (set ats) (assert-bound-loop ats (init t))}$   
 <proof>

**end**

**sublocale** *AssertAllState'' < AssertAllState assert-all-state*

<proof>

## 6.6 Update and Pivot

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

**locale** *Update* = **fixes** *update::var*  $\Rightarrow$  *'a::lrv*  $\Rightarrow$  (*'i,'a*) *state*  $\Rightarrow$  (*'i,'a*) *state*  
**assumes**

— Tableau, bounds, and the unsatisfiability flag are preserved.

*update-id*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x \notin \text{lvars} (\mathcal{T} s) \rrbracket \Longrightarrow$   
 $\text{let } s' = \text{update } x \text{ c s in } \mathcal{T} s' = \mathcal{T} s \wedge \mathcal{B}_i s' = \mathcal{B}_i s \wedge \mathcal{U} s' = \mathcal{U} s \wedge \mathcal{U}_c s' = \mathcal{U}_c s$   
**and**

— Tableau remains valuated.

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

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

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

— Updated valuation continues to satisfy the tableau.

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

**begin**

**lemma** *update-bounds-id*:

**assumes**  $\Delta (\mathcal{T} s) \nabla s x \notin \text{lvars} (\mathcal{T} s)$   
**shows**  $\mathcal{B}_i (\text{update } x \text{ c s}) = \mathcal{B}_i s$   
 $\mathcal{B}_l (\text{update } x \text{ c s}) = \mathcal{B}_l s$   
 $\mathcal{B}_u (\text{update } x \text{ c s}) = \mathcal{B}_u s$   
 $\langle \text{proof} \rangle$

**lemma** *update-indices-state-id*:

**assumes**  $\Delta (\mathcal{T} s) \nabla s x \notin \text{lvars} (\mathcal{T} s)$   
**shows**  $\text{indices-state } (\text{update } x \text{ c s}) = \text{indices-state } s$   
 $\langle \text{proof} \rangle$

**lemma** *update-tableau-id*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x \notin \text{lvars} (\mathcal{T} s) \rrbracket \Longrightarrow \mathcal{T} (\text{update } x \text{ c s}) =$   
 $\mathcal{T} s$   
 $\langle \text{proof} \rangle$

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

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

**definition** *assert-bound'* **where**

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

**fun**  $\text{assert-bound} :: ('i, 'a :: \text{lrV}) \ i\text{-atom} \Rightarrow ('i, 'a) \ \text{state} \Rightarrow ('i, 'a) \ \text{state}$  **where**  
 $\text{assert-bound} \ (i, \text{Leq } x \ c) \ s = \text{assert-bound}' \ \text{Positive } i \ x \ c \ s$   
 $|\ \text{assert-bound} \ (i, \text{Geq } x \ c) \ s = \text{assert-bound}' \ \text{Negative } i \ x \ c \ s$

**lemma**  $\text{assert-bound}'\text{-cases}$ :

**assumes**  $\llbracket \triangleright_{ub} \text{ (lt dir)} \ c \ ((\text{UB dir}) \ s \ x) \rrbracket \Longrightarrow P \ s$   
**assumes**  $\llbracket \neg (\triangleright_{ub} \text{ (lt dir)}) \ c \ ((\text{UB dir}) \ s \ x); \triangleleft_{lb} \text{ (lt dir)} \ c \ ((\text{LB dir}) \ s \ x) \rrbracket \Longrightarrow$   
 $P \ (\text{set-unsat } [i, ((\text{LI dir}) \ s \ x)] \ (\text{updateBI} \ (\text{UBI-upd dir}) \ i \ x \ c \ s))$   
**assumes**  $\llbracket x \notin \text{lvars} \ (\mathcal{T} \ s); \text{ (lt dir) } \ c \ (\langle \mathcal{V} \ s \rangle \ x); \neg (\triangleright_{ub} \text{ (lt dir)}) \ c \ ((\text{UB dir}) \ s \ x);$   
 $\neg (\triangleleft_{lb} \text{ (lt dir)}) \ c \ ((\text{LB dir}) \ s \ x) \rrbracket \Longrightarrow$   
 $P \ (\text{update } x \ c \ (\text{updateBI} \ (\text{UBI-upd dir}) \ i \ x \ c \ s))$   
**assumes**  $\llbracket \neg (\triangleright_{ub} \text{ (lt dir)}) \ c \ ((\text{UB dir}) \ s \ x); \neg (\triangleleft_{lb} \text{ (lt dir)}) \ c \ ((\text{LB dir}) \ s \ x); x$   
 $\in \text{lvars} \ (\mathcal{T} \ s) \rrbracket \Longrightarrow$   
 $P \ (\text{updateBI} \ (\text{UBI-upd dir}) \ i \ x \ c \ s)$   
**assumes**  $\llbracket \neg (\triangleright_{ub} \text{ (lt dir)}) \ c \ ((\text{UB dir}) \ s \ x); \neg (\triangleleft_{lb} \text{ (lt dir)}) \ c \ ((\text{LB dir}) \ s \ x); \neg$   
 $((\text{lt dir}) \ c \ (\langle \mathcal{V} \ s \rangle \ x)) \rrbracket \Longrightarrow$   
 $P \ (\text{updateBI} \ (\text{UBI-upd dir}) \ i \ x \ c \ s)$   
**assumes**  $\text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}$   
**shows**  $P \ (\text{assert-bound}' \ \text{dir} \ i \ x \ c \ s)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{assert-bound}\text{-cases}$ :

**assumes**  $\bigwedge \ c \ x \ \text{dir}.$   
 $\llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative};$   
 $a = \text{LE } \text{dir} \ x \ c;$   
 $\triangleright_{ub} \text{ (lt dir)} \ c \ ((\text{UB dir}) \ s \ x)$   
 $\rrbracket \Longrightarrow$   
 $P' \text{ (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd dir) (UI dir)}$   
 $(\text{LI dir}) \ (\text{LE dir}) \ (\text{GE dir}) \ s$   
**assumes**  $\bigwedge \ c \ x \ \text{dir}.$   
 $\llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative};$   
 $a = \text{LE } \text{dir} \ x \ c;$   
 $\neg \triangleright_{ub} \text{ (lt dir)} \ c \ ((\text{UB dir}) \ s \ x); \triangleleft_{lb} \text{ (lt dir)} \ c \ ((\text{LB dir}) \ s \ x)$   
 $\rrbracket \Longrightarrow$   
 $P' \text{ (lt dir) (UBI dir) (LBI dir) (UB dir) (LB dir) (UBI-upd dir) (UI dir)}$   
 $(\text{LI dir}) \ (\text{LE dir}) \ (\text{GE dir})$   
 $(\text{set-unsat } [i, ((\text{LI dir}) \ s \ x)] \ (\text{updateBI} \ (\text{UBI-upd dir}) \ i \ x \ c \ s))$   
**assumes**  $\bigwedge \ c \ x \ \text{dir}.$   
 $\llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative};$

$a = LE\ dir\ x\ c;$   
 $x \notin lvars\ (\mathcal{T}\ s); (lt\ dir)\ c\ (\langle \mathcal{V}\ s \rangle\ x);$   
 $\neg (\supset_{ub}\ (lt\ dir)\ c\ ((UB\ dir)\ s\ x)); \neg (\triangleleft_{lb}\ (lt\ dir)\ c\ ((LB\ dir)\ s\ x))$   
 $\Downarrow \implies$   
 $P' (lt\ dir)\ (UBI\ dir)\ (LBI\ dir)\ (UB\ dir)\ (LB\ dir)\ (UBI\text{-}upd\ dir)\ (UI\ dir)$   
 $(LI\ dir)\ (LE\ dir)\ (GE\ dir)$   
 $((update\ \mathcal{BI}\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s))$   
**assumes**  $\bigwedge c\ x\ dir.$   
 $\Downarrow\ dir = Positive \vee dir = Negative;$   
 $a = LE\ dir\ x\ c;$   
 $x \in lvars\ (\mathcal{T}\ s); \neg (\supset_{ub}\ (lt\ dir)\ c\ ((UB\ dir)\ s\ x));$   
 $\neg (\triangleleft_{lb}\ (lt\ dir)\ c\ ((LB\ dir)\ s\ x))$   
 $\Downarrow \implies$   
 $P' (lt\ dir)\ (UBI\ dir)\ (LBI\ dir)\ (UB\ dir)\ (LB\ dir)\ (UBI\text{-}upd\ dir)\ (UI\ dir)$   
 $(LI\ dir)\ (LE\ dir)\ (GE\ dir)$   
 $((update\ \mathcal{BI}\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s))$   
**assumes**  $\bigwedge c\ x\ dir.$   
 $\Downarrow\ dir = Positive \vee dir = Negative;$   
 $a = LE\ dir\ x\ c;$   
 $\neg (\supset_{ub}\ (lt\ dir)\ c\ ((UB\ dir)\ s\ x)); \neg (\triangleleft_{lb}\ (lt\ dir)\ c\ ((LB\ dir)\ s\ x));$   
 $\neg ((lt\ dir)\ c\ (\langle \mathcal{V}\ s \rangle\ x))$   
 $\Downarrow \implies$   
 $P' (lt\ dir)\ (UBI\ dir)\ (LBI\ dir)\ (UB\ dir)\ (LB\ dir)\ (UBI\text{-}upd\ dir)\ (UI\ dir)$   
 $(LI\ dir)\ (LE\ dir)\ (GE\ dir)$   
 $((update\ \mathcal{BI}\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s))$   
**assumes**  $\bigwedge s. P\ s = P' (>) \mathcal{B}_{il}\ \mathcal{B}_{iu}\ \mathcal{B}_l\ \mathcal{B}_u\ \mathcal{B}_{il\text{-}update}\ \mathcal{I}_l\ \mathcal{I}_u\ Geq\ Leq\ s$   
**assumes**  $\bigwedge s. P\ s = P' (<) \mathcal{B}_{iu}\ \mathcal{B}_{il}\ \mathcal{B}_u\ \mathcal{B}_l\ \mathcal{B}_{iu\text{-}update}\ \mathcal{I}_u\ \mathcal{I}_l\ Leq\ Geq\ s$   
**shows**  $P\ (assert\text{-}bound\ (i, a)\ s)$   
 $\langle proof \rangle$   
**end**

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

**lemma** *decrease-ub-satisfied-inverse*:

**assumes**  $lt: \triangleleft_{ub}\ (lt\ dir)\ c\ (UB\ dir\ s\ x)$  **and**  $dir: dir = Positive \vee dir = Negative$   
**assumes**  $v: v \models_b \mathcal{B}\ (update\ \mathcal{BI}\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s)$   
**shows**  $v \models_b \mathcal{B}\ s$   
 $\langle proof \rangle$

**lemma** *atoms-equiv-bounds-extend*:

**fixes**  $x\ c\ dir$   
**assumes**  $dir = Positive \vee dir = Negative$   $\neg \supset_{ub}\ (lt\ dir)\ c\ (UB\ dir\ s\ x)$   $ats \doteq$   
 $\mathcal{B}\ s$   
**shows**  $(ats \cup \{LE\ dir\ x\ c\}) \doteq \mathcal{B}\ (update\ \mathcal{BI}\ (UBI\text{-}upd\ dir)\ i\ x\ c\ s)$   
 $\langle proof \rangle$

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

**locale** *EqForLVar* =  
**fixes** *eq-idx-for-lvar* :: *tableau*  $\Rightarrow$  *var*  $\Rightarrow$  *nat*  
**assumes** *eq-idx-for-lvar*:  
 $\llbracket x \in \text{lvars } t \rrbracket \Longrightarrow \text{eq-idx-for-lvar } t \ x < \text{length } t \wedge \text{lhs } (t ! \text{eq-idx-for-lvar } t \ x) = x$   
**begin**  
**definition** *eq-for-lvar* :: *tableau*  $\Rightarrow$  *var*  $\Rightarrow$  *eq* **where**  
 $\text{eq-for-lvar } t \ v \equiv t ! (\text{eq-idx-for-lvar } t \ v)$   
**lemma** *eq-for-lvar*:  
 $\llbracket x \in \text{lvars } t \rrbracket \Longrightarrow \text{eq-for-lvar } t \ x \in \text{set } t \wedge \text{lhs } (\text{eq-for-lvar } t \ x) = x$   
 $\langle \text{proof} \rangle$

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

**lemma** *rvars-of-lvar-rvars*:  
**assumes**  $x \in \text{lvars } t$   
**shows**  $\text{rvars-of-lvar } t \ x \subseteq \text{rvars } t$   
 $\langle \text{proof} \rangle$

**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*:  $\langle v \rangle \models_e e \Longrightarrow \text{rhs-eq-val } v \ x \ c \ e = \text{rhs } e \ \llbracket \langle v \rangle (x := c) \rrbracket$

**begin**

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

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

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

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

**lemma** *update-no-left*:

**assumes**  $y \notin \text{lvars } t$

**shows**  $\text{look } (\mathcal{V} \ s) \ y = \text{look } (\text{foldl } (\text{update-eq } (\mathcal{V} \ s) \ x \ v) \ (\mathcal{V} \ s) \ t) \ y$

*<proof>*

**lemma** *update-left*:

**assumes**  $y \in \text{lvars } t$

**shows**  $\exists \ eq \in \text{set } t. \text{lhs } eq = y \wedge$

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

*<proof>*

**lemma** *update-valuate-rhs*:

**assumes**  $e \in \text{set } (\mathcal{T} \ s) \triangle (\mathcal{T} \ s)$

**shows**  $\text{rhs } e \llbracket \langle \mathcal{V} \ (\text{update } x \ c \ s) \rangle \rrbracket = \text{rhs } e \llbracket \langle \mathcal{V} \ s \rangle (x := c) \rrbracket$

*<proof>*

**end**

**sublocale** *RhsEqVal < Update update*

*<proof>*

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

**locale** *Pivot = EqForLVar + fixes*  $\text{pivot}::\text{var} \Rightarrow \text{var} \Rightarrow ('i, 'a::\text{lv}) \text{state} \Rightarrow ('i, 'a) \text{state}$

**assumes**

— Valuation, bounds, and the unsatisfiability flag are not changed.

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

$\text{let } s' = \text{pivot } x_i \ x_j \ s \ \text{in } \mathcal{V} \ s' = \mathcal{V} \ s \wedge \mathcal{B}_i \ s' = \mathcal{B}_i \ s \wedge \mathcal{U} \ s' = \mathcal{U} \ s \wedge \mathcal{U}_c \ s' = \mathcal{U}_c \ s \ \mathbf{and}$

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

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

$\text{let } s' = \text{pivot } x_i \ x_j \ s \ \text{in } ((v::'a \ \text{valuation}) \models_t \mathcal{T} \ s \longleftrightarrow v \models_t \mathcal{T} \ s') \wedge \Delta (\mathcal{T} \ s')$  **and**

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

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

$\text{rvars}(\mathcal{T} \ s') = \text{rvars}(\mathcal{T} \ s) - \{x_j\} \cup \{x_i\} \ \wedge \ \text{lvars}(\mathcal{T} \ s') = \text{lvars}(\mathcal{T} \ s) - \{x_i\} \cup \{x_j\}$



**begin**

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

**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**  $\mathcal{B}\mathcal{I} (\text{pivot } x_i x_j s) = \mathcal{B}\mathcal{I} s \mathcal{B} (\text{pivot } x_i x_j s) = \mathcal{B} s \mathcal{I} (\text{pivot } x_i x_j s) = \mathcal{I} s$   
*<proof>*

**lemma** *pivot-valuation-id*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket$   
 $\implies \mathcal{V} (\text{pivot } x_i x_j s) = \mathcal{V} s$   
*<proof>*

**lemma** *pivot-unsat-id*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket$   
 $\implies \mathcal{U} (\text{pivot } x_i x_j s) = \mathcal{U} s$   
*<proof>*

**lemma** *pivot-unsat-core-id*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket \implies \mathcal{U}_c (\text{pivot } x_i x_j s) = \mathcal{U}_c s$   
*<proof>*

**lemma** *pivot-tableau-equiv*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket \implies$   
 $(v::'a \text{ valuation}) \models_t \mathcal{T} s = v \models_t \mathcal{T} (\text{pivot } x_i x_j s)$   
*<proof>*

**lemma** *pivot-tableau-normalized*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket \implies \Delta (\mathcal{T} (\text{pivot } x_i x_j s))$   
*<proof>*

**lemma** *pivot-rvars*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket \implies$   
 $\text{rvars} (\mathcal{T} (\text{pivot } x_i x_j s)) = \text{rvars} (\mathcal{T} s) - \{x_j\} \cup \{x_i\}$   
*<proof>*

**lemma** *pivot-lvars*:  $\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket \implies$   
 $\text{lvars} (\mathcal{T} (\text{pivot } x_i x_j s)) = \text{lvars} (\mathcal{T} s) - \{x_i\} \cup \{x_j\}$   
*<proof>*

**lemma** *pivot-vars*:

$\llbracket \Delta (\mathcal{T} s); x_i \in \text{lvars} (\mathcal{T} s); x_j \in \text{rvars-of-lvar} (\mathcal{T} s) x_i \rrbracket \implies \text{tvars} (\mathcal{T} (\text{pivot } x_i x_j s)) = \text{tvars} (\mathcal{T} s)$   
*<proof>*

**lemma**

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

*<proof>*

**end**

Functions *pivot* and *update* can be used to implement the *check* function. In its context, *pivot* and *update* functions are always called together, so the following definition can be used: *pivot-and-update*  $x_i x_j c s = \text{update } x_i c (\text{pivot } x_i x_j s)$ . 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* ::  $var \Rightarrow 'a::lrv \Rightarrow ('i,'a) \text{ state} \Rightarrow ('i,'a) \text{ state}$   
**assumes** *pivotandupdate-unsat-id*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\mathcal{U} (\text{pivot-and-update } x_i x_j c s) = \mathcal{U} s$   
**assumes** *pivotandupdate-unsat-core-id*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\mathcal{U}_c (\text{pivot-and-update } x_i x_j c s) = \mathcal{U}_c s$   
**assumes** *pivotandupdate-bounds-id*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\mathcal{B}_i (\text{pivot-and-update } x_i x_j c s) = \mathcal{B}_i s$   
**assumes** *pivotandupdate-tableau-normalized*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\Delta (\mathcal{T} (\text{pivot-and-update } x_i x_j c s))$   
**assumes** *pivotandupdate-tableau-equiv*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $(v::'a \text{ valuation}) \models_t \mathcal{T} s \longleftrightarrow v \models_t \mathcal{T} (\text{pivot-and-update } x_i x_j c s)$   
**assumes** *pivotandupdate-satisfies-tableau*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\langle \mathcal{V} s \rangle \models_t \mathcal{T} s \longrightarrow \langle \mathcal{V} (\text{pivot-and-update } x_i x_j c s) \rangle \models_t \mathcal{T} s$   
**assumes** *pivotandupdate-rvars*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\text{rvars } (\mathcal{T} (\text{pivot-and-update } x_i x_j c s)) = \text{rvars } (\mathcal{T} s) - \{x_j\} \cup \{x_i\}$   
**assumes** *pivotandupdate-lvars*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\text{lvars } (\mathcal{T} (\text{pivot-and-update } x_i x_j c s)) = \text{lvars } (\mathcal{T} s) - \{x_i\} \cup \{x_j\}$   
**assumes** *pivotandupdate-valuation-nonlhs*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $x \notin \text{lvars } (\mathcal{T} s) - \{x_i\} \cup \{x_j\} \longrightarrow \text{look } (\mathcal{V} (\text{pivot-and-update } x_i x_j c s)) x =$   
*(if*  $x = x_i$  *then* *Some*  $c$  *else* *look*  $(\mathcal{V} s) x$ *)*  
**assumes** *pivotandupdate-tableau-validated*:  $\llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-of-lvar } (\mathcal{T} s) x_i \rrbracket \Longrightarrow$   
 $\nabla (\text{pivot-and-update } x_i x_j c s)$   
**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**  $\mathcal{BI} (\text{pivot-and-update } x_i \ x_j \ c \ s) = \mathcal{BI} \ s$   
 $\mathcal{B} (\text{pivot-and-update } x_i \ x_j \ c \ s) = \mathcal{B} \ s$   
 $\mathcal{I} (\text{pivot-and-update } x_i \ x_j \ c \ s) = \mathcal{I} \ s$   
 ⟨proof⟩

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

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

**lemma** *pivotandupdate-nolhs*:  
 $\llbracket \Delta (\mathcal{T} \ s); \nabla \ s; x_i \in \text{lvars } (\mathcal{T} \ s); x_j \in \text{rvars-of-lvar } (\mathcal{T} \ s) \ x_i; \models_{\text{nolhs}} \ s; \diamond \ s; \mathcal{B}_l \ s \ x_i = \text{Some } c \vee \mathcal{B}_u \ s \ x_i = \text{Some } c \rrbracket \implies \models_{\text{nolhs}} (\text{pivot-and-update } x_i \ x_j \ c \ s)$   
 ⟨proof⟩

**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**  $\diamond (\text{pivot-and-update } x_i \ x_j \ c \ s) = \diamond \ s$   
 ⟨proof⟩  
**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*  $x_i \ x_j \ c \ s \equiv \text{update } x_i \ c (\text{pivot } x_i \ x_j \ s)$

**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⟩

**end**

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

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

*assert-bound* (*Leq*  $x \ c$ )  $s \equiv$

if  $c \geq_{ub} \mathcal{B}_u s x$  then  $s$   
 else let  $s' = s \langle \mathcal{B}_u := (\mathcal{B}_u s) (x := \text{Some } c) \rangle$   
   in if  $c <_{lb} \mathcal{B}_l s x$  then  $s' \langle \mathcal{U} := \text{True} \rangle$   
   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  $\text{Geq } x c$  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*:  $\text{indices-state } (\text{set-unsat } I s) = \text{indices-state } s$   
 ⟨proof⟩

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

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

**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**  $(J, v) \models_{iae s} \text{as } v \models_t \mathcal{T} s$   
 ⟨proof⟩

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

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

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

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

**locale** *PivotEq* =  
**fixes** *pivot-eq*:  $\text{eq} \Rightarrow \text{var} \Rightarrow \text{eq}$   
**assumes**  
 — Lhs var of *eq* and  $x_j$  are swapped, while the other variables do not change sides.  
*vars-pivot-eq*:  
 $\llbracket x_j \in \text{rvars-eq } \text{eq}; \text{lhs } \text{eq} \notin \text{rvars-eq } \text{eq} \rrbracket \implies \text{let } \text{eq}' = \text{pivot-eq } \text{eq } x_j \text{ in}$   
 $\text{lhs } \text{eq}' = x_j \wedge \text{rvars-eq } \text{eq}' = \{\text{lhs } \text{eq}\} \cup (\text{rvars-eq } \text{eq} - \{x_j\})$  **and**

— Pivoting keeps the equation equisatisfiable.

*equiv-pivot-eq*:

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

**begin**

**lemma** *lhs-pivot-eq*:

$\llbracket x_j \in rvars\text{-}eq\ eq; lhs\ eq \notin rvars\text{-}eq\ eq \rrbracket \implies lhs\ (pivot\text{-}eq\ eq\ x_j) = x_j$   
 $\langle proof \rangle$

**lemma** *rvars-pivot-eq*:

$\llbracket x_j \in rvars\text{-}eq\ eq; lhs\ eq \notin rvars\text{-}eq\ eq \rrbracket \implies rvars\text{-}eq\ (pivot\text{-}eq\ eq\ x_j) = \{lhs\ eq\}$   
 $\cup (rvars\text{-}eq\ eq - \{x_j\})$   
 $\langle proof \rangle$

**end**

**abbreviation** *doublesub* ( -  $\subseteq$  s -  $\subseteq$  s - [50,51,51] 50) **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'*:

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

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

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

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

*equiv-subst-var*:

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

**begin**

**lemma** *vars-subst-var*:

$vars\ (subst\text{-}var\ x_j\ lp'\ lp) \subseteq (vars\ lp - \{x_j\}) \cup vars\ lp'$   
 $\langle proof \rangle$

**lemma** *vars-subst-var-supset*:

$vars\ (subst\text{-}var\ x_j\ lp'\ lp) \supseteq (vars\ lp - \{x_j\}) - vars\ lp'$   
 $\langle proof \rangle$

**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<sub>j</sub> lp eq)*  $\subseteq$  (*rvars-eq eq* - {*x<sub>j</sub>*})  $\cup$  *vars lp*  
 $\langle$ *proof* $\rangle$

**lemma** *rvars-eq-subst-var-eq-supset*:

*rvars-eq (subst-var-eq x<sub>j</sub> lp eq)*  $\supseteq$  (*rvars-eq eq*) - {*x<sub>j</sub>*} - (*vars lp*)  
 $\langle$ *proof* $\rangle$

**lemma** *equiv-subst-var-eq*:

**assumes** (*v::'a valuation*)  $\models_e$  (*x<sub>j</sub>*, *lp*)  
**shows** *v*  $\models_e$  *eq*  $\longleftrightarrow$  *v*  $\models_e$  *subst-var-eq x<sub>j</sub> lp' eq*  
 $\langle$ *proof* $\rangle$

**end**

**locale** *Pivot'* = *EqForLVar* + *PivotEq* + *SubstVar*

**begin**

**definition** *pivot-tableau'* :: *var*  $\Rightarrow$  *var*  $\Rightarrow$  *tableau*  $\Rightarrow$  *tableau* **where**

*pivot-tableau'* *x<sub>i</sub> x<sub>j</sub> t*  $\equiv$   
*let* *x<sub>i</sub>-idx* = *eq-idx-for-lvar t x<sub>i</sub>*; *eq* = *t ! x<sub>i</sub>-idx*; *eq'* = *pivot-eq eq x<sub>j</sub>* *in*  
*map* ( $\lambda$  *idx*. *if* *idx* = *x<sub>i</sub>-idx* *then*  
     *eq'*  
   *else*  
     *subst-var-eq x<sub>j</sub> (rhs eq') (t ! idx)*  
 ) [*0..<length t*]

**definition** *pivot'* :: *var*  $\Rightarrow$  *var*  $\Rightarrow$  (*'i,'a::lrv*) *state*  $\Rightarrow$  (*'i,'a*) *state* **where**  
*pivot'* *x<sub>i</sub> x<sub>j</sub> s*  $\equiv$   $\mathcal{T}$ -*update (pivot-tableau' x<sub>i</sub> x<sub>j</sub> ( $\mathcal{T}$  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<sub>i</sub> x<sub>j</sub> t*  $\equiv$  *let* *eq* = *eq-for-lvar t x<sub>i</sub>*; *eq'* = *pivot-eq eq x<sub>j</sub>* *in*  
*map* ( $\lambda$  *e*. *if* *lhs e* = *lhs eq* *then* *eq'* *else* *subst-var-eq x<sub>j</sub> (rhs eq') e*) *t*

**definition** *pivot* :: *var*  $\Rightarrow$  *var*  $\Rightarrow$  (*'i,'a::lrv*) *state*  $\Rightarrow$  (*'i,'a*) *state* **where**  
*pivot* *x<sub>i</sub> x<sub>j</sub> s*  $\equiv$   $\mathcal{T}$ -*update (pivot-tableau x<sub>i</sub> x<sub>j</sub> ( $\mathcal{T}$  s)) s*

**lemma** *pivot-tableau'pivot-tableau*:

**assumes**  $\Delta$  *t x<sub>i</sub>  $\in$  lvars t*  
**shows** *pivot-tableau' x<sub>i</sub> x<sub>j</sub> t* = *pivot-tableau x<sub>i</sub> x<sub>j</sub> t*  
 $\langle$ *proof* $\rangle$

**lemma** *pivot'pivot*: **fixes** *s* :: (*'i,'a::lrv*) *state*

**assumes**  $\Delta$  ( $\mathcal{T}$  *s*) *x<sub>i</sub>  $\in$  lvars ( $\mathcal{T}$  s)*  
**shows** *pivot' x<sub>i</sub> x<sub>j</sub> s* = *pivot x<sub>i</sub> x<sub>j</sub> s*  
 $\langle$ *proof* $\rangle$

**end**

**sublocale**  $Pivot' < Pivot$  eq-idx-for-lvar pivot  
{proof}

## 6.7 Check implementation

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

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

**assumes**

*min-lvar-not-in-bounds-None*: *min-lvar-not-in-bounds s = None*  $\longrightarrow$   $(\forall x \in \text{lvars } (\mathcal{T} s)). \text{in-bounds } x \langle \mathcal{V} s \rangle (\mathcal{B} s)$  **and**

*min-lvar-not-in-bounds-Some'*: *min-lvar-not-in-bounds s = Some x<sub>i</sub>*  $\longrightarrow$   $x_i \in \text{lvars } (\mathcal{T} s) \wedge \neg \text{in-bounds } x_i \langle \mathcal{V} s \rangle (\mathcal{B} s)$   
 $\wedge (\forall x \in \text{lvars } (\mathcal{T} s)). x < x_i \longrightarrow \text{in-bounds } x \langle \mathcal{V} s \rangle (\mathcal{B} s)$

**begin**

**lemma** *min-lvar-not-in-bounds-None'*:

*min-lvar-not-in-bounds s = None*  $\longrightarrow$   $(\langle \mathcal{V} s \rangle \models_b \mathcal{B} s \parallel \text{lvars } (\mathcal{T} s))$   
{proof}

**lemma** *min-lvar-not-in-bounds-lvars*: *min-lvar-not-in-bounds s = Some x<sub>i</sub>*  $\longrightarrow$   $x_i \in \text{lvars } (\mathcal{T} s)$   
{proof}

**lemma** *min-lvar-not-in-bounds-Some*: *min-lvar-not-in-bounds s = Some x<sub>i</sub>*  $\longrightarrow$   $\neg \text{in-bounds } x_i \langle \mathcal{V} s \rangle (\mathcal{B} s)$   
{proof}

**lemma** *min-lvar-not-in-bounds-Some-min*: *min-lvar-not-in-bounds*  $s = \text{Some } x_i$   
 $\longrightarrow (\forall x \in \text{lvars } (\mathcal{T} s). x < x_i \longrightarrow \text{in-bounds } x \langle \mathcal{V} s \rangle (\mathcal{B} s))$   
 $\langle \text{proof} \rangle$

**end**

**abbreviation** *reasable-var where*

*reasable-var*  $\text{dir } x \text{ eq } s \equiv$   
 $(\text{coeff } (\text{rhs eq}) x > 0 \wedge \triangleleft_{ub} (\text{lt dir}) (\langle \mathcal{V} s \rangle x) (\text{UB dir } s x)) \vee$   
 $(\text{coeff } (\text{rhs eq}) x < 0 \wedge \triangleright_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x) (\text{LB dir } s x))$

**locale** *MinRVarsEq* =

**fixes** *min-rvar-incdec-eq* ::  $('i, 'a) \text{ Direction} \Rightarrow ('i, 'a)::\text{lrv} \text{ state} \Rightarrow \text{eq} \Rightarrow 'i \text{ list} + \text{var}$

**assumes** *min-rvar-incdec-eq-None*:

*min-rvar-incdec-eq*  $\text{dir } s \text{ eq} = \text{Inl } is \Longrightarrow$

$(\forall x \in \text{rvars-eq eq}. \neg \text{reasable-var } \text{dir } x \text{ eq } s) \wedge$

$(\text{set } is = \{\text{LI dir } s (\text{lhs eq})\} \cup \{\text{LI dir } s x \mid x. x \in \text{rvars-eq eq} \wedge \text{coeff } (\text{rhs eq}) x < 0\})$

$\cup \{\text{UI dir } s x \mid x. x \in \text{rvars-eq eq} \wedge \text{coeff } (\text{rhs eq}) x > 0\}) \wedge$

$((\text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}) \longrightarrow \text{LI dir } s (\text{lhs eq}) \in \text{indices-state } s \longrightarrow$

$\text{set } is \subseteq \text{indices-state } s)$

**assumes** *min-rvar-incdec-eq-Some-rvars*:

*min-rvar-incdec-eq*  $\text{dir } s \text{ eq} = \text{Inr } x_j \Longrightarrow x_j \in \text{rvars-eq eq}$

**assumes** *min-rvar-incdec-eq-Some-incdec*:

*min-rvar-incdec-eq*  $\text{dir } s \text{ eq} = \text{Inr } x_j \Longrightarrow \text{reasable-var } \text{dir } x_j \text{ eq } s$

**assumes** *min-rvar-incdec-eq-Some-min*:

*min-rvar-incdec-eq*  $\text{dir } s \text{ eq} = \text{Inr } x_j \Longrightarrow$

$(\forall x \in \text{rvars-eq eq}. x < x_j \longrightarrow \neg \text{reasable-var } \text{dir } x \text{ eq } s)$

**begin**

**lemma** *min-rvar-incdec-eq-None'*:

**assumes** \*:  $\text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}$

**and** *min*: *min-rvar-incdec-eq*  $\text{dir } s \text{ eq} = \text{Inl } is$

**and** *sub*:  $I = \text{set } is$

**and** *Iv*:  $(I, v) \models_{ib} \mathcal{BI} s$

**shows**  $le (\text{lt dir}) ((\text{rhs eq}) \{\!\{v\}\!\}) ((\text{rhs eq}) \{\!\{\langle \mathcal{V} s \rangle\}\!\})$

$\langle \text{proof} \rangle$

**end**

**locale** *MinRVars* = *EqForLVar* + *MinRVarsEq* *min-rvar-incdec-eq*

**for** *min-rvar-incdec-eq* ::  $('i, 'a)::\text{lrv} \text{ Direction} \Rightarrow -$

**begin**

**abbreviation** *min-rvar-incdec* ::  $('i, 'a) \text{ Direction} \Rightarrow ('i, 'a) \text{ state} \Rightarrow \text{var} \Rightarrow 'i \text{ list}$

+ *var where*

*min-rvar-incdec*  $\text{dir } s x_i \equiv \text{min-rvar-incdec-eq } \text{dir } s (\text{eq-for-lvar } (\mathcal{T} s) x_i)$

**end**



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

**locale** *PivotUpdateMinVars* =  
*PivotAndUpdate* *eq-idx-for-lvar* *pivot-and-update* +  
*MinVars* *min-lvar-not-in-bounds* *eq-idx-for-lvar* *min-rvar-incdec-eq* **for**  
*eq-idx-for-lvar* :: *tableau* ⇒ *var* ⇒ *nat* **and**  
*min-lvar-not-in-bounds* :: ('i,'a::lrv) *state* ⇒ *var option* **and**  
*min-rvar-incdec-eq* :: ('i,'a) *Direction* ⇒ ('i,'a) *state* ⇒ *eq* ⇒ 'i *list* + *var* **and**  
*pivot-and-update* :: *var* ⇒ *var* ⇒ 'a ⇒ ('i,'a) *state* ⇒ ('i,'a) *state*  
**begin**

**definition** *check'* **where**

*check' dir x<sub>i</sub> s* ≡  
*let* *l<sub>i</sub>* = *the (LB dir s x<sub>i</sub>)*;  
*x<sub>j</sub>'* = *min-rvar-incdec dir s x<sub>i</sub>*  
*in case* *x<sub>j</sub>'* *of*  
*Inl I* ⇒ *set-unsat I s*  
| *Inr x<sub>j</sub>* ⇒ *pivot-and-update x<sub>i</sub> x<sub>j</sub> l<sub>i</sub> s*

**lemma** *check'-cases*:

**assumes**  $\bigwedge I. \llbracket \text{min-rvar-incdec dir s } x_i = \text{Inl } I; \text{check}' \text{ dir } x_i \text{ s} = \text{set-unsat } I \text{ s} \rrbracket$   
 $\implies P (\text{set-unsat } I \text{ s})$   
**assumes**  $\bigwedge x_j \ l_i. \llbracket \text{min-rvar-incdec dir s } x_i = \text{Inr } x_j; \$   
 $l_i = \text{the } (LB \text{ dir } s \ x_i);$   
 $\text{check}' \text{ dir } x_i \text{ s} = \text{pivot-and-update } x_i \ x_j \ l_i \ \text{s} \rrbracket \implies$   
 $P (\text{pivot-and-update } x_i \ x_j \ l_i \ \text{s})$   
**shows**  $P (\text{check}' \text{ dir } x_i \ \text{s})$   
*<proof>*

**partial-function** (*tailrec*) *check* **where**

*check s* =  
*(if*  $\mathcal{U} \ \text{s}$  *then* *s*  
*else let*  $x_i' = \text{min-lvar-not-in-bounds } \text{s}$   
*in case*  $x_i'$  *of*  
*None* ⇒ *s*  
| *Some*  $x_i$  ⇒ *let* *dir* = *if*  $\langle \mathcal{V} \ \text{s} \rangle \ x_i <_{lb} \ \mathcal{B}_l \ \text{s} \ x_i$  *then* *Positive*  
*else* *Negative*  
*in* *check* (*check' dir x<sub>i</sub> s*)

**declare** *check.simps*[*code*]

**inductive** *check-dom* **where**

*step*:  $\llbracket \bigwedge x_i. \llbracket \neg \mathcal{U} \ \text{s}; \text{Some } x_i = \text{min-lvar-not-in-bounds } \text{s}; \langle \mathcal{V} \ \text{s} \rangle \ x_i <_{lb} \ \mathcal{B}_l \ \text{s} \ x_i \rrbracket$

$$\begin{aligned}
&\implies \text{check-dom } (\text{check}' \text{ Positive } x_i \ s); \\
&\bigwedge x_i. \llbracket \neg \mathcal{U} \ s; \text{Some } x_i = \text{min-lvar-not-in-bounds } s; \neg \langle \mathcal{V} \ s \rangle \ x_i <_{lb} \mathcal{B}_l \ s \ x_i \rrbracket \\
&\implies \text{check-dom } (\text{check}' \text{ Negative } x_i \ s) \\
&\implies \text{check-dom } s
\end{aligned}$$

The definition of *check* can be given by:

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

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

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

**lemma** *pivotandupdate-check-precond*:

**assumes**

*dir* = (if  $\langle \mathcal{V} \ s \rangle \ x_i <_{lb} \mathcal{B}_l \ s \ x_i$  then *Positive* else *Negative*)

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

*min-rvar-incdec* *dir* *s*  $x_i$  = *Inr*  $x_j$

$l_i$  = *the* (*LB* *dir* *s*  $x_i$ )

$\nabla \ s \ \Delta \ (\mathcal{T} \ s) \models_{\text{noIhs}} \ s \ \diamond \ s$

**shows**  $\Delta \ (\mathcal{T} \ (\text{pivot-and-update } x_i \ x_j \ l_i \ s)) \wedge \models_{\text{noIhs}} \ (\text{pivot-and-update } x_i \ x_j \ l_i \ s) \wedge \diamond \ (\text{pivot-and-update } x_i \ x_j \ l_i \ s) \wedge \nabla \ (\text{pivot-and-update } x_i \ x_j \ l_i \ s)$   
*<proof>*

**abbreviation** *gt-state'* **where**

*gt-state'* *dir* *s*  $s' \ x_i \ x_j \ l_i \equiv$

*min-lvar-not-in-bounds* *s* = *Some*  $x_i \ \wedge$

$l_i$  = *the* (*LB* *dir* *s*  $x_i$ )  $\wedge$

*min-rvar-incdec* *dir* *s*  $x_i$  = *Inr*  $x_j \ \wedge$

$s' = \text{pivot-and-update } x_i \ x_j \ l_i \ s$

**definition** *gt-state* :: (*i*,*'a*) *state*  $\Rightarrow$  (*i*,*'a*) *state*  $\Rightarrow$  *bool* (**infixl**  $\succ_x$  100) **where**

$s \succ_x s' \equiv$

$\exists \ x_i \ x_j \ l_i.$

let  $dir = \text{if } \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_i s x_i \text{ then Positive else Negative in}$   
 $gt\text{-state}' dir s s' x_i x_j l_i$

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

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

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

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

**lemma**  $succ\text{-vars}$ :

**assumes**  $s \succ s'$

**obtains**  $x_i x_j$  **where**

$x_i \in lvars (\mathcal{T} s)$

$x_j \in rvars\text{-of}\text{-lvar} (\mathcal{T} s) x_i x_j \in rvars (\mathcal{T} s)$

$lvars (\mathcal{T} s') = lvars (\mathcal{T} s) - \{x_i\} \cup \{x_j\}$

$rvars (\mathcal{T} s') = rvars (\mathcal{T} s) - \{x_j\} \cup \{x_i\}$

$\langle proof \rangle$

**lemma**  $succ\text{-vars}\text{-id}$ :

**assumes**  $s \succ s'$

**shows**  $lvars (\mathcal{T} s) \cup rvars (\mathcal{T} s) =$

$lvars (\mathcal{T} s') \cup rvars (\mathcal{T} s')$

$\langle proof \rangle$

**lemma**  $succ\text{-inv}$ :

**assumes**  $s \succ s'$

**shows**  $\Delta (\mathcal{T} s') \nabla s' \Diamond s' \mathcal{B}_i s = \mathcal{B}_i s'$

$(v::'a \text{ valuation}) \models_t (\mathcal{T} s) \longleftrightarrow v \models_t (\mathcal{T} s')$

$\langle proof \rangle$

**lemma**  $succ\text{-rvar}\text{-valuation}\text{-id}$ :

**assumes**  $s \succ s' x \in rvars (\mathcal{T} s) x \in rvars (\mathcal{T} s')$

**shows**  $\langle \mathcal{V} s \rangle x = \langle \mathcal{V} s' \rangle x$

$\langle proof \rangle$

**lemma**  $succ\text{-min}\text{-lvar}\text{-not}\text{-in}\text{-bounds}$ :

**assumes**  $s \succ s'$

$xr \in lvars (\mathcal{T} s) xr \in rvars (\mathcal{T} s')$

**shows**  $\neg \text{in}\text{-bounds } xr ((\mathcal{V} s)) (\mathcal{B} s)$

$\forall x \in lvars (\mathcal{T} s). x < xr \longrightarrow \text{in}\text{-bounds } x ((\mathcal{V} s)) (\mathcal{B} s)$

$\langle proof \rangle$

**lemma** *succ-min-rvar*:

**assumes**  $s \succ s'$

$xs \in lvars (\mathcal{T} s)$   $xs \in rvars (\mathcal{T} s')$

$xr \in rvars (\mathcal{T} s)$   $xr \in lvars (\mathcal{T} s')$

$eq = eq\text{-for-lvar } (\mathcal{T} s) xs$  **and**

$dir: dir = Positive \vee dir = Negative$

**shows**

$\neg \triangleright_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle xs) (LB\ dir\ s\ xs) \longrightarrow$

$reasable\text{-var } dir\ xr\ eq\ s \wedge (\forall x \in rvars\text{-eq } eq. x < xr \longrightarrow \neg reasable\text{-var}$

$dir\ x\ eq\ s)$

$\langle proof \rangle$

**lemma** *succ-set-on-bound*:

**assumes**

$s \succ s'$   $x_i \in lvars (\mathcal{T} s)$   $x_i \in rvars (\mathcal{T} s')$  **and**

$dir: dir = Positive \vee dir = Negative$

**shows**

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

$\langle \mathcal{V} s' \rangle x_i = the (\mathcal{B}_l\ s\ x_i) \vee \langle \mathcal{V} s' \rangle x_i = the (\mathcal{B}_u\ s\ x_i)$

$\langle proof \rangle$

**lemma** *succ-rvar-valuation*:

**assumes**

$s \succ s'$   $x \in rvars (\mathcal{T} s')$

**shows**

$\langle \mathcal{V} s' \rangle x = \langle \mathcal{V} s \rangle x \vee \langle \mathcal{V} s' \rangle x = the (\mathcal{B}_l\ s\ x) \vee \langle \mathcal{V} s' \rangle x = the (\mathcal{B}_u\ s\ x)$

$\langle proof \rangle$

**lemma** *succ-no-vars-valuation*:

**assumes**

$s \succ s'$   $x \notin tvars (\mathcal{T} s')$

**shows**  $look (\mathcal{V} s') x = look (\mathcal{V} s) x$

$\langle proof \rangle$

**lemma** *succ-valuation-satisfies*:

**assumes**  $s \succ s'$   $\langle \mathcal{V} s \rangle \models_t \mathcal{T} s$

**shows**  $\langle \mathcal{V} s' \rangle \models_t \mathcal{T} s'$

$\langle proof \rangle$

**lemma** *succ-tableau-validated*:

**assumes**  $s \succ s'$   $\nabla s$

**shows**  $\nabla s'$

$\langle proof \rangle$

**abbreviation** *succ-chain where*

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

**lemma** *succ-chain-induct*:

**assumes** \*: *succ-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*:  $\bigwedge i\ j\ k. \llbracket P\ i\ j; P\ j\ k; i < j; j \leq k \rrbracket \implies P\ i\ k$   
**shows**  $P\ i\ j$   
 $\langle \text{proof} \rangle$

**lemma** *succ-chain-bounds-id*:

**assumes** *succ-chain*  $l\ i \leq j\ j < \text{length } l$   
**shows**  $\mathcal{B}_i\ (l!\ i) = \mathcal{B}_i\ (l!\ j)$   
 $\langle \text{proof} \rangle$

**lemma** *succ-chain-vars-id'*:

**assumes** *succ-chain*  $l\ i \leq j\ j < \text{length } l$   
**shows**  $lvars\ (\mathcal{T}\ (l!\ i)) \cup rvars\ (\mathcal{T}\ (l!\ i)) =$   
 $lvars\ (\mathcal{T}\ (l!\ j)) \cup rvars\ (\mathcal{T}\ (l!\ j))$   
 $\langle \text{proof} \rangle$

**lemma** *succ-chain-vars-id*:

**assumes** *succ-chain*  $l\ i < \text{length } l\ j < \text{length } l$   
**shows**  $lvars\ (\mathcal{T}\ (l!\ i)) \cup rvars\ (\mathcal{T}\ (l!\ i)) =$   
 $lvars\ (\mathcal{T}\ (l!\ j)) \cup rvars\ (\mathcal{T}\ (l!\ j))$   
 $\langle \text{proof} \rangle$

**lemma** *succ-chain-tableau-equiv'*:

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

**lemma** *succ-chain-tableau-equiv*:

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

**lemma** *succ-chain-no-vars-valuation*:

**assumes** *succ-chain*  $l\ i \leq j\ j < \text{length } l$   
**shows**  $\forall x. x \notin tvars\ (\mathcal{T}\ (l!\ i)) \longrightarrow \text{look}\ (\mathcal{V}\ (l!\ i))\ x = \text{look}\ (\mathcal{V}\ (l!\ j))\ x\ (\text{is } ?P\ i\ j)$   
 $\langle \text{proof} \rangle$

**lemma** *succ-chain-rvar-valuation*:

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

**lemma** *succ-chain-valuation-satisfies:*

**assumes** *succ-chain*  $l \ i \leq j \ j < \text{length } l$

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

*<proof>*

**lemma** *succ-chain-tableau-validated:*

**assumes** *succ-chain*  $l \ i \leq j \ j < \text{length } l$

**shows**  $\nabla (l! i) \longrightarrow \nabla (l! j)$

*<proof>*

**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*  $i \ j$ )

*<proof>*

**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**  $\exists k. \ i \leq k \wedge k < j \wedge \text{swap-rl } l \ k \ x$

*<proof>*

**lemma** *succ-chain-swap-lr-exists:*

**assumes** *succ-chain*  $l \ i < j \ j < \text{length } l$

$x \in \text{lvars } (\mathcal{T} (l! i)) \ x \in \text{rvars } (\mathcal{T} (l! j))$

**shows**  $\exists k. \ i \leq k \wedge k < j \wedge \text{swap-lr } l \ k \ x$

*<proof>*

**lemma** *finite-tableaus-aux:*

**shows** *finite*  $\{t. \text{lvars } t = L \wedge \text{rvars } t = V - L \wedge \Delta t \wedge (\forall v. \text{'a valuation. } v \models_t t = v \models_t t0)\}$  (**is** *finite* (*?Al*  $L$ ))

*<proof>*

**lemma** *finite-tableaus:*

**assumes** *finite*  $V$

**shows** *finite*  $\{t. \text{tvars } t = V \wedge \Delta t \wedge (\forall v. \text{'a valuation. } v \models_t t = v \models_t t0)\}$  (**is**

*finite ?A*  
*<proof>*

**lemma** *finite-accessible-tableaus*:  
**shows** *finite* ( $\mathcal{T}$  ‘  $\{s'. s \succ^* s'\}$  )  
*<proof>*

**abbreviation** *check-valuation where*

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

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

**lemma** *finite-accessible-valuations*:  
**shows** *finite* ( $\mathcal{V}$  ‘  $\{s'. s \succ^* s'\}$  )  
*<proof>*

**lemma** *accessible-bounds*:  
**shows**  $\mathcal{B}_i$  ‘  $\{s'. s \succ^* s'\} = \{\mathcal{B}_i\ s\}$   
*<proof>*

**lemma** *accessible-unsat-core*:  
**shows**  $\mathcal{U}_c$  ‘  $\{s'. s \succ^* s'\} = \{\mathcal{U}_c\ s\}$   
*<proof>*

**lemma** *state-eqI*:  
 $\mathcal{B}_{il}\ s = \mathcal{B}_{il}\ s' \implies \mathcal{B}_{iu}\ s = \mathcal{B}_{iu}\ s' \implies$   
 $\mathcal{T}\ s = \mathcal{T}\ s' \implies \mathcal{V}\ s = \mathcal{V}\ s' \implies$   
 $\mathcal{U}\ s = \mathcal{U}\ s' \implies \mathcal{U}_c\ s = \mathcal{U}_c\ s' \implies$   
 $s = s'$   
*<proof>*

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

**lemma** *acyclic-suc-rel*: *acyclic succ-rel*  
*<proof>*

**lemma** *check-unsat-terminates*:

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

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

**assumes**  
 $dir: dir = (if \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i \text{ then Positive else Negative})$  **and**  
 $*$ :  $\bigwedge s'. \llbracket s \succ s'; \nabla s'; \Delta (\mathcal{T} s'); \diamond s'; \models_{noth_s} s' \rrbracket \implies check\text{-}dom\ s'$  **and**  
 $\nabla s \Delta (\mathcal{T} s) \diamond s \models_{noth_s} s$   
 $\neg \mathcal{U} s \text{ min-lvar-not-in-bounds } s = \text{Some } x_i$   
 $\triangleleft_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i)$   
**shows** *check-dom*  
 (case *min-rvar-incdec dir s x\_i of Inl I*  $\implies$  *set-unsat I s*  
 | *Inr x\_j*  $\implies$  *pivot-and-update x\_i x\_j (the (LB dir s x\_i)) s*)  
 ⟨*proof*⟩

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

**assumes**  $\nabla s \Delta (\mathcal{T} s) \diamond s \models_{noth_s} s\ s_0 \succ^* s$   
**shows** *check-dom s*  
 ⟨*proof*⟩

**lemma** *check-sat-terminates:*

**assumes**  $\nabla s \Delta (\mathcal{T} s) \diamond s \models_{noth_s} s$   
**shows** *check-dom s*  
 ⟨*proof*⟩

**lemma** *check-cases:*

**assumes**  $\mathcal{U} s \implies P s$   
**assumes**  $\llbracket \neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{None} \rrbracket \implies P s$   
**assumes**  $\bigwedge x_i \text{ dir } I.$   
 $\llbracket dir = \text{Positive} \vee dir = \text{Negative};$   
 $\neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{Some } x_i;$   
 $\triangleleft_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i);$   
 $\text{min-rvar-incdec } dir\ s\ x_i = \text{Inl } I \rrbracket \implies$   
 $P (\text{set-unsat } I\ s)$   
**assumes**  $\bigwedge x_i x_j l_i \text{ dir}.$   
 $\llbracket dir = (if \langle \mathcal{V} s \rangle x_i <_{lb} \mathcal{B}_l s x_i \text{ then Positive else Negative});$   
 $\neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{Some } x_i;$   
 $\triangleleft_{lb} (lt\ dir) (\langle \mathcal{V} s \rangle x_i) (LB\ dir\ s\ x_i);$   
 $\text{min-rvar-incdec } dir\ s\ x_i = \text{Inr } x_j;$   
 $l_i = \text{the } (LB\ dir\ s\ x_i);$   
 $check'\ dir\ x_i\ s = \text{pivot-and-update } x_i\ x_j\ l_i\ s \rrbracket \implies$   
 $P (\text{check } (\text{pivot-and-update } x_i\ x_j\ l_i\ s))$   
**assumes**  $\Delta (\mathcal{T} s) \diamond s \models_{noth_s} s$   
**shows**  $P (\text{check } s)$   
 ⟨*proof*⟩



**lemma** *check-induct*:

**fixes**  $s :: ('i, 'a) \text{ state}$

**assumes**  $*$ :  $\nabla s \Delta (\mathcal{T} s) \models_{\text{no}hs} s \diamond s$

**assumes**  $**$ :

$\bigwedge s. \mathcal{U} s \implies P s s$

$\bigwedge s. \llbracket \neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{None} \rrbracket \implies P s s$

$\bigwedge s x_i \text{ dir } I. \llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}; \neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{Some } x_i;$

$\triangleleft_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x_i) (\text{LB dir } s x_i); \text{min-rvar-incdec dir } s x_i = \text{Inl } I \rrbracket$

$\implies P s (\text{set-unsat } I s)$

**assumes** *step'*:  $\bigwedge s x_i x_j l_i. \llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-eq } (\text{eq-for-lvar } (\mathcal{T} s) x_i) \rrbracket \implies P s (\text{pivot-and-update } x_i x_j l_i s)$

**assumes** *trans'*:  $\bigwedge si sj sk. \llbracket P si sj; P sj sk \rrbracket \implies P si sk$

**shows**  $P s (\text{check } s)$

*<proof>*

**lemma** *check-induct'*:

**fixes**  $s :: ('i, 'a) \text{ state}$

**assumes**  $\nabla s \Delta (\mathcal{T} s) \models_{\text{no}hs} s \diamond s$

**assumes**  $\bigwedge s x_i \text{ dir } I. \llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}; \neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{Some } x_i;$

$\triangleleft_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x_i) (\text{LB dir } s x_i); \text{min-rvar-incdec dir } s x_i = \text{Inl } I; P s \rrbracket$

$\implies P (\text{set-unsat } I s)$

**assumes**  $\bigwedge s x_i x_j l_i. \llbracket \Delta (\mathcal{T} s); \nabla s; x_i \in \text{lvars } (\mathcal{T} s); x_j \in \text{rvars-eq } (\text{eq-for-lvar } (\mathcal{T} s) x_i); P s \rrbracket \implies P (\text{pivot-and-update } x_i x_j l_i s)$

**assumes**  $P s$

**shows**  $P (\text{check } s)$

*<proof>*

**lemma** *check-induct''*:

**fixes**  $s :: ('i, 'a) \text{ state}$

**assumes**  $*$ :  $\nabla s \Delta (\mathcal{T} s) \models_{\text{no}hs} s \diamond s$

**assumes**  $**$ :

$\mathcal{U} s \implies P s$

$\bigwedge s. \llbracket \nabla s; \Delta (\mathcal{T} s); \models_{\text{no}hs} s; \diamond s; \neg \mathcal{U} s; \text{min-lvar-not-in-bounds } s = \text{None} \rrbracket \implies P s$

$\bigwedge s x_i \text{ dir } I. \llbracket \text{dir} = \text{Positive} \vee \text{dir} = \text{Negative}; \nabla s; \Delta (\mathcal{T} s); \models_{\text{no}hs} s; \diamond s; \neg \mathcal{U} s;$

$\text{min-lvar-not-in-bounds } s = \text{Some } x_i; \triangleleft_{lb} (\text{lt dir}) (\langle \mathcal{V} s \rangle x_i) (\text{LB dir } s x_i);$

$\text{min-rvar-incdec dir } s x_i = \text{Inl } I \rrbracket$

$\implies P (\text{set-unsat } I s)$

**shows**  $P (\text{check } s)$

*<proof>*

**end**

**lemma** *poly-eval-update*:  $(p \llbracket v (x := c :: 'a :: lrv) \rrbracket) = (p \llbracket v \rrbracket) + \text{coeff } p x * R$

$(c - v x)$   
 $\langle proof \rangle$

**context** *PivotUpdateMinVars*

**begin**

**context**

**fixes** *rhs-eq-val* ::  $(var, 'a::lrv) \text{ mapping} \Rightarrow var \Rightarrow 'a \Rightarrow eq \Rightarrow 'a$

**assumes** *RhsEqVal rhs-eq-val*

**begin**

**lemma** *check-minimal-unsat-state-core*:

**assumes** \*:  $\neg \mathcal{U} s \models_{nohs} s \diamond s \Delta (\mathcal{T} s) \nabla s$

**shows**  $\mathcal{U} (check\ s) \longrightarrow \text{minimal-unsat-state-core} (check\ s)$

**(is** *?P*  $(check\ s)$ )

$\langle proof \rangle$

**lemma** *Check-check: Check check*

$\langle proof \rangle$

**end**

**end**

## 6.8 Symmetries

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

## 6.9 Concrete implementation

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

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

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

**definition** *min-lvar-not-in-bounds* :: ('i,'a::{linorder,zero}) state  $\Rightarrow$  var option  
**where**  
*min-lvar-not-in-bounds* s  $\equiv$   
*min-satisfying* ( $\lambda$  x.  $\neg$  in-bounds x ( $\langle \mathcal{V} \rangle$  s)) ( $\mathcal{B}$  s) (map lhs ( $\mathcal{T}$  s))

**interpretation** *MinLVarNotInBounds min-lvar-not-in-bounds* :: ('i,'a::lrv) state  
 $\Rightarrow$  -  
 ⟨proof⟩

**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. reasable-var dir x eq s) rvars of  
 None  $\Rightarrow$  Inl (*unsat-indices* dir s rvars eq)  
 | Some  $x_j$   $\Rightarrow$  Inr  $x_j$ )

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

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

**definition** *eq-idx-for-lvar* **where**  
*eq-idx-for-lvar* t x  $\equiv$  *eq-idx-for-lvar-aux* t x 0

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

**global-interpretation** *EqForLVarDefault*: *EqForLVar eq-idx-for-lvar*  
**defines** *eq-for-lvar-code* = *EqForLVarDefault.eq-for-lvar*  
 $\langle \text{proof} \rangle$

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

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

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

**interpretation** *PivotEq pivot-eq*  
 $\langle \text{proof} \rangle$

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

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

**global-interpretation** *SubstVar subst-var rewrites*  
 $\text{SubstVar.subst-var-eq } \text{subst-var} = \text{subst-var-eq-code}$   
 $\langle \text{proof} \rangle$

**definition** *rhs-eq-val* **where**

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

**definition** *update-code* = *RhsEqVal.update rhs-eq-val*

**definition** *assert-bound'-code* = *Update.assert-bound' update-code*

**definition** *assert-bound-code* = *Update.assert-bound update-code*

**global-interpretation** *RhsEqValDefault'*: *RhsEqVal rhs-eq-val*

**rewrites**

*RhsEqVal.update rhs-eq-val* = *update-code* **and**

*Update.assert-bound update-code* = *assert-bound-code* **and**

*Update.assert-bound' update-code* = *assert-bound'-code*

*<proof>*

**sublocale** *PivotUpdateMinVars* < *Check check*

*<proof>*

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

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

**global-interpretation** *Pivot'Default*: *Pivot' eq-idx-for-lvar pivot-eq subst-var*

**rewrites**

*Pivot'.pivot eq-idx-for-lvar pivot-eq subst-var* = *pivot-code* **and**

*Pivot'.pivot-tableau eq-idx-for-lvar pivot-eq subst-var* = *pivot-tableau-code* **and**

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

*<proof>*

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

*<proof>*

**sublocale** *Update* < *AssertBoundNoLhs assert-bound*

*<proof>*

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

**definition** *check'-code* = *PivotUpdateMinVars.check' eq-idx-for-lvar min-rvar-incdec-eq pivot-and-update-code*

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

**rewrites**

*PivotUpdateMinVars.check eq-idx-for-lvar min-lvar-not-in-bounds min-rvar-incdec-eq*

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

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

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

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

**definition** *assert-all-code* = *AssertAllState.assert-all assert-all-state-code*

**global-interpretation** *AssertAllStateDefault: AssertAllState'' init-state assert-bound-code*  
*check-code*

**rewrites**  
*AssertAllState''.assert-bound-loop assert-bound-code* = *assert-bound-loop-code*  
**and**  
*AssertAllState''.assert-all-state init-state assert-bound-code check-code* = *assert-all-state-code* **and**  
*AssertAllState.assert-all assert-all-state-code* = *assert-all-code*  
⟨*proof*⟩

**primrec**

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

**primrec**

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

**primrec**

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

**fun** *linear-poly-to-eq* :: *linear-poly*  $\Rightarrow$  *var*  $\Rightarrow$  *eq* **where**

*linear-poly-to-eq p v* = (*v, p*)

**datatype** *'i istate* = *IState*

(*FirstFreshVariable*: *var*)

(*Tableau*: *tableau*)

(*Atoms*: (*'i, QDelta*) *i-atom list*)

(*Poly-Mapping*: *linear-poly*  $\rightarrow$  *var*)

(*UnsatIndices*: *'i list*)

**primrec** *zero-satisfies* :: *'a* :: *lrv ns-constraint*  $\Rightarrow$  *bool* **where**

*zero-satisfies* (*LEQ-ns l r*)  $\longleftrightarrow$   $0 \leq r$

| *zero-satisfies* (*GEQ-ns l r*)  $\longleftrightarrow$   $0 \geq r$

**lemma** *zero-satisfies*: *poly c = 0*  $\Longrightarrow$  *zero-satisfies c*  $\Longrightarrow$  *v*  $\models_{ns}$  *c*

(*proof*)

**lemma** *not-zero-satisfies*: *poly c = 0*  $\Longrightarrow$   $\neg$  *zero-satisfies c*  $\Longrightarrow$   $\neg$  *v*  $\models_{ns}$  *c*

(*proof*)

**fun**

*preprocess'* :: (*'i, QDelta*) *i-ns-constraint list*  $\Rightarrow$  *var*  $\Rightarrow$  *'i istate* **where**

*preprocess'* [] *v* = *IState v* [] [] ( $\lambda$  *p*. *None*) []

| *preprocess'* ((*i, h*) # *t*) *v* = (*let s' = preprocess' t v*; *p = poly h*; *is-monom-h = is-monom p*;

*v'* = *FirstFreshVariable s'*;

*t'* = *Tableau s'*;

*a'* = *Atoms s'*;

*m'* = *Poly-Mapping s'*;

*u'* = *UnsatIndices s' in*

*if is-monom-h then IState v' t'*

((*i, qdelta-constraint-to-atom h v'*) # *a'*) *m' u'*

*else if p = 0 then*

*if zero-satisfies h then s' else*

*IState v' t' a' m' (i # u')*

*else (case m' p of Some v  $\Rightarrow$*

*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 =>
  IState v' t' ((i,qdelta-constraint-to-atom' h v) # a') m' u'
| None => IState (v' + 1) (linear-poly-to-eq p v' # t')
  ((i,qdelta-constraint-to-atom' h v') # a') (m' (p ↦ v')) u')
) <proof>

```

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

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

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

**lemma** normalize-ns-constraint[simp]:  $v \models_{ns} (\text{normalize-ns-constraint } c) \iff v \models_{ns} (c :: 'a :: lrv \text{ ns-constraint})$   
<proof>

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

**lemma** i-satisfies-normalize-ns-constraint[simp]:  $Iv \models_{inss} (\text{map-prod id normalize-ns-constraint } 'cs)$   
 $\iff Iv \models_{inss} cs$   
<proof>

**abbreviation** max-var:: QDelta ns-constraint => var **where**



$max\text{-}var\ C \equiv Abstract\text{-}Linear\text{-}Poly.max\text{-}var\ (poly\ C)$

**fun**

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

**definition**

$preprocess\text{-}part\ 1 :: ('i, QDelta)\ i\text{-}ns\text{-}constraint\ list \Rightarrow tableau \times (('i, QDelta)\ i\text{-}atom\ list) \times 'i\ list$  **where**  
 $preprocess\text{-}part\ 1\ l \equiv let\ start = start\text{-}fresh\text{-}variable\ l; is = preprocess'\ l\ start\ in$   
 $(Tableau\ is,\ Atoms\ is,\ UnsatIndices\ is)$

**lemma**  $lhs\text{-}linear\text{-}poly\text{-}to\text{-}eq$  [simp]:

$lhs\ (linear\text{-}poly\text{-}to\text{-}eq\ h\ v) = v$   
 $\langle proof \rangle$

**lemma**  $rvars\text{-}eq\text{-}linear\text{-}poly\text{-}to\text{-}eq$  [simp]:

$rvars\text{-}eq\ (linear\text{-}poly\text{-}to\text{-}eq\ h\ v) = vars\ h$   
 $\langle proof \rangle$

**lemma**  $fresh\text{-}var\text{-}monoinc$ :

$FirstFreshVariable\ (preprocess'\ cs\ start) \geq start$   
 $\langle proof \rangle$

**abbreviation**  $vars\text{-}constraints$  **where**

$vars\text{-}constraints\ cs \equiv \bigcup\ (set\ (map\ vars\ (map\ poly\ cs)))$

**lemma**  $start\text{-}fresh\text{-}variable\text{-}fresh$ :

$\forall\ var \in vars\text{-}constraints\ (flat\text{-}list\ cs). var < start\text{-}fresh\text{-}variable\ cs$   
 $\langle proof \rangle$

**lemma**  $vars\text{-}tableau\text{-}vars\text{-}constraints$ :

$rvars\ (Tableau\ (preprocess'\ cs\ start)) \subseteq vars\text{-}constraints\ (flat\text{-}list\ cs)$   
 $\langle proof \rangle$

**lemma**  $lvars\text{-}tableau\text{-}ge\text{-}start$ :

$\forall\ var \in lvars\ (Tableau\ (preprocess'\ cs\ start)). var \geq start$   
 $\langle proof \rangle$

**lemma**  $rhs\text{-}no\text{-}zero\text{-}tableau\text{-}start$ :

$0 \notin rhs\ 'set\ (Tableau\ (preprocess'\ cs\ start))$   
 $\langle proof \rangle$

**lemma**  $first\text{-}fresh\text{-}variable\text{-}not\text{-}in\text{-}lvars$ :

$\forall\ var \in lvars\ (Tableau\ (preprocess'\ cs\ start)). FirstFreshVariable\ (preprocess'\ cs\ start) > var$   
 $\langle proof \rangle$

**lemma** *sat-atom-sat-eq-sat-constraint-non-monom*:

**assumes**  $v \models_a \text{qdelta-constraint-to-atom } h \text{ var } v \models_e \text{linear-poly-to-eq } (\text{poly } h) \text{ var}$   
 $\neg \text{is-monom } (\text{poly } h)$   
**shows**  $v \models_{ns} h$   
 $\langle \text{proof} \rangle$

**lemma** *qdelta-constraint-to-atom-monom*:

**assumes**  $\text{is-monom } (\text{poly } h)$   
**shows**  $v \models_a \text{qdelta-constraint-to-atom } h \text{ var} \longleftrightarrow v \models_{ns} h$   
 $\langle \text{proof} \rangle$

**lemma** *preprocess'-Tableau-Poly-Mapping-None*: (*Poly-Mapping* (*preprocess' cs start*))

$p = \text{None}$   
 $\implies \text{linear-poly-to-eq } p \text{ v} \notin \text{set } (\text{Tableau } (\text{preprocess' cs start}))$   
 $\langle \text{proof} \rangle$

**lemma** *preprocess'-Tableau-Poly-Mapping-Some*: (*Poly-Mapping* (*preprocess' cs start*))

$p = \text{Some } v$   
 $\implies \text{linear-poly-to-eq } p \text{ v} \in \text{set } (\text{Tableau } (\text{preprocess' cs start}))$   
 $\langle \text{proof} \rangle$

**lemma** *preprocess'-Tableau-Poly-Mapping-Some'*: (*Poly-Mapping* (*preprocess' cs start*))  $p = \text{Some } v$

$\implies \exists h. \text{poly } h = p \wedge \neg \text{is-monom } (\text{poly } h) \wedge \text{qdelta-constraint-to-atom } h \text{ v} \in \text{flat } (\text{set } (\text{Atoms } (\text{preprocess' cs start})))$   
 $\langle \text{proof} \rangle$

**lemma** *not-one-le-zero-qdelta*:  $\neg (1 \leq (0 :: QDelta)) \langle \text{proof} \rangle$

**lemma** *one-zero-contrad[dest,consumes 2]*:  $1 \leq x \implies (x :: QDelta) \leq 0 \implies \text{False}$

$\langle \text{proof} \rangle$

**lemma** *i-preprocess'-sat*:

**assumes**  $(I, v) \models_{ias} \text{set } (\text{Atoms } (\text{preprocess' s start})) \text{ v} \models_t \text{Tableau } (\text{preprocess' s start})$

$I \cap \text{set } (\text{UnsatIndices } (\text{preprocess' s start})) = \{\}$

**shows**  $(I, v) \models_{inss} \text{set } s$

$\langle \text{proof} \rangle$

**lemma** *preprocess'-sat*:

**assumes**  $v \models_{as} \text{flat } (\text{set } (\text{Atoms } (\text{preprocess' s start}))) \text{ v} \models_t \text{Tableau } (\text{preprocess' s start}) \text{ set } (\text{UnsatIndices } (\text{preprocess' s start})) = \{\}$

**shows**  $v \models_{nss} \text{flat } (\text{set } s)$

$\langle \text{proof} \rangle$

**lemma** *sat-constraint-valuation*:

**assumes**  $\forall \text{var} \in \text{vars } (\text{poly } c). v1 \text{ var} = v2 \text{ var}$

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

$\langle proof \rangle$

**lemma** *atom-var-first*:

**assumes**  $a \in flat (set (Atoms (preprocess' cs start))) \forall var \in vars-constraints$   
(*flat-list cs*).  $var < start$

**shows** *atom-var*  $a < FirstFreshVariable (preprocess' cs start)$

$\langle proof \rangle$

**lemma** *satisfies-tableau-satisfies-tableau*:

**assumes**  $v1 \models_t t \forall var \in tvars t. v1 var = v2 var$

**shows**  $v2 \models_t t$

$\langle proof \rangle$

**lemma** *preprocess'-unsat-indices*:

**assumes**  $i \in set (UnsatIndices (preprocess' s start))$

**shows**  $\neg (\{i\}, v) \models_{inss} set s$

$\langle proof \rangle$

**lemma** *preprocess'-unsat*:

**assumes**  $(I, v) \models_{inss} set s vars-constraints (flat-list s) \subseteq V \forall var \in V. var <$   
*start*

**shows**  $\exists v'. (\forall var \in V. v var = v' var)$

$\wedge v' \models_{as} restrict-to I (set (Atoms (preprocess' s start)))$

$\wedge v' \models_t Tableau (preprocess' s start)$

$\langle proof \rangle$

**lemma** *lvars-distinct*:

*distinct (map lhs (Tableau (preprocess' cs start)))*

$\langle proof \rangle$

**lemma** *normalized-tableau-preprocess'*:  $\Delta (Tableau (preprocess' cs (start-fresh-variable$   
*cs)))*

$\langle proof \rangle$

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**  $t = t1 @ (x, p) \# t2$

**and**  $t': t' = t1 @ t2$

**and**  $tv: tv = (\lambda v. upd x (p \{ \langle v \rangle \}) v)$

**and**  $x-as: x \notin atom-var 'snd' 'set as$

**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

$(I, \langle w \rangle) \models_{ias} set as \implies (I, \langle tv w \rangle) \models_{ias} set as$  — 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

$\langle proof \rangle$

**definition** *pivot-tableau-eq* ::  $tableau \Rightarrow eq \Rightarrow tableau \Rightarrow var \Rightarrow tableau \times eq \times$

**tableau where**

*pivot-tableau-eq*  $t1$   $eq$   $t2$   $x \equiv let$   $eq' = pivot\text{-}eq$   $eq$   $x$ ;  $m = map$   $(\lambda e. subst\text{-}var\text{-}eq$   
 $x$   $(rhs$   $eq')$   $e)$  **in**  
 $(m$   $t1, eq', m$   $t2)$

**lemma** *pivot-tableau-eq*: **assumes**  $t: t = t1 @ eq \# t2$   $t' = t1' @ eq' \# t2'$

**and**  $x: x \in rvars\text{-}eq$   $eq$  **and**  $norm: \Delta t$  **and**  $pte: pivot\text{-}tableau\text{-}eq$   $t1$   $eq$   $t2$   $x =$   
 $(t1', eq', t2')$

**shows**  $\Delta t'$   $lhs$   $eq' = x$   $(v :: 'a :: lrv$   $valuation)$   $\models_t t' \longleftrightarrow v \models_t t$

$\langle proof \rangle$

**function** *preprocess-opt*  $:: var$   $set \Rightarrow tableau \Rightarrow tableau \Rightarrow tableau \times ((var, 'a ::$   
 $lrv)mapping \Rightarrow (var, 'a)mapping)$  **where**

*preprocess-opt*  $X$   $t1$   $\square = (t1, id)$

$| preprocess\text{-}opt$   $X$   $t1$   $((x, p) \# t2) = (if$   $x \notin X$  **then**

*case* *preprocess-opt*  $X$   $t1$   $t2$  **of**  $(t, tv) \Rightarrow (t, (\lambda v. upd$   $x$   $(p$   $\{\! \{v\} \!\})$   $v)$   $o$   $tv)$

*else* *case* *find*  $(\lambda x. x \notin X)$   $(Abstract\text{-}Linear\text{-}Poly.vars\text{-}list$   $p)$  **of**

*None*  $\Rightarrow preprocess\text{-}opt$   $X$   $((x, p) \# t1)$   $t2$

$| Some$   $y \Rightarrow case$  *pivot-tableau-eq*  $t1$   $(x, p)$   $t2$   $y$  **of**

$(tt1, (z, q), tt2) \Rightarrow case$  *preprocess-opt*  $X$   $tt1$   $tt2$  **of**  $(t, tv) \Rightarrow (t, (\lambda v. upd$   $z$   $(q$   $\{\! \{$

$\langle v \rangle \!\})$   $v)$   $o$   $tv))$

$\langle proof \rangle$

**termination**  $\langle proof \rangle$

**lemma** *preprocess-opt*: **assumes**  $X = atom\text{-}var$   $'$   $snd$   $'$   $set$   $as$

*preprocess-opt*  $X$   $t1$   $t2 = (t', tv)$   $\Delta t = rev$   $t1 @ t2$

**shows**  $\Delta t'$

$(\langle w \rangle :: 'a :: lrv$   $valuation)$   $\models_t t' \Longrightarrow \langle tv$   $w \rangle \models_t t$

$(I, \langle w \rangle) \models_{ias}$   $set$   $as \Longrightarrow (I, \langle tv$   $w \rangle) \models_{ias}$   $set$   $as$

$v \models_t t \Longrightarrow (v :: 'a$   $valuation)$   $\models_t t'$

$\langle proof \rangle$

**definition** *preprocess-part-2*  $as$   $t = preprocess\text{-}opt$   $(atom\text{-}var$   $'$   $snd$   $'$   $set$   $as)$   $\square t$

**lemma** *preprocess-part-2*: **assumes** *preprocess-part-2*  $as$   $t = (t', tv)$   $\Delta t$

**shows**  $\Delta t'$

$(\langle w \rangle :: 'a :: lrv$   $valuation)$   $\models_t t' \Longrightarrow \langle tv$   $w \rangle \models_t t$

$(I, \langle w \rangle) \models_{ias}$   $set$   $as \Longrightarrow (I, \langle tv$   $w \rangle) \models_{ias}$   $set$   $as$

$v \models_t t \Longrightarrow (v :: 'a$   $valuation)$   $\models_t t'$

$\langle proof \rangle$

**definition** *preprocess*  $:: ('i, QDelta)$   $i\text{-}ns\text{-}constraint$   $list \Rightarrow - \times - \times (- \Rightarrow (var, QDelta)mapping)$

$\times 'i$   $list$  **where**

*preprocess*  $l = (case$  *preprocess-part-1*  $(map$   $(map\text{-}prod$   $id$   $normalize\text{-}ns\text{-}constraint)$

$l)$  **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**  $\Delta t$   
*fst* ' set as  $\cup$  set ui  $\subseteq$  *fst* ' set cs  
*distinct-indices-ns* (set cs)  $\implies$  *distinct-indices-atoms* (set as)  
 $I \cap \text{set } ui = \{\}$   $\implies (I, \langle v \rangle) \models_{ias} \text{set } as \implies$   
 $\langle v \rangle \models_t t \implies (I, \langle \text{trans-}v \ v \rangle) \models_{inss} \text{set } cs$   
 $i \in \text{set } ui \implies \nexists v. (\{i\}, v) \models_{inss} \text{set } cs$   
 $\exists v. (I, v) \models_{inss} \text{set } cs \implies \exists v'. (I, v') \models_{ias} \text{set } as \wedge v' \models_t t$   
 <proof>

**interpretation** *PreprocessDefault*: Preprocess preprocess  
 <proof>

**global-interpretation** *Solve-exec-ns'Default*: Solve-exec-ns' preprocess assert-all-code  
**defines** solve-exec-ns-code = Solve-exec-ns'Default.solve-exec-ns  
 <proof>

### primrec

*constraint-to-qdelta-constraint*:: constraint  $\implies$  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)]  
 | *constraint-to-qdelta-constraint* (LTPP l1 l2) = [LEQ-ns (l1 - l2) (QDelta.QDelta 0 (-1))]  
 | *constraint-to-qdelta-constraint* (GTPP l1 l2) = [GEQ-ns (l1 - l2) (QDelta.QDelta 0 1)]  
 | *constraint-to-qdelta-constraint* (LEQPP l1 l2) = [LEQ-ns (l1 - l2) 0]  
 | *constraint-to-qdelta-constraint* (GEQPP l1 l2) = [GEQ-ns (l1 - l2) 0]  
 | *constraint-to-qdelta-constraint* (EQPP l1 l2) = [LEQ-ns (l1 - l2) 0, GEQ-ns (l1 - l2) 0]

### primrec

*i-constraint-to-qdelta-constraint*:: 'i i-constraint  $\implies$  ('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  $\implies$  ('i, QDelta) i-ns-constraint list **where**  
*to-ns* l  $\equiv$  concat (map i-constraint-to-qdelta-constraint l)

### primrec

$\delta 0\text{-val} :: QDelta\ ns\text{-constraint} \Rightarrow QDelta\ valuation \Rightarrow rat$  **where**  
 $\delta 0\text{-val}\ (LEQ\text{-ns}\ lll\ rrr)\ vl = \delta 0\ lll\ \{vl\}\ rrr$   
 $|\ \delta 0\text{-val}\ (GEQ\text{-ns}\ lll\ rrr)\ vl = \delta 0\ rrr\ lll\ \{vl\}$

**primrec**

$\delta 0\text{-val-min} :: QDelta\ ns\text{-constraint}\ list \Rightarrow QDelta\ valuation \Rightarrow rat$  **where**  
 $\delta 0\text{-val-min}\ []\ vl = 1$   
 $|\ \delta 0\text{-val-min}\ (h\#\!t)\ vl = \min\ (\delta 0\text{-val-min}\ t\ vl)\ (\delta 0\text{-val}\ h\ vl)$

**abbreviation vars-list-constraints where**

$vars\text{-list-constraints}\ cs \equiv \text{remdups}\ (\text{concat}\ (\text{map}\ \text{Abstract-Linear-Poly}\ \text{vars-list}\ (\text{map}\ \text{poly}\ cs)))$

**definition**

$from\text{-ns} :: (var,\ QDelta)\ mapping \Rightarrow QDelta\ ns\text{-constraint}\ list \Rightarrow (var,\ rat)\ mapping$   
**where**  
 $from\text{-ns}\ vl\ cs \equiv \text{let}\ \delta = \delta 0\text{-val-min}\ cs\ \langle vl \rangle\ \text{in}$   
 $\text{Mapping}\ \text{tabulate}\ (vars\text{-list-constraints}\ cs)\ (\lambda\ var.\ \text{val}\ (\langle vl \rangle\ var)\ \delta)$

**global-interpretation SolveExec'Default: SolveExec' to-ns from-ns solve-exec-ns-code**

**defines**  $solve\text{-exec-code} = \text{SolveExec'Default}\ \text{solve-exec}$

**and**  $solve\text{-code} = \text{SolveExec'Default}\ \text{solve}$

$\langle proof \rangle$

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

Simplex version with indexed constraints as input

**definition**  $simplex\text{-index} :: 'i\ i\text{-constraint}\ list \Rightarrow 'i\ list + (var,\ rat)\ mapping$  **where**  
 $simplex\text{-index} = solve\text{-exec-code}$

**lemma**  $simplex\text{-index}$ :

$simplex\text{-index}\ cs = \text{Unsat}\ I \implies \text{set}\ I \subseteq \text{fst}\ ' \text{set}\ cs \wedge \neg (\exists\ v.\ (\text{set}\ I,\ v) \models_{ics}\ \text{set}\ cs) \wedge$

$(\text{distinct-indices}\ cs \longrightarrow (\forall\ J \subset \text{set}\ I.\ (\exists\ v.\ (J,\ v) \models_{ics}\ \text{set}\ cs)))$  — minimal unsat core

$simplex\text{-index}\ cs = \text{Sat}\ v \implies \langle v \rangle \models_{cs}\ (\text{snd}\ ' \text{set}\ cs)$  — satisfying assingment  
 $\langle proof \rangle$

Simplex version where indices will be created

**definition**  $simplex$  **where**  $simplex\ cs = simplex\text{-index}\ (\text{zip}\ [0..\text{length}\ cs]\ cs)$

**lemma**  $simplex$ :

$simplex\ cs = \text{Unsat}\ I \implies \neg (\exists\ v.\ v \models_{cs}\ \text{set}\ cs)$  — unsat of original constraints

$simplex\ cs = \text{Unsat}\ I \implies \text{set}\ I \subseteq \{0..\text{length}\ cs\} \wedge \neg (\exists\ v.\ v \models_{cs}\ \{cs\ !\ i \mid i.\ i \in \text{set}\ I\})$

$\wedge (\forall\ J \subset \text{set}\ I.\ \exists\ v.\ v \models_{cs}\ \{cs\ !\ i \mid i.\ i \in J\})$  — minimal unsat core

*simplex cs = Sat v*  $\implies \langle v \rangle \models_{cs} \text{set } cs$  — satisfying assignment  
 $\langle \text{proof} \rangle$

check executability

**lemma** *case simplex [LTPP (lp-monom 2 1) (lp-monom 3 2 - lp-monom 3 0), GT (lp-monom 1 1) 5]*

*of Sat -  $\implies$  True | Unsat -  $\implies$  False*

$\langle \text{proof} \rangle$

check unsat core

**lemma**

*case simplex-index [*

*(1 :: int, LT (lp-monom 1 1) 4),*

*(2, GTPP (lp-monom 2 1) (lp-monom 1 2)),*

*(3, EQPP (lp-monom 1 1) (lp-monom 2 2)),*

*(4, GT (lp-monom 2 2) 5),*

*(5, GT (lp-monom 3 0) 7)]*

*of Sat -  $\implies$  False | Unsat I  $\implies$  set I = {1,3,4} — Constraints 1,3,4 are unsat core*

$\langle \text{proof} \rangle$

**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  $\implies$  's and*

*assert-s :: ('i,'a :: lrv) i-atom  $\implies$  's  $\implies$  'i list + 's and*

*check-s :: 's  $\implies$  'i list + 's and*

*solution-s :: 's  $\implies$  (var, 'a) mapping and*

*checkpoint-s :: 's  $\implies$  'c and*

*backtrack-s :: 'c  $\implies$  's  $\implies$  's and*

*precond-s :: tableau  $\implies$  bool and*

*invariant-s :: tableau  $\implies$  ('i,'a) i-atom set  $\implies$  's  $\implies$  bool and*

*checked-s :: tableau  $\implies$  ('i,'a) i-atom set  $\implies$  's  $\implies$  bool*

**assumes**

*assert-s-ok*:  $\text{invariant-s } t \text{ as } s \implies \text{assert-s } a \text{ s} = \text{Inr } s' \implies$   
 $\text{invariant-s } t \text{ (insert } a \text{ as) } s' \text{ and}$   
*assert-s-unsat*:  $\text{invariant-s } t \text{ as } s \implies \text{assert-s } a \text{ s} = \text{Unsat } I \implies$   
 $\text{minimal-unsat-core-tabl-atoms (set } I) t \text{ (insert } a \text{ as) and}$   
*check-s-ok*:  $\text{invariant-s } t \text{ as } s \implies \text{check-s } s = \text{Inr } s' \implies$   
 $\text{checked-s } t \text{ as } s' \text{ and}$   
*check-s-unsat*:  $\text{invariant-s } t \text{ as } s \implies \text{check-s } s = \text{Unsat } I \implies$   
 $\text{minimal-unsat-core-tabl-atoms (set } I) t \text{ as and}$   
*init-s*:  $\text{precond-s } t \implies \text{checked-s } t \text{ \{\}} \text{ (init-s } t) \text{ and}$   
*solution-s*:  $\text{checked-s } t \text{ as } s \implies \text{solution-s } s = v \implies \langle v \rangle \models_t t \wedge \langle v \rangle \models_{as} \text{Simplex.flat}$   
 $as \text{ and}$   
*backtrack-s*:  $\text{checked-s } t \text{ as } s \implies \text{checkpoint-s } s = c$   
 $\implies \text{invariant-s } t \text{ bs } s' \implies \text{backtrack-s } c \text{ s}' = s'' \implies as \subseteq bs \implies \text{invariant-s } t$   
 $as \text{ s}'' \text{ and}$   
*checked-invariant-s*:  $\text{checked-s } t \text{ as } s \implies \text{invariant-s } t \text{ as } s$   
**begin**

**fun** *assert-all-s* ::  $('i, 'a) \text{ i-atom list} \Rightarrow 's \Rightarrow 'i \text{ list} + 's \text{ where}$   
 $\text{assert-all-s } [] \text{ s} = \text{Inr } s$   
 $| \text{assert-all-s } (a \# as) \text{ s} = (\text{case } \text{assert-s } a \text{ s} \text{ of } \text{Unsat } I \Rightarrow \text{Unsat } I$   
 $| \text{Inr } s' \Rightarrow \text{assert-all-s } as \text{ s}')$

**lemma** *assert-all-s-ok*:  $\text{invariant-s } t \text{ as } s \implies \text{assert-all-s } bs \text{ s} = \text{Inr } s' \implies$   
 $\text{invariant-s } t \text{ (set } bs \cup as) \text{ s}'$   
 $\langle \text{proof} \rangle$

**lemma** *assert-all-s-unsat*:  $\text{invariant-s } t \text{ as } s \implies \text{assert-all-s } bs \text{ s} = \text{Unsat } I \implies$   
 $\text{minimal-unsat-core-tabl-atoms (set } I) t \text{ (as } \cup \text{ set } bs)$   
 $\langle \text{proof} \rangle$

**end**

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

**locale** *Incremental-State-Ops-Simplex* = *AssertBoundNoLhs* *assert-bound* + *Init*  
 $\text{init} + \text{Check } \text{check}$

**for** *assert-bound* ::  $('i, 'a::\text{lrV}) \text{ i-atom} \Rightarrow ('i, 'a) \text{ state} \Rightarrow ('i, 'a) \text{ state and}$   
 $\text{init} :: \text{tableau} \Rightarrow ('i, 'a) \text{ state and}$   
 $\text{check} :: ('i, 'a) \text{ state} \Rightarrow ('i, 'a) \text{ state}$   
**begin**

**definition** *invariant-s* **where**  
 $\text{invariant-s } t \text{ (as :: ('i, 'a) i-atom set) } s =$   
 $(\neg \mathcal{U} \text{ s} \wedge \models_{\text{noLhs}} \text{ s} \wedge$   
 $\triangle (\mathcal{T} \text{ s}) \wedge$   
 $\nabla \text{ s} \wedge$   
 $\diamond \text{ s} \wedge$   
 $(\forall v :: (\text{var} \Rightarrow 'a). v \models_t \mathcal{T} \text{ s} \longleftrightarrow v \models_t t) \wedge$   
 $\text{index-valid as } s \wedge$



*Simplex.flat as*  $\doteq \mathcal{B} s \wedge$   
*as*  $\models_i \mathcal{BI} s$ )

**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* *Inl (the ( $\mathcal{U}_c s'$ ))* *else* *Inr s'*)

**definition** *checkpoint-s* **where** *checkpoint-s s* =  $\mathcal{B}_i s$

**fun** *backtrack-s* **where** *backtrack-s (bl, bu)* (*State t bl-old bu-old v u uc*) = *State t*  
*bl bu v u uc*

**lemma** *invariant-sD*: **assumes** *invariant-s t as s*

**shows**  $\neg \mathcal{U} s \models_{noth_s} s \triangle (\mathcal{T} s) \nabla s \diamond s$   
*Simplex.flat as*  $\doteq \mathcal{B} s$  *as*  $\models_i \mathcal{BI} s$  *index-valid as s*  
 $(\forall v :: (\text{var} \Rightarrow 'a). v \models_t \mathcal{T} s \longleftrightarrow v \models_t t)$   
 ⟨*proof*⟩

**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*  
 ⟨*proof*⟩

**lemma** *incremental-atom-ops*: *Incremental-Atom-Ops*

*init assert-s check-s*  $\mathcal{V}$  *checkpoint-s backtrack-s*  $\triangle$  *invariant-s checked-s*  
 ⟨*proof*⟩

**end**

## 7.2 Intermediate Layer: Incremental Non-Strict Constraints

Interface

**locale** *Incremental-NS-Constraint-Ops* = **fixes**

*init-nsc* ::  $('i, 'a :: lrv)$  *i-ns-constraint list*  $\Rightarrow 's$  **and**  
*assert-nsc* ::  $'i \Rightarrow 's \Rightarrow 'i \text{ list} + 's$  **and**  
*check-nsc* ::  $'s \Rightarrow 'i \text{ list} + 's$  **and**  
*solution-nsc* ::  $'s \Rightarrow (\text{var}, 'a)$  *mapping* **and**  
*checkpoint-nsc* ::  $'s \Rightarrow 'c$  **and**  
*backtrack-nsc* ::  $'c \Rightarrow 's \Rightarrow 's$  **and**  
*invariant-nsc* ::  $('i, 'a)$  *i-ns-constraint list*  $\Rightarrow 'i \text{ set} \Rightarrow 's \Rightarrow \text{bool}$  **and**

$checked\text{-}nsc :: ('i, 'a) \text{ i-ns-constraint list} \Rightarrow 'i \text{ set} \Rightarrow 's \Rightarrow \text{bool}$   
**assumes**  
 $assert\text{-}nsc\text{-}ok: invariant\text{-}nsc \ nsc \ J \ s \Longrightarrow assert\text{-}nsc \ j \ s = Inr \ s' \Longrightarrow$   
 $invariant\text{-}nsc \ nsc \ (insert \ j \ J) \ s' \ \mathbf{and}$   
 $assert\text{-}nsc\text{-}unsat: invariant\text{-}nsc \ nsc \ J \ s \Longrightarrow assert\text{-}nsc \ j \ s = Unsat \ I \Longrightarrow$   
 $set \ I \subseteq insert \ j \ J \wedge minimal\text{-}unsat\text{-}core\text{-}ns \ (set \ I) \ (set \ nsc) \ \mathbf{and}$   
 $check\text{-}nsc\text{-}ok: invariant\text{-}nsc \ nsc \ J \ s \Longrightarrow check\text{-}nsc \ s = Inr \ s' \Longrightarrow$   
 $checked\text{-}nsc \ nsc \ J \ s' \ \mathbf{and}$   
 $check\text{-}nsc\text{-}unsat: invariant\text{-}nsc \ nsc \ J \ s \Longrightarrow check\text{-}nsc \ s = Unsat \ I \Longrightarrow$   
 $set \ I \subseteq J \wedge minimal\text{-}unsat\text{-}core\text{-}ns \ (set \ I) \ (set \ nsc) \ \mathbf{and}$   
 $init\text{-}nsc: checked\text{-}nsc \ nsc \ \{\} \ (init\text{-}nsc \ nsc) \ \mathbf{and}$   
 $solution\text{-}nsc: checked\text{-}nsc \ nsc \ J \ s \Longrightarrow solution\text{-}nsc \ s = v \Longrightarrow (J, \langle v \rangle) \models_{ins\ s} set$   
 $nsc \ \mathbf{and}$   
 $backtrack\text{-}nsc: checked\text{-}nsc \ nsc \ J \ s \Longrightarrow checkpoint\text{-}nsc \ s = c$   
 $\Longrightarrow invariant\text{-}nsc \ nsc \ K \ s' \Longrightarrow backtrack\text{-}nsc \ c \ s' = s'' \Longrightarrow J \subseteq K \Longrightarrow$   
 $invariant\text{-}nsc \ nsc \ J \ s'' \ \mathbf{and}$   
 $checked\text{-}invariant\text{-}nsc: checked\text{-}nsc \ nsc \ J \ s \Longrightarrow invariant\text{-}nsc \ nsc \ J \ s$

Implementation via the Simplex operation preprocess and the incremental operations for atoms.

**fun**  $create\text{-}map :: ('i \times 'a) \text{ list} \Rightarrow ('i, ('i \times 'a) \text{ list}) \text{ mapping}$  **where**  
 $create\text{-}map \ [] = Mapping.empty$   
 $| create\text{-}map \ ((i, a) \# xs) = (let \ m = create\text{-}map \ xs \ in$   
 $case \ Mapping.lookup \ m \ i \ of$   
 $None \Rightarrow Mapping.update \ i \ [(i, a)] \ m$   
 $| Some \ ias \Rightarrow Mapping.update \ i \ ((i, a) \# ias) \ m)$

**definition**  $list\text{-}map\text{-}to\text{-}fun :: ('i, ('i \times 'a) \text{ list}) \text{ mapping} \Rightarrow 'i \Rightarrow ('i \times 'a) \text{ list}$  **where**  
 $list\text{-}map\text{-}to\text{-}fun \ m \ i = (case \ Mapping.lookup \ m \ i \ of \ None \Rightarrow [] \ | \ Some \ ias \Rightarrow ias)$

**lemma**  $list\text{-}map\text{-}to\text{-}fun\text{-}create\text{-}map: set \ (list\text{-}map\text{-}to\text{-}fun \ (create\text{-}map \ ias) \ i) = set$   
 $ias \cap \{i\} \times UNIV$   
*<proof>*

**fun**  $sum\text{-}wrap :: ('c \Rightarrow 's \Rightarrow 'i \text{ list} + 's)$   
 $\Rightarrow 'c \times 's \Rightarrow 'i \text{ list} + ('c \times 's)$  **where**  
 $sum\text{-}wrap \ f \ (asi, s) = (case \ f \ asi \ s \ of \ Unsat \ I \Rightarrow Unsat \ I \ | \ Inr \ s' \Rightarrow Inr \ (asi, s'))$

**definition**  $check\text{-}nsc$  **where**  $check\text{-}nsc \ check\text{-}s = sum\text{-}wrap \ (\lambda \ asitv. \ check\text{-}s)$

**definition**  $assert\text{-}nsc$  **where**  $assert\text{-}nsc \ assert\text{-}all\text{-}s \ i = (\lambda \ ((asi, tv, ui), s).$   
 $if \ i \in set \ ui \ then \ Unsat \ [i] \ else$   
 $case \ assert\text{-}all\text{-}s \ (list\text{-}map\text{-}to\text{-}fun \ asi \ i) \ s \ of \ Unsat \ I \Rightarrow Unsat \ I \ | \ Inr \ s' \Rightarrow Inr$   
 $((asi, tv, ui), s'))$

**fun**  $checkpoint\text{-}nsc$  **where**  $checkpoint\text{-}nsc \ checkpoint\text{-}s \ (asi\text{-}tv\text{-}ui, s) = checkpoint\text{-}s$   
 $s$

**fun**  $backtrack\text{-}nsc$  **where**  $backtrack\text{-}nsc \ backtrack\text{-}s \ c \ (asi\text{-}tv\text{-}ui, s) = (asi\text{-}tv\text{-}ui, back$   
 $track\text{-}s \ c \ s)$

**fun**  $solution\text{-}nsc$  **where**  $solution\text{-}nsc \ solution\text{-}s \ ((asi, tv, ui), s) = tv \ (solution\text{-}s \ s)$

**locale** *Incremental-Atom-Ops-For-NS-Constraint-Ops* =  
*Incremental-Atom-Ops* *init-s* *assert-s* *check-s* *solution-s* *checkpoint-s* *backtrack-s*  
 $\Delta$   
*invariant-s* *checked-s*  
+ *Preprocess* *preprocess*  
**for**  
*init-s* :: *tableau*  $\Rightarrow$  's **and**  
*assert-s* :: ('i :: *linorder*, 'a :: *lrv*) *i-atom*  $\Rightarrow$  's  $\Rightarrow$  'i list + 's **and**  
*check-s* :: 's  $\Rightarrow$  'i list + 's **and**  
*solution-s* :: 's  $\Rightarrow$  (*var*, 'a) *mapping* **and**  
*checkpoint-s* :: 's  $\Rightarrow$  'c **and**  
*backtrack-s* :: 'c  $\Rightarrow$  's  $\Rightarrow$  's **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** *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'*  $\wedge$   
 $(\forall i. \text{set } (\text{list-map-to-fun } \text{asi } i) = (\text{as} \cap (\{i\} \times \text{UNIV})))$ )

**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*)  $\models_{i.as}$  (*ats*  $\cap$  (*J*  $\times$  *UNIV*)))  $\longleftrightarrow$  ((*I*  
 $\cap$  *J, v*)  $\models_{i.as}$  *ats*)  
<*proof*>

**lemma** *ns-constraints-ops*: *Incremental-NS-Constraint-Ops* *init-nsc* (*assert-nsc* *as-*  
*sert-all-s*)  
(*check-nsc* *check-s*) (*solution-nsc* *solution-s*) (*checkpoint-nsc* *checkpoint-s*) (*backtrack-nsc*  
*backtrack-s*)  
*invariant-nsc* *checked-nsc*  
<*proof*>

end

### 7.3 Highest Layer: Incremental Constraints

Interface

```
locale Incremental-Simplex-Ops = fixes
  init-cs :: 'i i-constraint list  $\Rightarrow$  's and
  assert-cs :: 'i  $\Rightarrow$  's  $\Rightarrow$  'i list + 's and
  check-cs :: 's  $\Rightarrow$  'i list + 's and
  solution-cs :: 's  $\Rightarrow$  rat valuation and
  checkpoint-cs :: 's  $\Rightarrow$  'c and
  backtrack-cs :: 'c  $\Rightarrow$  's  $\Rightarrow$  's and
  invariant-cs :: 'i i-constraint list  $\Rightarrow$  'i set  $\Rightarrow$  's  $\Rightarrow$  bool and
  checked-cs :: 'i i-constraint list  $\Rightarrow$  'i set  $\Rightarrow$  's  $\Rightarrow$  bool
assumes
  assert-cs-ok: invariant-cs cs J s  $\Longrightarrow$  assert-cs j s = Inr s'  $\Longrightarrow$ 
    invariant-cs cs (insert j J) s' and
  assert-cs-unsat: invariant-cs cs J s  $\Longrightarrow$  assert-cs j s = Unsat I  $\Longrightarrow$ 
    set I  $\subseteq$  insert j J  $\wedge$  minimal-unsat-core (set I) cs and
  check-cs-ok: invariant-cs cs J s  $\Longrightarrow$  check-cs s = Inr s'  $\Longrightarrow$ 
    checked-cs cs J s' and
  check-cs-unsat: invariant-cs cs J s  $\Longrightarrow$  check-cs s = Unsat I  $\Longrightarrow$ 
    set I  $\subseteq$  J  $\wedge$  minimal-unsat-core (set I) cs and
  init-cs: checked-cs cs {} (init-cs cs) and
  solution-cs: checked-cs cs J s  $\Longrightarrow$  solution-cs s = v  $\Longrightarrow$  (J, v)  $\models_{ics}$  set cs and
  backtrack-cs: checked-cs cs J s  $\Longrightarrow$  checkpoint-cs s = c
     $\Longrightarrow$  invariant-cs cs K s'  $\Longrightarrow$  backtrack-cs c s' = s''  $\Longrightarrow$  J  $\subseteq$  K  $\Longrightarrow$  invariant-cs
cs J s'' and
  checked-invariant-cs: checked-cs cs J s  $\Longrightarrow$  invariant-cs cs J s
```

Implementation via the Simplex-operation To-Ns and the Incremental Operations for Non-Strict Constraints

```
fun assert-cs where assert-cs ass i (cs,s) = (case ass i s of
  Unsat I  $\Rightarrow$  Unsat I
| Inr s'  $\Rightarrow$  Inr (cs, s'))
```

```
definition init-cs init tons cs = (let tons-cs = tons cs in (map snd (tons-cs), init tons-cs))
```

```
definition check-cs check s = sum-wrap ( $\lambda$  cs. check) s
```

```
fun checkpoint-cs where checkpoint-cs checkp (cs,s) = (checkp s)
```

```
fun backtrack-cs where backtrack-cs backt c (cs,s) = (cs, backt c s)
```

```
fun solution-cs where solution-cs sol from (cs,s) = ( $\langle$ from (sol s) cs $\rangle$ )
```

```
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
  invariant-nsc checked-nsc + To-ns to-ns from-ns
```

```

for
  init-nsc :: ('i,'a :: lrv) i-ns-constraint list ⇒ 's and
  assert-nsc :: 'i ⇒ 's ⇒ 'i list + 's and
  check-nsc :: 's ⇒ 'i list + 's and
  solution-nsc :: 's ⇒ (var, 'a) mapping and
  checkpoint-nsc :: 's ⇒ 'c and
  backtrack-nsc :: 'c ⇒ 's ⇒ 's and
  invariant-nsc :: ('i,'a) i-ns-constraint list ⇒ 'i set ⇒ 's ⇒ bool and
  checked-nsc :: ('i,'a) i-ns-constraint list ⇒ 'i set ⇒ 's ⇒ bool and
  to-ns :: 'i i-constraint list ⇒ ('i,'a) i-ns-constraint list and
  from-ns :: (var, 'a) mapping ⇒ 'a ns-constraint list ⇒ (var, rat) mapping
begin

fun invariant-cs where
  invariant-cs cs J (ds,s) = (ds = map snd (to-ns cs) ∧ invariant-nsc (to-ns cs) J s)
fun checked-cs where
  checked-cs cs J (ds,s) = (ds = map snd (to-ns cs) ∧ checked-nsc (to-ns cs) J s)

sublocale Incremental-Simplex-Ops init-cs init-nsc to-ns assert-cs assert-nsc
  check-cs check-nsc solution-cs solution-nsc from-ns checkpoint-cs checkpoint-nsc
  backtrack-cs backtrack-nsc
  invariant-cs checked-cs
  ⟨proof⟩

end

```

## 7.4 Concrete Implementation

### 7.4.1 Connecting all the locales

**definition** *assert-s* = *Incremental-State-Ops-Simplex.assert-s* *assert-bound-code*

**definition** *check-s* = *Incremental-State-Ops-Simplex.check-s* *check-code*

**definition** *checkpoint-s* = *Incremental-State-Ops-Simplex.checkpoint-s*

**definition** *backtrack-s* = *Incremental-State-Ops-Simplex.backtrack-s*

**definition** *invariant-s* = *Incremental-State-Ops-Simplex.invariant-s*

**definition** *checked-s* = *Incremental-State-Ops-Simplex.checked-s*

**global-interpretation** *Incremental-State-Ops-Simplex-Default*:

*Incremental-State-Ops-Simplex* *assert-bound-code* *init-state* *check-code*

**rewrites**

*Incremental-State-Ops-Simplex.assert-s* *assert-bound-code* = *assert-s* **and**

*Incremental-State-Ops-Simplex.check-s* *check-code* = *check-s* **and**

*Incremental-State-Ops-Simplex.backtrack-s* = *backtrack-s* **and**

*Incremental-State-Ops-Simplex.checkpoint-s* = *checkpoint-s* **and**

*Incremental-State-Ops-Simplex.invariant-s* = *invariant-s* **and**

*Incremental-State-Ops-Simplex.checked-s* = *checked-s*

⟨*proof*⟩

**definition** *assert-all-s* = *Incremental-Atom-Ops.assert-all-s* *assert-s*

**global-interpretation** *Incremental-Atom-Ops-Default:*

*Incremental-Atom-Ops init-state assert-s check-s  $\mathcal{V}$  checkpoint-s backtrack-s  $\Delta$  invariant-s checked-s*

**rewrites** *Incremental-Atom-Ops.assert-all-s assert-s = assert-all-s*  
*(proof)*

**definition** *init-nsc-code = (Incremental-Atom-Ops-For-NS-Constraint-Ops.init-nsc init-state preprocess :: -  $\Rightarrow$  (('a :: linorder, ('a  $\times$  QDelta atom) list) mapping  $\times$  -)  $\times$  ('a, QDelta) state)*

**definition** *assert-nsc-code = assert-nsc assert-all-s*

**definition** *check-nsc-code = check-nsc check-s*

**definition** *checkpoint-nsc-code = checkpoint-nsc checkpoint-s*

**definition** *solution-nsc-code = solution-nsc  $\mathcal{V}$*

**definition** *backtrack-nsc-code = backtrack-nsc backtrack-s*

**definition** *invariant-nsc = Incremental-Atom-Ops-For-NS-Constraint-Ops.invariant-nsc invariant-s preprocess*

**definition** *checked-nsc = Incremental-Atom-Ops-For-NS-Constraint-Ops.checked-nsc checked-s preprocess*

**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 invariant-s checked-s preprocess*

**rewrites**

*Incremental-Atom-Ops-For-NS-Constraint-Ops.init-nsc init-state preprocess = init-nsc-code* **and**

*check-nsc check-s = check-nsc-code* **and**

*checkpoint-nsc checkpoint-s = checkpoint-nsc-code* **and**

*solution-nsc  $\mathcal{V}$  = solution-nsc-code* **and**

*backtrack-nsc backtrack-s = backtrack-nsc-code* **and**

*assert-nsc assert-all-s = assert-nsc-code* **and**

*Incremental-Atom-Ops-For-NS-Constraint-Ops.invariant-nsc invariant-s preprocess = invariant-nsc* **and**

*Incremental-Atom-Ops-For-NS-Constraint-Ops.checked-nsc checked-s preprocess = checked-nsc* **and**

*Incremental-Atom-Ops.assert-all-s assert-s = assert-all-s*  
*(proof)*

**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*

**definition** *init-simplex' = (init-cs init-nsc-code to-ns :: 'i :: linorder i-constraint list  $\Rightarrow$  'i simplex-state')*

**definition** *assert-simplex' = (assert-cs assert-nsc-code :: 'i  $\Rightarrow$  'i simplex-state'  $\Rightarrow$  'i list + 'i simplex-state')*

**definition** *check-simplex' = (check-cs check-nsc-code :: 'i simplex-state'  $\Rightarrow$  'i list*

+ 'i simplex-state')

**definition** *solution-simplex'* = (*solution-cs solution-nsc-code from-ns* :: 'i simplex-state' ⇒ -)

**definition** *backtrack-simplex'* = (*backtrack-cs backtrack-nsc-code* :: - ⇒ 'i simplex-state' ⇒ -)

**definition** *checkpoint-simplex'* = (*checkpoint-cs checkpoint-nsc-code* :: 'i simplex-state' ⇒ -)

**definition** *invariant-simplex'* = *Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex.invariant-cs invariant-nsc to-ns*

**definition** *checked-simplex'* = *Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex.checked-cs checked-nsc to-ns*

**lemma** *case-sum-case-sum*: (*case (case x of Inl y ⇒ Inl (f1 y) | Inr z ⇒ Inr (f2 z))*)

*of Inl y ⇒ Inl (g1 y) | Inr z ⇒ Inr (g2 z) = (case x of Inl y ⇒ Inl (g1 (f1 y)) | Inr z ⇒ Inr (g2 (f2 z)))*

*<proof>*

The following code-lemmas unfold some layers in the code of the simplex-operations, so that the implementation immediately points to the operations working on simplex type (*i, QDelta*) *state*.

**lemmas** *code-lemmas* =

*fun-cong[OF init-simplex'-def, of cs for cs, unfolded init-cs-def*

*Incremental-Atom-Ops-For-NS-Constraint-Ops-Default.init-nsc-def]*

*fun-cong[OF fun-cong[OF assert-simplex'-def], of i (cs,((asi,tv,ui),s)) for i cs asi tv ui s,*

*unfolded assert-cs.simps assert-nsc-code-def assert-nsc-def case-sum-case-sum split]*

*fun-cong[OF check-simplex'-def, of (cs,(asi-tv,s)) for cs asi-tv s,*

*unfolded check-cs-def check-nsc-code-def check-nsc-def sum-wrap.simps case-sum-case-sum]*

*fun-cong[OF solution-simplex'-def, of (cs,((asi,tv),s)) for cs asi tv s,*

*unfolded solution-cs.simps solution-nsc-code-def solution-nsc.simps]*

*fun-cong[OF checkpoint-simplex'-def, of (cs,(asi-tv,s)) for cs asi-tv s,*

*unfolded checkpoint-nsc-code-def checkpoint-cs.simps checkpoint-nsc.simps]*

*fun-cong[OF fun-cong[OF backtrack-simplex'-def], of c (cs,(asi-tv,s)) for c cs asi-tv s,*

*unfolded backtrack-nsc-code-def backtrack-nsc.simps backtrack-cs.simps]*

**declare** *code-lemmas*[code]

**global-interpretation** *Incremental-Simplex*:

*Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex*

*init-nsc-code assert-nsc-code check-nsc-code solution-nsc-code checkpoint-nsc-code backtrack-nsc-code*

*invariant-nsc checked-nsc to-ns from-ns*

**rewrites**

*init-cs init-nsc-code to-ns = init-simplex' and*

*backtrack-cs backtrack-nsc-code = backtrack-simplex' and*

```

checkpoint-cs checkpoint-nsc-code = checkpoint-simplex' and
check-cs check-nsc-code = check-simplex' and
assert-cs assert-nsc-code = assert-simplex' and
solution-cs solution-nsc-code from-ns = solution-simplex' and
Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex.invariant-cs in-
variant-nsc to-ns = invariant-simplex' and
Incremental-NS-Constraint-Ops-To-Ns-For-Incremental-Simplex.checked-cs checked-nsc
to-ns = checked-simplex'
⟨proof⟩

```

### 7.4.2 An implementation which encapsulates the state

In principle, we now already have a complete implementation of the incremental simplex algorithm with *init-simplex'*, *assert-simplex'*, etc. However, this implementation results in code where the 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) ⇒ ((create-map as, trans-v,
remdups ui), init-state t)))

```

```

fun assert-simplex where assert-simplex i (Simplex-State (cs, (asi, tv, ui), s)) =
  (if i ∈ set ui then Inl [i] else case assert-all-s (list-map-to-fun asi i) s of Inl y ⇒
Inl y | Inr s' ⇒ Inr (Simplex-State (cs, (asi, tv, ui), s')))

```

```

fun check-simplex where
  check-simplex (Simplex-State (cs, asi-tv, s)) = (case check-s s of Inl y ⇒ Inl y |
Inr s' ⇒ Inr (Simplex-State (cs, asi-tv, s')))

```

```

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)) = Sim-
plex-State (cs, asi-tv, backtrack-s c s)

```

### 7.4.3 Soundness of the incremental simplex implementation

First link the unprimed constants against their primed counterparts.



**lemma** *init-simplex'*: *init-simplex cs = Simplex-State (init-simplex' cs)*  
⟨proof⟩

**lemma** *assert-simplex'*: *assert-simplex i (Simplex-State s) = map-sum id Simplex-State (assert-simplex' i s)*  
⟨proof⟩

**lemma** *check-simplex'*: *check-simplex (Simplex-State s) = map-sum id Simplex-State (check-simplex' s)*  
⟨proof⟩

**lemma** *solution-simplex'*: *solution-simplex (Simplex-State s) = solution-simplex' s*  
⟨proof⟩

**lemma** *checkpoint-simplex'*: *checkpoint-simplex (Simplex-State s) = Simplex-Checkpoint (checkpoint-simplex' s)*  
⟨proof⟩

**lemma** *backtrack-simplex'*: *backtrack-simplex (Simplex-Checkpoint c) (Simplex-State s) = Simplex-State (backtrack-simplex' c s)*  
⟨proof⟩

**fun** *invariant-simplex where*  
*invariant-simplex cs J (Simplex-State s) = invariant-simplex' cs J s*

**fun** *checked-simplex where*  
*checked-simplex cs J (Simplex-State s) = checked-simplex' cs J s*

Hide implementation

**declare** *init-simplex.simps[simp del]*  
**declare** *assert-simplex.simps[simp del]*  
**declare** *check-simplex.simps[simp del]*  
**declare** *solution-simplex.simps[simp del]*  
**declare** *checkpoint-simplex.simps[simp del]*  
**declare** *backtrack-simplex.simps[simp del]*

Soundness lemmas

**lemma** *init-simplex: checked-simplex cs {} (init-simplex cs)*  
⟨proof⟩

**lemma** *assert-simplex-ok:*  
*invariant-simplex cs J s  $\implies$  assert-simplex j s = Inr s'  $\implies$  invariant-simplex cs (insert j J) s'*  
⟨proof⟩

**lemma** *assert-simplex-unsat:*  
*invariant-simplex cs J s  $\implies$  assert-simplex j s = Inl I  $\implies$  set I  $\subseteq$  insert j J  $\wedge$  minimal-unsat-core (set I) cs*

*<proof>*

**lemma** *check-simplex-ok:*

*invariant-simplex cs J s  $\implies$  check-simplex s = Inr s'  $\implies$  checked-simplex cs J s'*

*<proof>*

**lemma** *check-simplex-unsat:*

*invariant-simplex cs J s  $\implies$  check-simplex s = Unsat I  $\implies$*

*set I  $\subseteq$  J  $\wedge$  minimal-unsat-core (set I) cs*

*<proof>*

**lemma** *solution-simplex:*

*checked-simplex cs J s  $\implies$  solution-simplex s = v  $\implies$  (J, v)  $\models_{ics}$  set cs*

*<proof>*

**lemma** *backtrack-simplex:*

*checked-simplex cs J s  $\implies$*

*checkpoint-simplex s = c  $\implies$*

*invariant-simplex cs K s'  $\implies$*

*backtrack-simplex c s' = s''  $\implies$*

*J  $\subseteq$  K  $\implies$*

*invariant-simplex cs J s''*

*<proof>*

**lemma** *checked-invariant-simplex:*

*checked-simplex cs J s  $\implies$  invariant-simplex cs J s*

*<proof>*

**declare** *checked-simplex.simps[simp del]*

**declare** *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  $\Rightarrow$  'i simplex-state  $\Rightarrow$  '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  $\Rightarrow$  Unsat I*

*| Inr s'  $\Rightarrow$  assert-all-simplex J s')*

**lemma** *assert-all-simplex-ok: invariant-simplex cs J s  $\implies$  assert-all-simplex K s*

*= Inr s'  $\implies$*

*invariant-simplex cs (J  $\cup$  set K) s'*

*<proof>*

**lemma** *assert-all-simplex-unsat: invariant-simplex cs J s  $\implies$  assert-all-simplex K*

*s = Unsat I  $\implies$*

*set I  $\subseteq$  set K  $\cup$  J  $\wedge$  minimal-unsat-core (set I) cs*

*<proof>*

The collection of soundness lemmas for the incremental simplex algorithm.

```

lemmas incremental-simplex =
  init-simplex
  assert-simplex-ok
  assert-simplex-unsat
  assert-all-simplex-ok
  assert-all-simplex-unsat
  check-simplex-ok
  check-simplex-unsat
  solution-simplex
  backtrack-simplex
  checked-invariant-simplex

```

## 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, GTPP (lp-monom 2 1) (lp-monom 1 2)), —  $2x_1 > x_2$ 
  (3, EQPP (lp-monom 1 1) (lp-monom 2 2)), —  $x_1 = 2x_2$ 
  (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  $\Rightarrow$  s | Unsat -  $\Rightarrow$  undefined); —
assert 1,2,3
  s3 = (case check-simplex s2 of Inr s  $\Rightarrow$  s | Unsat -  $\Rightarrow$  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  $\Rightarrow$  s | Unsat -  $\Rightarrow$  undefined); — assert 4
  I = (case check-simplex s4 of Unsat I  $\Rightarrow$  I | Inr -  $\Rightarrow$  undefined); — checking
detects unsat-core 1,3,4
  s5 = backtrack-simplex c123 s4; — backtrack to constraints 1,2,3
  s6 = (case assert-all-simplex [5,6] s5 of Inr s  $\Rightarrow$  s | Unsat -  $\Rightarrow$  undefined); —
assert 5,6
  s7 = (case check-simplex s6 of Inr s  $\Rightarrow$  s | Unsat -  $\Rightarrow$  undefined); — check that
1,2,3,5,6 are sat.
  sol = solution-simplex s7 — 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.