

A Preprocessor for Linear Diophantine Equalities and Inequalities

René Thiemann

University of Innsbruck, Austria

May 20, 2026

Abstract

We formalize a combination algorithm to preprocess a set of linear diophantine equations and inequalities. It consists of three techniques that are applied exhaustively.

- Pugh’s technique of tightening linear inequalities [4],
- Bromberger and Weidenbach’s algorithm to detect implicit equalities [1] – here we make use of an incremental implementation of the simplex algorithm [3], and
- Griggio’s diophantine equation solver [2] to eliminate all detected equations.

In total, given some linear input constraints, the preprocessor will either detect unsatisfiability in \mathbb{Z} , or it returns equi-satisfiable inequalities, which moreover are all strictly satisfiable in \mathbb{Q} .

Contents

1	Linear Polynomials	2
1.1	An Abstract Type for Multivariate Linear Polynomials	2
1.2	An Implementation of Linear Polynomials as Ordered Association Lists	5
2	Linear Diophantine Equations and Inequalities	11
3	Tightening	12
4	Linear Diophantine Equation Solver	13
4.1	Abstract Algorithm	13
4.2	Executable Algorithm	17

5	Detection of Implicit Equalities	21
5.1	Main Abstract Reasoning Step	21
5.2	Algorithm to Detect all Implicit Equalities in \mathbb{Q}	22
5.3	Algorithm to Detect Implicit Equalities in \mathbb{Z}	28
6	A Combined Preprocessor	29
7	Examples	31

1 Linear Polynomials

1.1 An Abstract Type for Multivariate Linear Polynomials

```

theory Linear-Polynomial
  imports
    Main
  begin

  typedef (overloaded) ('a :: zero,'v) lpoly = { c :: 'v option  $\Rightarrow$  'a. finite {v. c v  $\neq$  0}}
    <proof>

  setup-lifting type-definition-lpoly

  instantiation lpoly :: (ab-group-add,type)ab-group-add
  begin

  lift-definition uminus-lpoly :: ('a, 'b) lpoly  $\Rightarrow$  ('a, 'b) lpoly is  $\lambda$  c x. - c x <proof>

  lift-definition minus-lpoly :: ('a, 'b) lpoly  $\Rightarrow$  ('a, 'b) lpoly  $\Rightarrow$  ('a, 'b) lpoly is  $\lambda$  c1
  c2 x. c1 x - c2 x
  <proof>

  lift-definition plus-lpoly :: ('a, 'b) lpoly  $\Rightarrow$  ('a, 'b) lpoly  $\Rightarrow$  ('a, 'b) lpoly is  $\lambda$  c1
  c2 x. c1 x + c2 x
  <proof>

  lift-definition zero-lpoly :: ('a, 'b) lpoly is  $\lambda$  c. 0 <proof>

  instance <proof>
  end

  lift-definition var-l :: 'v  $\Rightarrow$  ('a :: {comm-monoid-mult,zero-neq-one}, 'v) lpoly is
   $\lambda$  x. ( $\lambda$  c. 0)(Some x := 1) <proof>
  lift-definition constant-l :: ('a :: zero, 'v) lpoly  $\Rightarrow$  'a is  $\lambda$  c. c None <proof>
  lift-definition coeff-l :: ('a :: zero, 'v) lpoly  $\Rightarrow$  'v  $\Rightarrow$  'a is  $\lambda$  c x. c (Some x) <proof>

  lift-definition vars-l :: ('a :: zero, 'v) lpoly  $\Rightarrow$  'v set is  $\lambda$  c. { x. c (Some x)  $\neq$  0}
  <proof>

```

lemma *vars-l-conv-coeff-l*: $\text{vars-l } p = \{x. \text{coeff-l } p \ x \neq 0\}$
<proof>

lemma *finite-vars-l[simp,intro]*: $\text{finite } (\text{vars-l } p)$
<proof>

type-synonym (*'a','v*) *assign* = *'v* \Rightarrow *'a*

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

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

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

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

lemma *vars-l-zero[simp]*: $\text{vars-l } 0 = \{\}$
<proof>

definition *eval-l* :: (*'a* :: *comm-ring*, *'v*) *assign* \Rightarrow (*'a','v*) *lpoly* \Rightarrow *'a* **where**
 $\text{eval-l } \alpha \ p = \text{constant-l } p + \text{sum } (\lambda \ x. \text{coeff-l } p \ x * \alpha \ x) (\text{vars-l } p)$

lemma *eval-l-mono*: **assumes** $\text{finite } V \ \text{vars-l } p \subseteq V$
shows $\text{eval-l } \alpha \ p = \text{constant-l } p + \text{sum } (\lambda \ x. \text{coeff-l } p \ x * \alpha \ x) \ V$
<proof>

lemma *eval-l-cong*: **assumes** $\bigwedge \ x. x \in \text{vars-l } p \implies \alpha \ x = \beta \ x$
shows $\text{eval-l } \alpha \ p = \text{eval-l } \beta \ p$
<proof>

lemma *eval-l-0[simp]*: $\text{eval-l } \alpha \ 0 = 0$ *<proof>*

lemma *eval-l-plus[simp]*: $\text{eval-l } \alpha \ (p1 + p2) = \text{eval-l } \alpha \ p1 + \text{eval-l } \alpha \ p2$
<proof>

lemma *eval-l-minus[simp]*: $\text{eval-l } \alpha \ (p1 - p2) = \text{eval-l } \alpha \ p1 - \text{eval-l } \alpha \ p2$
<proof>

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

lemma *eval-l-var[simp]*: $\text{eval-l } \alpha \ (\text{var-l } x) = \alpha \ x$
<proof>

lift-definition $substitute-l :: 'v \Rightarrow ('a :: comm-ring, 'v) lpoly \Rightarrow ('a, 'v) lpoly \Rightarrow ('a, 'v) lpoly$ **is**

$\lambda x p q y. (q(Some\ x := 0))\ y + q\ (Some\ x) * p\ y$
 $\langle proof \rangle$

lemma $vars-substitute-l: vars-l\ (substitute-l\ x\ p\ q) \subseteq vars-l\ p \cup (vars-l\ q - \{x\})$
 $\langle proof \rangle$

lemma $substitute-l-id: x \notin vars-l\ q \Longrightarrow substitute-l\ x\ p\ q = q$
 $\langle proof \rangle$

lemma $eval-substitute-l: eval-l\ \alpha\ (substitute-l\ x\ p\ q) = eval-l\ (\alpha\ (x := eval-l\ \alpha\ p))\ q$
 $\langle proof \rangle$

lift-definition $fun-of-lpoly :: ('a :: zero, 'v) lpoly \Rightarrow 'v\ option \Rightarrow 'a$ **is** $\lambda x. x$ $\langle proof \rangle$

lift-definition $smult-l :: 'a :: comm-ring \Rightarrow ('a, 'v)lpoly \Rightarrow ('a, 'v)lpoly$ **is**
 $\lambda y\ c\ z. y * c\ z$
 $\langle proof \rangle$

lemma $coeff-smult-l[simp]: coeff-l\ (smult-l\ c\ p)\ x = c * coeff-l\ p\ x$
 $\langle proof \rangle$

lemma $constant-smult-l[simp]: constant-l\ (smult-l\ c\ p) = c * constant-l\ p$
 $\langle proof \rangle$

lemma $eval-smult-l[simp]: eval-l\ \alpha\ (smult-l\ c\ p) = c * eval-l\ \alpha\ p$
 $\langle proof \rangle$

lift-definition $const-l :: 'a :: zero \Rightarrow ('a, 'v) lpoly$ **is** $\lambda c. (\lambda z. 0)(None := c)$
 $\langle proof \rangle$

lemma $eval-l-const-l-constant: eval-l\ \alpha\ (const-l\ (constant-l\ p)) = constant-l\ p$
 $\langle proof \rangle$

definition $substitute-all-l :: ('v \Rightarrow ('a, 'w) lpoly) \Rightarrow ('a :: comm-ring, 'v) lpoly \Rightarrow ('a, 'w) lpoly$ **where**
 $substitute-all-l\ \sigma\ p = (const-l\ (constant-l\ p) + sum\ (\lambda x. smult-l\ (coeff-l\ p\ x)\ (\sigma\ x))\ (vars-l\ p))$

lemma $eval-substitute-all-l: eval-l\ \alpha\ (substitute-all-l\ \sigma\ p) = eval-l\ (\lambda x. eval-l\ \alpha\ (\sigma\ x))\ p$
 $\langle proof \rangle$

lift-definition $sdiv-l :: (int, 'v) lpoly \Rightarrow int \Rightarrow (int, 'v) lpoly$ **is** $\lambda c\ q\ x. c\ x\ div\ q$
 $\langle proof \rangle$

definition $\text{vars-l-list } p = \text{sorted-list-of-set } (\text{vars-l } p)$

lemma $\text{vars-l-list}[simp]: \text{set } (\text{vars-l-list } p) = \text{vars-l } p$
 $\langle \text{proof} \rangle$

definition $\text{min-var} :: ('a :: \{\text{linorder}, \text{ordered-ab-group-add-abs}\}, 'v :: \text{linorder})$
 $\text{lpoly} \Rightarrow 'v$ **where**
 $\text{min-var } p = (\text{let}$
 $\quad \text{xcs} = \text{map } (\lambda x. (x, \text{coeff-l } p \ x)) (\text{vars-l-list } p);$
 $\quad \text{axcs} = \text{map } (\text{map-prod id abs}) \text{xcs};$
 $\quad \text{m} = \text{min-list } (\text{map snd axcs})$
 $\quad \text{in } (\text{case filter } (\lambda xa. \text{snd } xa = \text{m}) \text{axcs of}$
 $\quad (x, a) \# _ \Rightarrow x))$

lemma $\text{min-var}: \text{vars-l } p \neq \{\}$ $\implies \text{coeff-l } p (\text{min-var } p) \neq 0$
 $x \in \text{vars-l } p \implies \text{abs } (\text{coeff-l } p (\text{min-var } p)) \leq \text{abs } (\text{coeff-l } p \ x)$
 $\langle \text{proof} \rangle$

definition $\text{gcd-coeffs-l} :: ('a :: \text{Gcd}, 'v) \text{lpoly} \Rightarrow 'a$ **where**
 $\text{gcd-coeffs-l } p = \text{Gcd } (\text{coeff-l } p \ ` \ \text{vars-l } p)$

lift-definition $\text{change-const} :: 'a :: \text{zero} \Rightarrow ('a, 'v) \text{lpoly} \Rightarrow ('a, 'v) \text{lpoly}$ **is** $\lambda x \ c.$
 $c(\text{None} := x)$
 $\langle \text{proof} \rangle$

lemma $\text{lpoly-fun-of-eqI}: \text{assumes } \bigwedge x. \text{fun-of-lpoly } p \ x = \text{fun-of-lpoly } q \ x$
shows $p = q$
 $\langle \text{proof} \rangle$

lift-definition $\text{reorder-nontriv-var} :: 'v \Rightarrow (\text{int}, 'v) \text{lpoly} \Rightarrow 'v \Rightarrow (\text{int}, 'v) \text{lpoly}$ **is**
 $\lambda x \ c \ y. (\lambda z. c \ z \ \text{div } c \ (\text{Some } x)) (\text{Some } x := 1, \text{Some } y := -1)$
 $\langle \text{proof} \rangle$

lemma $\text{coeff-l-reorder-nontriv-var}: \text{coeff-l } (\text{reorder-nontriv-var } x \ p \ y)$
 $= (\lambda z. \text{coeff-l } p \ z \ \text{div } \text{coeff-l } p \ x) (x := 1, y := -1)$
 $\langle \text{proof} \rangle$

lemma $\text{vars-reorder-non-triv}: \text{vars-l } (\text{reorder-nontriv-var } x \ p \ y) \subseteq \text{insert } x \ (\text{insert}$
 $\ y \ (\text{vars-l } p))$
 $\langle \text{proof} \rangle$

end

1.2 An Implementation of Linear Polynomials as Ordered Association Lists

theory $\text{Linear-Polynomial-Impl}$
imports

HOL-Library.AList
Linear-Polynomial

begin

typedef (overloaded) (*'a* :: *zero*, *'v* :: *linorder*) *lpoly-impl* =
 { (*c* :: *'a*, *vcs* :: (*'v* × *'a*) *list*).
 sorted (*map fst vcs*) ∧
 distinct (*map fst vcs*) ∧
 Ball (*snd* ' *set vcs*) ((≠) 0)}
 ⟨*proof*⟩

setup-lifting *type-definition-lpoly-impl*

definition *lookup-0* :: (*'a* × *'b* :: *zero*)*list* ⇒ *'a* ⇒ *'b* **where**
lookup-0 *xs* *x* = (*case map-of xs x of None* ⇒ 0 | *Some y* ⇒ *y*)

lemma *lookup-0-empty[simp]*: *lookup-0* [] = (λ *x*. 0)
 ⟨*proof*⟩

lemma *lookup-0-single[simp]*: *lookup-0* [(*x*,*c*)] = (λ *y*. 0)(*x* := *c*)
 ⟨*proof*⟩

lemma *finite-lookup-0[simp, intro]*: *finite* {*x* . *lookup-0* *xs* *x* ≠ 0}
 ⟨*proof*⟩

lift-definition *lpoly-of* :: (*'a* :: *zero*, *'v* :: *linorder*) *lpoly-impl* ⇒ (*'a*,*'v*)*lpoly* **is**
 λ (*c*, *vcs*) *cx*. *case cx of None* ⇒ *c* | *Some x* ⇒ *lookup-0 vcs x*
 ⟨*proof*⟩

code-datatype *lpoly-of*

lift-definition *zero-lpoly-impl* :: (*'a* :: *zero*, *'v* :: *linorder*) *lpoly-impl* **is**
 (0,[]) ⟨*proof*⟩

lemma *zero-lpoly-impl[code]*: 0 = *lpoly-of zero-lpoly-impl*
 ⟨*proof*⟩

lift-definition *const-lpoly-impl* :: *'a* ⇒ (*'a* :: *zero*, *'v* :: *linorder*) *lpoly-impl* **is**
 λ *c*. (*c*,[]) ⟨*proof*⟩

lemma *const-lpoly-impl[code]*: *const-l c* = *lpoly-of (const-lpoly-impl c)*
 ⟨*proof*⟩

lift-definition *constant-lpoly-impl* :: (*'a* :: *zero*, *'v* :: *linorder*) *lpoly-impl* ⇒ *'a* **is**
fst ⟨*proof*⟩

lemma *constant-lpoly-impl[code]*: *constant-l (lpoly-of p)* = *constant-lpoly-impl p*
 ⟨*proof*⟩

lift-definition *var-lpoly-impl* :: 'v :: linorder \Rightarrow ('a :: {comm-monoid-mult, zero-neq-one}, 'v) *lpoly-impl* **is**
 $\lambda x. (0, [(x,1)])$ *<proof>*

lemma *var-lpoly-impl[code]*: *var-l* *x* = *lpoly-of* (*var-lpoly-impl* *x*)
<proof>

lift-definition *uminus-lpoly-impl* :: ('a :: ab-group-add, 'v :: linorder) *lpoly-impl* \Rightarrow ('a, 'v) *lpoly-impl* **is**
 $\lambda (c, vcs). (uminus\ c, map\ (map\ prod\ id\ uminus)\ vcs)$
<proof>

lemma *uminus-lpoly-impl[code]*: $-$ *lpoly-of* *p* = *lpoly-of* (*uminus-lpoly-impl* *p*)
<proof>

fun *merge-coeffs-main* :: ('a :: zero \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('v :: linorder \times 'a) list \Rightarrow ('v \times 'a) list \Rightarrow ('v \times 'a) list **where**
 $merge\ coeffs\ main\ f\ ((x,c)\ \#)\ xs\ ((y,d)\ \#)\ ys =$
 $\quad if\ x = y\ then\ (x,f\ c\ d)\ \# merge\ coeffs\ main\ f\ xs\ ys$
 $\quad else\ if\ x < y\ then\ (x,f\ c\ 0)\ \# merge\ coeffs\ main\ f\ xs\ ((y,d)\ \#)\ ys$
 $\quad else\ (y,f\ 0\ d)\ \# merge\ coeffs\ main\ f\ ((x,c)\ \#)\ xs\ ys$
 $| merge\ coeffs\ main\ f\ []\ ys = map\ (map\ prod\ id\ (f\ 0))\ ys$
 $| merge\ coeffs\ main\ f\ xs\ [] = map\ (map\ prod\ id\ (\lambda x. f\ x\ 0))\ xs$

lemma *merge-coeffs-main*: **assumes** *sorted* (*map* *fst* *vxs*) *distinct* (*map* *fst* *vxs*)
sorted (*map* *fst* *vys*) *distinct* (*map* *fst* *vys*)
and *f* 0 0 = 0

shows *sorted* (*map* *fst* (*merge-coeffs-main* *f* *vxs* *vys*))
 \wedge *distinct* (*map* *fst* (*merge-coeffs-main* *f* *vxs* *vys*))
 \wedge *fst* ' set (*merge-coeffs-main* *f* *vxs* *vys*) = *fst* ' set *vxs* \cup *fst* ' set *vys*
 \wedge *lookup-0* (*merge-coeffs-main* *f* *vxs* *vys*) *x* = *f* (*lookup-0* *vxs* *x*) (*lookup-0* *vys* *x*)
<proof>

definition *filter-0* **where** *filter-0* = *filter* ($\lambda p. snd\ p \neq 0$)

lemma *filter-0*: **assumes** *distinct* (*map* *fst* *xs*) *sorted* (*map* *fst* *xs*)

shows *lookup-0* (*filter-0* *xs*) = *lookup-0* *xs*
distinct (*map* *fst* (*filter-0* *xs*))
sorted (*map* *fst* (*filter-0* *xs*))
Ball (*snd* ' set (*filter-0* *xs*)) ((\neq) 0)
<proof>

definition *merge-coeffs* :: ('a :: zero \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('v :: linorder \times 'a) list \Rightarrow ('v \times 'a) list \Rightarrow ('v \times 'a) list **where**
 $merge\ coeffs\ f\ xs\ ys = filter\ 0\ (merge\ coeffs\ main\ f\ xs\ ys)$

lemma *merge-coeffs*: **assumes** *sorted* (*map* *fst* *vxs*) *distinct* (*map* *fst* *vxs*)
sorted (*map* *fst* *vys*) *distinct* (*map* *fst* *vys*)

and $f\ 0\ 0 = 0$
shows *sorted* (*map fst* (*merge-coeffs f vxs vys*)) (**is** ?A)
distinct (*map fst* (*merge-coeffs f vxs vys*)) (**is** ?B)
Ball (*snd* ‘*set* (*merge-coeffs f vxs vys*)) ($\neq 0$) (**is** ?C)
lookup-0 (*merge-coeffs f vxs vys*) $x = f$ (*lookup-0 vxs x*) (*lookup-0 vys x*) (**is** ?D)
 ⟨*proof*⟩

lift-definition *minus-lpoly-impl* :: ($'a :: ab\text{-group-add}, 'v :: linorder$) *lpoly-impl* \Rightarrow
 ($'a, 'v$) *lpoly-impl* \Rightarrow ($'a, 'v$) *lpoly-impl* **is**
 $\lambda (c, vxs) (d, vys). (c - d, \text{merge-coeffs minus } vxs vys)$
 ⟨*proof*⟩

lemma *minus-lpoly-impl[code]*: *lpoly-of p* - *lpoly-of q* = *lpoly-of* (*minus-lpoly-impl p q*)
 ⟨*proof*⟩

lift-definition *plus-lpoly-impl* :: ($'a :: ab\text{-group-add}, 'v :: linorder$) *lpoly-impl* \Rightarrow
 ($'a, 'v$) *lpoly-impl* \Rightarrow ($'a, 'v$) *lpoly-impl* **is**
 $\lambda (c, vxs) (d, vys). (c + d, \text{merge-coeffs plus } vxs vys)$
 ⟨*proof*⟩

lemma *plus-lpoly-impl[code]*: *lpoly-of p* + *lpoly-of q* = *lpoly-of* (*plus-lpoly-impl p q*)
 ⟨*proof*⟩

lift-definition *map-lpoly-impl* :: ($'a :: zero \Rightarrow 'a$) \Rightarrow ($'a, 'v :: linorder$) *lpoly-impl*
 \Rightarrow ($'a, 'v$) *lpoly-impl* **is**
 $\lambda f (c, vcs). (f\ c, \text{filter-0 } (\text{map } (\text{map-prod id } f) vcs))$
 ⟨*proof*⟩

lemma *map-lpoly-impl*: $f\ 0 = 0 \implies \text{fun-of-lpoly } (\text{lpoly-of } (\text{map-lpoly-impl } f\ p)) =$
 $(\lambda x. f (\text{fun-of-lpoly } (\text{lpoly-of } p) x))$
 ⟨*proof*⟩

definition *sdiv-lpoly-impl p x* = *map-lpoly-impl* ($\lambda y. y\ \text{div } x$) *p*

lemma *sdiv-lpoly-impl[code]*: *sdiv-l* (*lpoly-of p*) $x = \text{lpoly-of } (\text{sdiv-lpoly-impl } p\ x)$
 ⟨*proof*⟩

definition *smult-lpoly-impl x p* = *map-lpoly-impl* ($(*)\ x$) *p*

lemma *smult-lpoly-impl[code]*: *smult-l* x (*lpoly-of p*) = *lpoly-of* (*smult-lpoly-impl x p*)
 ⟨*proof*⟩

instantiation *lpoly* :: (*type, type*) *equal* **begin**

definition *equal-lpoly* :: ($'a, 'b$) *lpoly* \Rightarrow ($'a, 'b$) *lpoly* \Rightarrow *bool* **where** *equal-lpoly* =
 (=)

instance

<proof>
end

instantiation *lpoly-impl* :: (zero,linorder)equal **begin**
lift-definition *equal-lpoly-impl* :: ('a, 'b) *lpoly-impl* \Rightarrow ('a, 'b) *lpoly-impl* \Rightarrow bool
is $\lambda (c, xs) (d, ys). c = d \wedge xs = ys$ *<proof>*
instance
<proof>
end

lift-definition *vars-coeffs-impl* :: ('a :: zero, 'v :: linorder) *lpoly-impl* \Rightarrow ('v \times 'a) list **is** *snd* *<proof>*

lemma *vars-coeffs-impl*:
 $set (vars-coeffs-impl p) = (\lambda v. (v, coeff-l (lpoly-of p) v)) \text{ 'vars-l } (lpoly-of p) \text{ (is ?A)}$
 $distinct (map fst (vars-coeffs-impl p)) \text{ (is ?B)}$
 $sorted (map fst (vars-coeffs-impl p)) \text{ (is ?C)}$
 $vars-l-list (lpoly-of p) = map fst (vars-coeffs-impl p) \text{ (is ?D)}$
 $vars-coeffs-impl p = map (\lambda v. (v, coeff-l (lpoly-of p) v)) (vars-l-list (lpoly-of p)) \text{ (is ?E)}$
<proof>

declare *vars-coeffs-impl(4)*[code]

lemma *eval-lpoly-impl*[code]: $eval-l \alpha (lpoly-of p) = constant-lpoly-impl p + (\sum (x, c) \leftarrow vars-coeffs-impl p. c * \alpha x)$
<proof>

lemma *substitute-all-impl*[code]: $substitute-all-l \sigma (lpoly-of p) = const-l (constant-lpoly-impl p) + (\sum (x, c) \leftarrow vars-coeffs-impl p. smult-l c (\sigma x))$
<proof>

lemma *equal-lpoly-impl*[code]: $HOL.equal (lpoly-of p) (lpoly-of q) = (p = q)$
<proof>

fun *update-main* :: 'v :: linorder \Rightarrow 'a :: zero \Rightarrow ('v \times 'a) list \Rightarrow ('v \times 'a) list
where
 $update-main x a ((y,b) \# zs) = (if x > y then (y,b) \# update-main x a zs$
 $else if x = y then (y, a) \# zs else (x,a) \# (y, b) \# zs)$
 $| update-main x a [] = [(x,a)]$

lemma *update-main*: **assumes** $sorted (map fst vcs) distinct (map fst vcs) Ball (snd \text{ ' set } vcs) ((\neq) 0)$
and $vcs' = update-main x a vcs$
and $a: a \neq 0$
shows $sorted (map fst vcs') distinct (map fst vcs') Ball (snd \text{ ' set } vcs') ((\neq) 0)$
 $fst \text{ ' set } vcs' = insert x (fst \text{ ' set } vcs)$

$lookup-0\ vcs'\ z = ((lookup-0\ vcs)(x := a))\ z$
 ⟨proof⟩

fun $update-main-0 :: 'v :: linorder \Rightarrow ('v \times 'a)\ list \Rightarrow ('v \times 'a)\ list$ **where**
 $update-main-0\ x\ ((y,b) \# zs) = (if\ x > y\ then\ (y,b) \# update-main-0\ x\ zs$
 $else\ if\ x = y\ then\ zs\ else\ (y, b) \# zs)$
 | $update-main-0\ x\ [] = []$

lemma $update-main-0$: **assumes** $sorted\ (map\ fst\ vcs)\ distinct\ (map\ fst\ vcs)\ Ball$
 $(snd\ 'set\ vcs)\ ((\neq)\ 0)$
and $vcs' = update-main-0\ x\ vcs$
shows $sorted\ (map\ fst\ vcs')\ distinct\ (map\ fst\ vcs')\ Ball\ (snd\ 'set\ vcs')\ ((\neq)\ 0)$
 $fst\ 'set\ vcs' = fst\ 'set\ vcs - \{x\}$
 $lookup-0\ vcs'\ z = ((lookup-0\ vcs)(x := 0))\ z$
 ⟨proof⟩

lift-definition $update-lpoly-impl :: 'v :: linorder \Rightarrow 'a :: zero \Rightarrow ('a, 'v)lpoly-impl$
 $\Rightarrow ('a, 'v)lpoly-impl$ **is**
 $\lambda\ x\ a\ (c, vs). if\ a = 0\ then\ (c, update-main-0\ x\ vs)\ else\ (c, update-main\ x\ a\ vs)$
 ⟨proof⟩

lemma $update-lpoly-impl$: $fun-of-lpoly\ (lpoly-of\ (update-lpoly-impl\ x\ a\ p)) = (fun-of-lpoly$
 $(lpoly-of\ p))(Some\ x := a)$
 ⟨proof⟩

lift-definition $coeff-lpoly-impl :: ('a :: zero, 'v :: linorder)lpoly-impl \Rightarrow 'v \Rightarrow 'a$ **is**
 $\lambda\ (c,p)\ x. lookup-0\ p\ x$ ⟨proof⟩

lemma $coeff-lpoly-impl[code]$: $coeff-l\ (lpoly-of\ p)\ x = coeff-lpoly-impl\ p\ x$
 ⟨proof⟩

definition $substitute-l-impl$ **where**
 $substitute-l-impl\ x\ p\ q = (let\ c = coeff-lpoly-impl\ q\ x\ in$
 $plus-lpoly-impl\ (update-lpoly-impl\ x\ 0\ q)\ (smult-lpoly-impl\ c\ p))$

lemma $substitute-l-impl[code]$:
 $substitute-l\ x\ (lpoly-of\ p)\ (lpoly-of\ q) = lpoly-of\ (substitute-l-impl\ x\ p\ q)$
 ⟨proof⟩

definition $reorder-nontriv-var-impl$ **where**
 $reorder-nontriv-var-impl\ x\ p\ y = (let\ c = coeff-lpoly-impl\ p\ x$
 $in\ update-lpoly-impl\ y\ (-1)\ (update-lpoly-impl\ x\ 1\ (sdiv-lpoly-impl\ p\ c)))$

lemma $reorder-nontriv-var-impl[code]$:
 $reorder-nontriv-var\ x\ (lpoly-of\ p)\ y = lpoly-of\ (reorder-nontriv-var-impl\ x\ p\ y)$
 ⟨proof⟩

lemmas $min-var-impl = min-var-def$ [of $lpoly-of\ p$ for p ,

folded vars-coeffs-impl(5)]

declare *min-var-impl*[code]

lemma *Gcd-set*: $Gcd (set (xs :: 'a :: semiring-Gcd list)) = gcd-list xs$
<proof>

lemma *gcd-coeffs-impl*[code]:
 $gcd-coeffs-l (lpoly-of (p :: ('a :: semiring-Gcd, -)lpoly-impl)) = fold gcd (map snd (vars-coeffs-impl p)) 0$
<proof>

lift-definition *change-const-impl* :: $'a \Rightarrow ('a :: zero, 'v :: linorder)lpoly-impl \Rightarrow ('a, 'v)lpoly-impl$
is $\lambda c (d, vs). (c, vs)$ *<proof>*

lemma *change-const-impl*[code]: $change-const c (lpoly-of p) = lpoly-of (change-const-impl c p)$
<proof>

end

2 Linear Diophantine Equations and Inequalities

We just represent equations and inequalities as polynomials, i.e., $p = 0$ or $p \leq 0$. There is no need for strict inequalities $p < 0$ since for integers this is equivalent to $p + 1 \leq 0$.

theory *Diophantine-Eqs-and-Ineqs*
imports *Linear-Polynomial*
begin

type-synonym $'v dleq = (int, 'v) lpoly$
type-synonym $'v dlineq = (int, 'v) lpoly$

definition *satisfies-dleq* :: $(int, 'v) assign \Rightarrow 'v dleq \Rightarrow bool$ **where**
 $satisfies-dleq \alpha p = (eval-l \alpha p = 0)$

definition *satisfies-dlineq* :: $(int, 'v) assign \Rightarrow 'v dlineq \Rightarrow bool$ **where**
 $satisfies-dlineq \alpha p = (eval-l \alpha p \leq 0)$

abbreviation *satisfies-eq-ineqs* :: $(int, 'v) assign \Rightarrow 'v dleq set \Rightarrow 'v dlineq set \Rightarrow bool$ $(\langle \cdot \models_{dio} \langle \cdot, \cdot \rangle \rangle)$ **where**
 $satisfies-eq-ineqs \alpha eqs ineqs \equiv Ball eqs (satisfies-dleq \alpha) \wedge Ball ineqs (satisfies-dlineq \alpha)$

definition *trivial-ineq* :: $(int, 'v :: linorder)lpoly \Rightarrow bool option$ **where**
 $trivial-ineq c = (if vars-l-list c = [] then Some (constant-l c \leq 0) else None)$

lemma *trivial-ineq-None*: $\text{trivial-ineq } c = \text{None} \implies \text{vars-l } c \neq \{\}$
 ⟨proof⟩

lemma *trivial-ineq-Some*: **assumes** $\text{trivial-ineq } c = \text{Some } b$
shows $b = \text{satisfies-dlineq } \alpha \ c$
 ⟨proof⟩

fun *trivial-ineq-filter* :: $'v :: \text{linorder dlineq list} \Rightarrow 'v \text{ dlineq list option}$
where $\text{trivial-ineq-filter } [] = \text{Some } []$
 $| \text{trivial-ineq-filter } (c \# cs) = (\text{case } \text{trivial-ineq } c \text{ of } \text{Some } \text{True} \Rightarrow \text{trivial-ineq-filter } cs$
 $| \text{Some } \text{False} \Rightarrow \text{None}$
 $| \text{None} \Rightarrow \text{map-option } ((\#) \ c) \ (\text{trivial-ineq-filter } cs))$

lemma *trivial-ineq-filter*: $\text{trivial-ineq-filter } cs = \text{None} \implies (\exists \ \alpha. \ \alpha \models_{\text{dio}} (\{\}, \text{set } cs))$
 $\text{trivial-ineq-filter } cs = \text{Some } ds \implies$
 $\text{Ball } (\text{set } ds) \ (\lambda \ c. \ \text{vars-l } c \neq \{\}) \wedge$
 $(\alpha \models_{\text{dio}} (\{\}, \text{set } cs) \longleftrightarrow \alpha \models_{\text{dio}} (\{\}, \text{set } ds)) \wedge$
 $\text{length } ds \leq \text{length } cs$
 ⟨proof⟩

lemma *trivial-lhe*: **assumes** $\text{vars-l } p = \{\}$
shows $\text{eval-l } \alpha \ p = \text{constant-l } p$
 $\text{satisfies-dleq } \alpha \ p \longleftrightarrow p = 0$
 ⟨proof⟩

end

3 Tightening

replace $p + c \leq 0$ by $p / g + \lceil c / g \rceil \leq 0$ where c is a constant and g is the gcd of the variable coefficients of p .

theory *Diophantine-Tightening*
imports
Diophantine-Eqs-and-Ineqs
begin

definition *tighten-ineq* :: $'v \text{ dlineq} \Rightarrow 'v \text{ dlineq}$ **where**
 $\text{tighten-ineq } p = (\text{let } g = \text{gcd-coeffs-l } p;$
 $\ c = \text{constant-l } p$
 $\ \text{in if } g = 1 \text{ then } p \text{ else let } d = -((-c) \ \text{div } g)$
 $\ \text{in change-const } d \ (\text{sdiv-l } p \ g))$

lemma *tighten-ineq*: **assumes** $\text{vars-l } p \neq \{\}$
shows $\text{satisfies-dlineq } \alpha \ (\text{tighten-ineq } p) = \text{satisfies-dlineq } \alpha \ p$

<proof>

definition *tighten-ineqs* :: 'v dlineq list \Rightarrow 'v :: linorder dlineq list option **where**
tighten-ineqs cs = map-option (map tighten-ineq) (trivial-ineq-filter cs)

lemma *tighten-ineqs*: *tighten-ineqs cs = None $\implies \nexists \alpha. \alpha \models_{dio} (\{\}, set cs)$*
tighten-ineqs cs = Some ds \implies
($\alpha \models_{dio} (\{\}, set cs) \longleftrightarrow \alpha \models_{dio} (\{\}, set ds)$) \wedge
length ds \leq length cs

<proof>

end

4 Linear Diophantine Equation Solver

We verify Griggio's algorithm to eliminate equations or detect unsatisfiability.

4.1 Abstract Algorithm

theory *Linear-Diophantine-Solver*

imports

Diophantine-Eqs-and-Ineqs

HOL.Map

begin

lift-definition *normalize-dleq* :: 'v dleq \Rightarrow int \times 'v dleq **is**

$\lambda c. (Gcd (range c), \lambda x. c \ x \ div \ Gcd (range c))$

<proof>

lemma *normalize-dleq-gcd*: **assumes** *normalize-dleq p = (g,q)*

and *p \neq 0*

shows *g = Gcd (insert (constant-l p) (coeff-l p ' vars-l p))*

and *g \geq 1*

and *normalize-dleq q = (1,q)*

<proof>

lemma *vars-l-normalize*: *normalize-dleq p = (g,q) \implies vars-l q = vars-l p*

<proof>

lemma *eval-normalize-dleq*: *normalize-dleq p = (g,q) \implies eval-l α p = g * eval-l*

α q

<proof>

lemma *gcd-unsat-detection*: **assumes** $g = \text{Gcd} (\text{coeff-l } p \text{ ' vars-l } p)$
and $\neg g \text{ dvd constant-l } p$
shows $\neg \text{satisfies-dleg } \alpha \text{ } p$
 $\langle \text{proof} \rangle$

lemma *substitute-l-in-equation*: **assumes** $\alpha \text{ } x = \text{eval-l } \alpha \text{ } p$
shows $\text{eval-l } \alpha (\text{substitute-l } x \text{ } p \text{ } q) = \text{eval-l } \alpha \text{ } q$
 $\text{satisfies-dleg } \alpha (\text{substitute-l } x \text{ } p \text{ } q) \longleftrightarrow \text{satisfies-dleg } \alpha \text{ } q$
 $\langle \text{proof} \rangle$

type-synonym $'v \text{ dleg-sf} = 'v \times (\text{int}, 'v) \text{lpoly}$

fun *satisfies-dleg-sf*:: $(\text{int}, 'v) \text{ assign} \Rightarrow 'v \text{ dleg-sf} \Rightarrow \text{bool}$ **where**
 $\text{satisfies-dleg-sf } \alpha (x, p) = (\alpha \text{ } x = \text{eval-l } \alpha \text{ } p)$

type-synonym $'v \text{ dleg-system} = 'v \text{ dleg-sf set} \times 'v \text{ dleg set}$

fun *satisfies-system* :: $(\text{int}, 'v) \text{ assign} \Rightarrow 'v \text{ dleg-system} \Rightarrow \text{bool}$ **where**
 $\text{satisfies-system } \alpha (S, E) = (\text{Ball } S (\text{satisfies-dleg-sf } \alpha) \wedge \text{Ball } E (\text{satisfies-dleg } \alpha))$

fun *invariant-system* :: $'v \text{ dleg-system} \Rightarrow \text{bool}$ **where**
 $\text{invariant-system } (S, E) = (\text{Ball } (\text{fst } ' S) (\lambda x. x \notin \bigcup (\text{vars-l } ' (\text{snd } ' S \cup E))) \wedge (\exists! e. (x, e) \in S))$

definition *reorder-for-var* **where**

$\text{reorder-for-var } x \text{ } p = (\text{if } \text{coeff-l } p \text{ } x = 1 \text{ then } - (p - \text{var-l } x) \text{ else } p + \text{var-l } x)$

lemma *reorder-for-var*: **assumes** $\text{abs } (\text{coeff-l } p \text{ } x) = 1$
shows $\text{satisfies-dleg } \alpha \text{ } p \longleftrightarrow \text{satisfies-dleg-sf } \alpha (x, \text{reorder-for-var } x \text{ } p)$ (**is** *?prop1*)
 $\text{vars-l } (\text{reorder-for-var } x \text{ } p) = \text{vars-l } p - \{x\}$ (**is** *?prop2*)
 $\langle \text{proof} \rangle$

lemma *reorder-nontriv-var-sat*: $\exists a. \text{satisfies-dleg } (\alpha(y := a)) (\text{reorder-nontriv-var } x \text{ } p \text{ } y)$
 $\langle \text{proof} \rangle$

lemma *reorder-nontriv-var*: **assumes** $a: a = \text{coeff-l } p \text{ } x \text{ } a \neq 0$

and $y: y \notin \text{vars-l } p$

and $q: q = \text{reorder-nontriv-var } x \text{ } p \text{ } y$

and $e: e = \text{reorder-for-var } x \text{ } q$

and $r: r = \text{substitute-l } x \text{ } e \text{ } p$

shows $\text{fun-of-lpoly } r = (\lambda z. \text{fun-of-lpoly } p \text{ } z \text{ mod } a)(\text{Some } x := 0, \text{Some } y := a)$

$\text{constant-l } r = \text{constant-l } p \text{ mod } a$

$\text{coeff-l } r = (\lambda z. \text{coeff-l } p \text{ } z \text{ mod } a)(x := 0, y := a)$

$\langle \text{proof} \rangle$

inductive *griggio-equiv-step* :: 'v dleq-system \Rightarrow 'v dleq-system \Rightarrow bool **where**
griggio-solve: $\text{abs } (\text{coeff-l } p \ x) = 1 \implies e = \text{reorder-for-var } x \ p \implies$
griggio-equiv-step (*S*, *insert p E*) (*insert* (*x*, *e*) (*map-prod id* (*substitute-l x e*) ' *S*), *substitute-l x e* ' *E*)
| *griggio-normalize*: $\text{normalize-dleq } p = (g, q) \implies g \geq 1 \implies$
griggio-equiv-step (*S*, *insert p E*) (*S*, *insert q E*)
| *griggio-trivial*: *griggio-equiv-step* (*S*, *insert 0 E*) (*S*, *E*)

fun *vars-system* :: 'v dleq-system \Rightarrow 'v set **where**
vars-system (*S*, *E*) = *fst* ' *S* $\cup \cup$ (*vars-l* ' (*snd* ' *S* \cup *E*))

lemma *griggio-equiv-step*: **assumes** *griggio-equiv-step SE TF*
shows (*satisfies-system* α *SE* \longleftrightarrow *satisfies-system* α *TF*) \wedge
(*invariant-system SE* \longrightarrow *invariant-system TF*) \wedge
vars-system TF \subseteq *vars-system SE*
<*proof*>

inductive *griggio-unsat* :: 'v dleq \Rightarrow bool **where**
griggio-gcd-unsat: $\neg \text{Gcd } (\text{coeff-l } p \ ' \ \text{vars-l } p) \ \text{dvd } \text{constant-l } p \implies \text{griggio-unsat } p$
| *griggio-constant-unsat*: $\text{vars-l } p = \{\} \implies p \neq 0 \implies \text{griggio-unsat } p$

lemma *griggio-unsat*: **assumes** *griggio-unsat p*
shows $\neg \text{satisfies-system } \alpha \ (S, \text{insert } p \ E)$
<*proof*>

definition *adjust-assign* :: 'v dleq-sf list \Rightarrow ('v \Rightarrow int) \Rightarrow ('v \Rightarrow int) **where**
adjust-assign S α *x* = (*case map-of S x of Some p* \Rightarrow *eval-l* α *p* | *None* \Rightarrow α *x*)

definition *solution-subst* :: 'v dleq-sf list \Rightarrow ('v \Rightarrow (int, 'v)lpoly) **where**
solution-subst S x = (*case map-of S x of Some p* \Rightarrow *p* | *None* \Rightarrow *var-l x*)

locale *griggio-input* = **fixes**

V :: 'v :: linorder set **and**
E :: 'v dleq set

begin

fun *invariant-state* **where**

invariant-state (*Some* (*SF*, *X*)) = (*invariant-system SF*
 \wedge *vars-system SF* \subseteq *V* \cup *X*
 \wedge *V* \cap *X* = $\{\}$
 \wedge (\forall α . (*satisfies-system* α *SF* \longrightarrow *Ball E* (*satisfies-dleq* α))
 \wedge (*Ball E* (*satisfies-dleq* α) \longrightarrow (\exists β . *satisfies-system* β *SF* \wedge (\forall *x*. *x* \notin
X \longrightarrow α *x* = β *x*))))))
| *invariant-state None* = (\forall α . \neg *Ball E* (*satisfies-dleq* α))

inductive-set *griggio-step* :: ('v dleq-system \times 'v set) option rel **where**

griggio-eq-step: *griggio-equiv-step* $SF\ TG \implies (Some\ (SF,X),\ Some\ (TG,\ X)) \in$
griggio-step
| *griggio-fail-step*: *griggio-unsat* $p \implies (Some\ ((S,insert\ p\ F),X),\ None) \in$ *grig-*
gio-step
| *griggio-complex-step*: *coeff-l* $p\ x \neq 0$
 $\implies q = reorder-nontriv-var\ x\ p\ y$
 $\implies e = reorder-for-var\ x\ q$
 $\implies y \notin V \cup X$
 $\implies (Some\ ((S,insert\ p\ F),X),$
 $Some\ ((insert\ (x,e)\ (map-prod\ id\ (substitute-l\ x\ e)\ 'S),\ substitute-l\ x\ e\ ')$
 $insert\ p\ F),\ insert\ y\ X))$
 \in *griggio-step*

lemma *griggio-step*: **assumes** $(A,B) \in$ *griggio-step*

and *invariant-state* A

shows *invariant-state* B

<proof>

context

assumes $VE: \bigcup (vars-l\ 'E) \subseteq V$

begin

lemma *griggio-steps*: **assumes** $(Some\ ((\{\},E),\ \{\}),\ SFO) \in$ *griggio-step* $\hat{*}$ (**is** $(?I,-)$
 $\in -)$

shows *invariant-state* SFO

<proof>

lemma *griggio-fail*: **assumes** $(Some\ ((\{\},E),\ \{\}),\ None) \in$ *griggio-step* $\hat{*}$

shows $\nexists \alpha. \alpha \models_{dio} (E,\ \{\})$

<proof>

lemma *griggio-success*: **assumes** $(Some\ ((\{\},E),\ \{\}),\ Some\ ((S,\ \{\}),X)) \in$ *grig-*
gio-step $\hat{*}$

and $\beta: \beta = adjust-assign\ S-list\ \alpha\ set\ S-list = S$

shows $\beta \models_{dio} (E,\ \{\})$

<proof>

In the following lemma we not only show that the equations E are solvable, but also how the solution S can be used to process other constraints. Assume P describes an indexed set of polynomials, and f is a formula that describes how these polynomials must be evaluated, e.g., $f\ i = (i\ 1 \leq 0 \wedge i\ 2 > 5 * i\ 3)$ for some inequalities.

Then $f(P) \wedge E$ is equi-satisfiable to $f(\sigma(P))$ where σ is a substitution computed from S , and *adjust-assign* S is used to translated a solution in one direction.

theorem *griggio-success-translations*:

fixes $P :: 'i \Rightarrow (int,'v)lpoly$ **and** $f :: ('i \Rightarrow int) \Rightarrow bool$

assumes $(Some ((\{\}, E), \{\}), Some ((S, \{\}), X)) \in griggio-step \hat{*}$
and $\sigma: \sigma = solution-subst\ S-list$
and $S-list: set\ S-list = S$
shows

$$f (\lambda i. eval-l\ \alpha\ (substitute-all-l\ \sigma\ (P\ i))) \implies$$

$$\beta = adjust-assign\ S-list\ \alpha \implies$$

$$f (\lambda i. eval-l\ \beta\ (P\ i)) \wedge \beta \models_{dio}\ (E, \{\})$$

$$f (\lambda i. eval-l\ \alpha\ (P\ i)) \wedge \alpha \models_{dio}\ (E, \{\}) \implies$$

$$(\bigwedge i. vars-l\ (P\ i) \subseteq V) \implies$$

$$\exists \gamma. f (\lambda i. eval-l\ \gamma\ (substitute-all-l\ \sigma\ (P\ i)))$$

<proof>

corollary *griggio-success-equivalence:*
fixes $P :: 'i \Rightarrow (int, 'v)lpoly$ **and** $f :: ('i \Rightarrow int) \Rightarrow bool$
assumes $(Some ((\{\}, E), \{\}), Some ((S, \{\}), X)) \in griggio-step \hat{*}$
and $\sigma: \sigma = solution-subst\ S-list$
and $S-list: set\ S-list = S$
and $vV: \bigwedge i. vars-l\ (P\ i) \subseteq V$
shows

$$(\exists \alpha. f (\lambda i. eval-l\ \alpha\ (substitute-all-l\ \sigma\ (P\ i))))$$

$$\longleftrightarrow (\exists \alpha. f (\lambda i. eval-l\ \alpha\ (P\ i)) \wedge Ball\ E\ (satisfies-dleq\ \alpha))$$

<proof>

end
end

end

4.2 Executable Algorithm

theory *Linear-Diophantine-Solver-Impl*
imports
Linear-Diophantine-Solver
begin

definition *simplify-dleq* $:: 'v\ dleq \Rightarrow 'v\ dleq + bool$ **where**
simplify-dleq $p = (let$
 $g = gcd-coeffs-l\ p;$
 $c = constant-l\ p$
in *if* $g = 0$ **then**
 $Inr\ (c = 0)$
else if $g = 1$ **then** $Inl\ p$
else if $g\ dvd\ c$ **then** $Inl\ (sdiv-l\ p\ g)$ **else** $Inr\ False)$

lemma *simplify-dleq-0:* **assumes** *simplify-dleq* $p = Inr\ True$
shows $p = 0$

<proof>

lemma *simplify-dleq-fail*: **assumes** *simplify-dleq p = Inr False*
shows *griggio-unsat p*
<proof>

definition *dleq-normalized* **where** *dleq-normalized p = (Gcd (coeff-l p ' vars-l p) = 1)*

definition *size-dleq* :: *'v dleq ⇒ int* **where**
size-dleq p = (if vars-l p = {} then 0 else Min ((abs o coeff-l p) ' (vars-l p)))

lemma *size-dleq-pos*: *size-dleq p ≥ 0* *<proof>*

lemma *size-dleq-ge*:
assumes *vars-l p ≠ {} ∧ v. v ∈ vars-l p ⇒ abs (coeff-l p v) ≥ c*
shows *size-dleq p ≥ c*
<proof>

lemma *size-dleq-le*:
assumes *v ∈ vars-l p*
shows *abs (coeff-l p v) ≥ size-dleq p*
<proof>

lemma *Min-image-mono*:
assumes *X ≠ {} finite X*
assumes $\bigwedge x. x \in X \implies f x \leq g x$
shows $\text{Min } (f ' X) \leq \text{Min } (g ' X)$
<proof>

lemma *size-dleq-mono*:
assumes *vars-l p = vars-l q*
assumes $\bigwedge x. x \in \text{vars-l } p \implies |\text{coeff-l } p x| \leq |\text{coeff-l } q x|$
shows *size-dleq p ≤ size-dleq q*
<proof>

lemma *simplify-dleq-keep*: **assumes** *simplify-dleq p = Inl q*
shows
 $\exists g \geq 1. \text{normalize-dleq } p = (g, q)$
 size-dleq p ≥ size-dleq q
 dleq-normalized q
<proof>

fun *simplify-dleqs* :: *'v dleq list ⇒ 'v dleq list option* **where**
 simplify-dleqs [] = Some []
| *simplify-dleqs (e # es) = (case simplify-dleq e of*
 Inr False ⇒ None
 | *Inr True ⇒ simplify-dleqs es*

| Inl e' \Rightarrow map-option (Cons e') (simplify-dleqs es))

context griggio-input
begin

lemma simplify-dleqs: simplify-dleqs es = None \Longrightarrow (Some ((S,set es \cup F),X), None) \in griggio-step^{*}

simplify-dleqs es = Some fs \Longrightarrow
(Some ((S,set es \cup F),X), Some ((S,set fs \cup F),X)) \in griggio-step^{*}
 \wedge Ball (set fs) dleq-normalized \wedge length fs \leq length es \wedge
(length fs < length es \vee fs = [] \vee size-dleq (hd fs) \leq size-dleq (hd es))
<proof>

context
fixes fresh-var :: nat \Rightarrow 'v
begin

partial-function (option) dleq-solver-main
:: nat \Rightarrow ('v \times 'v dleq) list \Rightarrow 'v dleq list \Rightarrow ('v \times (int,'v)lpoly) list option **where**
dleq-solver-main n s es = (case simplify-dleqs es of

None \Rightarrow None
| Some [] \Rightarrow Some s
| Some (p # fs) \Rightarrow
let x = min-var p; c = abs (coeff-l p x)
in if c = 1 then
let e = reorder-for-var x p;
sigma = substitute-l x e in
dleq-solver-main n ((x, e) # map (map-prod id sigma) s) (map sigma fs) else
let y = fresh-var n;
q = reorder-nontriv-var x p y;
e = reorder-for-var x q;
sigma = substitute-l x e in
dleq-solver-main (Suc n) ((x, e) # map (map-prod id sigma) s) (sigma p # map
sigma fs))

fun state-of **where** state-of n s es = Some ((set s, set es), fresh-var ' {.. n })

lemma dleq-solver-main: **assumes** fresh-var: range fresh-var \cap V = {} inj fresh-var
and inv: invariant-state (state-of n s es)
shows dleq-solver-main n s es = None \Longrightarrow (state-of n s es, None) \in griggio-step^{*}

dleq-solver-main n s es = Some s' \Longrightarrow \exists X. (state-of n s es, Some ((set s', {}), X)) \in griggio-step^{*}
<proof>

end

end

declare *griggio-input.dleq-solver-main.simps*[code]

definition *fresh-var-gen* :: ('v list \Rightarrow nat \Rightarrow 'v) \Rightarrow bool **where**
fresh-var-gen *fv* = (\forall *vs*. range (*fv vs*) \cap set *vs* = {} \wedge inj (*fv vs*))

context

fixes *fresh-var* :: 'v :: linorder list \Rightarrow nat \Rightarrow 'v
begin

definition *dleq-solver* :: 'v list \Rightarrow 'v dleq list \Rightarrow ('v \times (int,'v)lpoly) list option
where

dleq-solver *v e* = (let *fv* = *fresh-var* (*v @ concat* (*map vars-l-list e*))
in *griggio-input.dleq-solver-main* *fv* 0 [] *e*)

lemma *dleq-solver: assumes fresh-var-gen fresh-var*
and *dleq-solver v e = res*

shows

res = None \Longrightarrow \nexists α . $\alpha \models_{dio}$ (set *e*, {})
res = Some *s* \Longrightarrow *adjust-assign* *s* $\alpha \models_{dio}$ (set *e*, {})
res = Some *s* \Longrightarrow σ = *solution-subst* *s* \Longrightarrow
f (λ *i*. *eval-l* α (*substitute-all-l* σ (*P i*))) \Longrightarrow
 β = *adjust-assign* *s* $\alpha \Longrightarrow$
f (λ *i*. *eval-l* β (*P i*)) \wedge $\beta \models_{dio}$ (set *e*, {})
res = Some *s* \Longrightarrow σ = *solution-subst* *s* \Longrightarrow (\bigwedge *i*. vars-l (*P i*) \subseteq set *v*) \Longrightarrow
f (λ *i*. *eval-l* α (*P i*)) \wedge $\alpha \models_{dio}$ (set *e*, {})
 \exists γ . *f* (λ *i*. *eval-l* γ (*substitute-all-l* σ (*P i*)))
<proof>

definition *equality-elim-for-inequalities* :: 'v dleq list \Rightarrow 'v dlineq list \Rightarrow
('v dlineq list \times ((int,'v)assign \Rightarrow (int,'v)assign)) option **where**
equality-elim-for-inequalities *eqs ineqs* = (let *v* = *concat* (*map vars-l-list ineqs*)
in case *dleq-solver v eqs of*
None \Rightarrow None
| Some *s* \Rightarrow let σ = *substitute-all-l* (*solution-subst* *s*);
adj = *adjust-assign* *s*
in Some (*map* σ *ineqs*, *adj*))

lemma *equality-elim-for-inequalities: assumes fresh-var-gen fresh-var*
and *equality-elim-for-inequalities eqs ineqs = res*

shows *res* = None \Longrightarrow \nexists α . $\alpha \models_{dio}$ (set *eqs*, {})
res = Some (*ineqs'*, *adj*) \Longrightarrow $\alpha \models_{dio}$ (set *ineqs'*, {*adj*}) \Longrightarrow (*adj* α) \models_{dio} (set *eqs*,
set *ineqs*)
res = Some (*ineqs'*, *adj*) \Longrightarrow \nexists α . $\alpha \models_{dio}$ (set *ineqs'*, {*adj*}) \Longrightarrow \nexists α . $\alpha \models_{dio}$ (set
eqs, set *ineqs*)
res = Some (*ineqs'*, *adj*) \Longrightarrow length *ineqs'* = length *ineqs*

<proof>

end

definition *fresh-vars-nat* :: *nat list* \Rightarrow *nat* \Rightarrow *nat* **where**
fresh-vars-nat *xs* = (*let* *m* = *Suc* (*Max* (*set* (*0* # *xs*))) *in* (λ *n*. *m* + *n*))

lemma *fresh-vars-nat*: *fresh-var-gen* *fresh-vars-nat*
<proof>

lemmas *equality-elim-for-inequalities-nat* = *equality-elim-for-inequalities*[*OF fresh-vars-nat*]

end

5 Detection of Implicit Equalities

5.1 Main Abstract Reasoning Step

The abstract reasoning steps is due to Bromberger and Weidenbach. Make all inequalities strict and detect a minimal unsat core; all inequalities in this core are implied equalities.

theory *Equality-Detection-Theory*

imports

Farkas.Farkas

Jordan-Normal-Form.Matrix

begin

lemma *lec-rel-sum-list*: *lec-rel* (*sum-list* *cs*) =
(*if* (\exists *c* \in *set* *cs*. *lec-rel* *c* = *Lt-Rel*) *then* *Lt-Rel* *else* *Leq-Rel*)
<proof>

lemma *equality-detection-rat*: **fixes** *cs* :: *rat le-constraint set*

and *p* :: '*i* \Rightarrow *linear-poly*

and *co* :: '*i* \Rightarrow *rat*

and *I* :: '*i* *set*

defines *n* \equiv λ *i*. *Le-Constraint* *Leq-Rel* (*p* *i*) (*co* *i*)

and *s* \equiv λ *i*. *Le-Constraint* *Lt-Rel* (*p* *i*) (*co* *i*)

assumes *fin*: *finite* *cs* *finite* *I*

and *C*: $C \subseteq cs \cup s \text{ ' } I$

and *unsat*: $\nexists v. \forall c \in C. v \models_{le} c$

and *min*: $\bigwedge D. D \subset C \implies \exists v. \forall c \in D. v \models_{le} c$

and *sol*: $\forall c \in cs \cup n \text{ ' } I. v \models_{le} c$

and *i*: $i \in I$ s $i \in C$

shows (*p* *i*) \Downarrow *v* = *co* *i*

<proof>

end

5.2 Algorithm to Detect all Implicit Equalities in Q

Use incremental simplex algorithm to recursively detect all implied equalities.

theory *Equality-Detection-Impl*

imports

Equality-Detection-Theory

Simplex.Simplex-Incremental

Deriving.Compare-Instances

begin

lemma *indexed-sat-mono*: $(S, v) \models_{ics} cs \implies T \subseteq S \implies (T, v) \models_{ics} cs$
<proof>

lemma *assert-all-simplex-plain-unsat*: **assumes** *invariant-simplex cs J s*
and *assert-all-simplex K s = Unsat I*
shows $\neg (set K \cup J, v) \models_{ics} set cs$
<proof>

lemma *check-simplex-plain-unsat*: **assumes** *invariant-simplex cs J s*
and *check-simplex s = (s', Some I)*
shows $\neg (J, v) \models_{ics} set cs$
<proof>

hide-const (**open**) *Congruence.eq*

fun *le-of-constraint* :: *constraint* \Rightarrow *rat le-constraint* **where**
 le-of-constraint (LEQ p c) = *Le-Constraint Leq-Rel p c*
 | *le-of-constraint* (LT p c) = *Le-Constraint Lt-Rel p c*
 | *le-of-constraint* (GEQ p c) = *Le-Constraint Leq-Rel (-p) (-c)*
 | *le-of-constraint* (GT p c) = *Le-Constraint Lt-Rel (-p) (-c)*

fun *poly-of-constraint* :: *constraint* \Rightarrow *linear-poly* **where**
 poly-of-constraint (LEQ p c) = p
 | *poly-of-constraint* (LT p c) = p
 | *poly-of-constraint* (GEQ p c) = (-p)
 | *poly-of-constraint* (GT p c) = (-p)

fun *const-of-constraint* :: *constraint* \Rightarrow *rat* **where**
 const-of-constraint (LEQ p c) = c
 | *const-of-constraint* (LT p c) = c
 | *const-of-constraint* (GEQ p c) = (-c)
 | *const-of-constraint* (GT p c) = (-c)

fun *is-no-equality* :: *constraint* \Rightarrow *bool* **where**
is-no-equality (EQ *p c*) = *False*
| *is-no-equality* - = *True*

fun *is-equality* :: *constraint* \Rightarrow *bool* **where**
is-equality (EQ *p c*) = *True*
| *is-equality* - = *False*

lemma *le-of-constraint*: *is-no-equality c* \Longrightarrow $v \models_c c \longleftrightarrow (v \models_{le} \text{le-of-constraint } c)$
⟨*proof*⟩

lemma *le-of-constraints*: *Ball cs is-no-equality* \Longrightarrow $v \models_{cs} cs \longleftrightarrow (\forall c \in cs. v \models_{le} \text{le-of-constraint } c)$
⟨*proof*⟩

fun *is-strict* :: *constraint* \Rightarrow *bool* **where**
is-strict (GT -) = *True*
| *is-strict* (LT -) = *True*
| *is-strict* - = *False*

fun *is-nstrict* :: *constraint* \Rightarrow *bool* **where**
is-nstrict (GEQ -) = *True*
| *is-nstrict* (LEQ -) = *True*
| *is-nstrict* - = *False*

lemma *is-equality-iff*: *is-equality c* = $(\neg \text{is-strict } c \wedge \neg \text{is-nstrict } c)$
⟨*proof*⟩

lemma *is-nstrict-iff*: *is-nstrict c* = $(\neg \text{is-strict } c \wedge \neg \text{is-equality } c)$
⟨*proof*⟩

fun *make-strict* :: *constraint* \Rightarrow *constraint* **where**
make-strict (GEQ *p c*) = GT *p c*
| *make-strict* (LEQ *p c*) = LT *p c*
| *make-strict* *c* = *c*

fun *make-equality* :: *constraint* \Rightarrow *constraint* **where**
make-equality (GEQ *p c*) = EQ *p c*
| *make-equality* (LEQ *p c*) = EQ *p c*
| *make-equality* *c* = *c*

fun *make-ineq* :: *constraint* \Rightarrow *constraint* **where**
make-ineq (GEQ *p c*) = GEQ *p c*
| *make-ineq* (LEQ *p c*) = LEQ *p c*
| *make-ineq* (EQ *p c*) = LEQ *p c*

fun *make-flipped-ineq* :: *constraint* \Rightarrow *constraint* **where**

```

  make-flipped-ineq (GEQ p c) = LEQ p c
| make-flipped-ineq (LEQ p c) = GEQ p c
| make-flipped-ineq (EQ p c)  = GEQ p c

```

lemma *poly-const-repr*: **assumes** *is-nstrict c*
shows *le-of-constraint c = Le-Constraint Leq-Rel (poly-of-constraint c) (const-of-constraint c)*
le-of-constraint (make-strict c) = Le-Constraint Lt-Rel (poly-of-constraint c) (const-of-constraint c)
le-of-constraint (make-flipped-ineq c) = Le-Constraint Leq-Rel (– poly-of-constraint c) (– const-of-constraint c)
 ⟨proof⟩

lemma *poly-const-repr-set*: **assumes** *Ball cs is-nstrict*
shows *le-of-constraint ‘ cs = (λ c. Le-Constraint Leq-Rel (poly-of-constraint c) (const-of-constraint c)) ‘ cs*
le-of-constraint ‘ (make-strict ‘ cs) = (λ c. Le-Constraint Lt-Rel (poly-of-constraint c) (const-of-constraint c)) ‘ cs
 ⟨proof⟩

datatype *eqd-index* =
Ineq nat |
FIneq nat |
SIneq nat |
TmpSIneq nat

fun *num-of-index* :: *eqd-index* ⇒ *nat* **where**
num-of-index (FIneq n) = n
| *num-of-index (Ineq n) = n*
| *num-of-index (SIneq n) = n*
| *num-of-index (TmpSIneq n) = n*

derive *compare-order eqd-index*

fun *index-constraint* :: *nat* × *constraint* ⇒ *eqd-index i-constraint list* **where**
index-constraint (n, c) = (
if is-nstrict c then [(Ineq n, c), (FIneq n, make-flipped-ineq c), (TmpSIneq n, make-strict c)] else
if is-strict c then [(SIneq n, c)] else
[(Ineq n, make-ineq c), (FIneq n, make-flipped-ineq c)]
)

definition *init-constraints* :: *constraint list* ⇒ *eqd-index i-constraint list* × *nat list* × *nat list* **where**
init-constraints cs = (let
ics' = zip [0 ..< length cs] cs;
ics = concat (map index-constraint ics');
ineqs = map fst (filter (is-nstrict o snd) ics');
)

```

sneqs = map fst (filter (is-strict o snd) ics^);
eqs = map fst (filter (is-equality o snd) ics^);
in (ics, ineqs, sneqs, eqs)

```

definition *index-of* :: *nat list* \Rightarrow *nat list* \Rightarrow *nat list* \Rightarrow *eqd-index list* **where**
index-of ineqs sineqs eqs = *map SIneq sineqs @ map Ineq eqs @ map FIneq eqs @ map Ineq ineqs*

context

```

fixes cs :: constraint list
and ics :: eqd-index i-constraint list

```

begin

definition *cs-of* :: *nat list* \Rightarrow *nat list* \Rightarrow *nat list* \Rightarrow *constraint set* **where**
cs-of ineqs sineqs eqs = *Simplex.restrict-to (set (index-of ineqs sineqs eqs)) (set ics)*

lemma *init-constraints*: **assumes** *init*: *init-constraints cs* = (*ics, ineqs, sineqs, eqs*)

```

shows v  $\models_{cs}$  cs-of ineqs sineqs eqs  $\longleftrightarrow$  v  $\models_{cs}$  set cs
distinct-indices ics

```

```

fst ' set ics = set (map SIneq sineqs @ map Ineq eqs @ map FIneq eqs @ map Ineq ineqs @ map FIneq ineqs @ map TmpSIneq ineqs) (is - = ?l)

```

```

set eqs = {i. i < length cs  $\wedge$  is-equality (cs ! i)}
set ineqs = {i. i < length cs  $\wedge$  is-nstrict (cs ! i)}
set sineqs = {i. i < length cs  $\wedge$  is-strict (cs ! i)}

```

```

set ics =
  ( $\lambda$ i. (Ineq i, make-ineq (cs ! i))) ' set eqs  $\cup$ 
  ( $\lambda$ i. (FIneq i, make-flipped-ineq (cs ! i))) ' set eqs  $\cup$ 
  (( $\lambda$ i. (Ineq i, cs ! i)) ' set ineqs  $\cup$ 
  ( $\lambda$ i. (FIneq i, make-flipped-ineq (cs ! i))) ' set ineqs  $\cup$ 
  ( $\lambda$ i. (TmpSIneq i, make-strict (cs ! i))) ' set ineqs)  $\cup$ 
  ( $\lambda$ i. (SIneq i, cs ! i)) ' set sineqs (is - = ?Large)

```

```

distinct (eqs @ ineqs @ sineqs)
set (eqs @ ineqs @ sineqs) = {0 ..< length cs}

```

<proof>

definition *init-eq-finder-rat* :: (*eqd-index simplex-state* \times *nat list* \times *nat list* \times *nat list*) *option* **where**

```

init-eq-finder-rat = (case init-constraints cs of (ics, ineqs, sineqs, eqs)
 $\Rightarrow$  let s0 = init-simplex ics
  in (case assert-all-simplex (index-of ineqs sineqs eqs) s0
    of Unsat -  $\Rightarrow$  None
    | Inr s1  $\Rightarrow$  (case check-simplex s1
      of (-, Some -)  $\Rightarrow$  None
      | (s2, None)  $\Rightarrow$  Some (s2, ineqs, sineqs, eqs))))

```

partial-function (*tailrec*) *eq-finder-main-rat* :: *eqd-index simplex-state* \Rightarrow *nat list*

\Rightarrow *nat list* \Rightarrow *nat list* \times *nat list* \times (*var* \Rightarrow *rat*) **where**
`[code]: eq-finder-main-rat s ineq eq = (if ineq = [] then (ineq, eq, solution-simplex s) else let`
`cp = checkpoint-simplex s;`
`res-strict = (case assert-all-simplex (map TmpSIneq ineq) s — Make all`
inequalities strict and test sat
`of Unsat C \Rightarrow Inl (s, C)`
`| Inr s1 \Rightarrow (case check-simplex s1 of`
`(s2, None) \Rightarrow Inr (solution-simplex s2)`
`(s2, Some C) \Rightarrow Inl (backtrack-simplex cp s2, C)))`
`in case res-strict of`
`Inr sol \Rightarrow (ineq, eq, sol) — if indeed all equalities are strictly sat, then no`
further equality is implied
`| Inl (s2, C) \Rightarrow let`
`eq' = remdups [i. TmpSIneq i <- C]; — collect all indices of the strict`
inequalities within the minimal unsat-core
`— the remdups might not be necessary, however the simplex interfact`
does not ensure distinctness of C
`s3 = sum.projr (assert-all-simplex (map FIneq eq') s2); — and permantly`
add the flipped inequalities
`s4 = fst (check-simplex s3); — this check will succeed, no unsat can be`
reported here
`ineq' = filter (λ i. i \notin set eq') ineq — add eq' from inequalities to equalities`
and continue
`in eq-finder-main-rat s4 ineq' (eq' @ eq))`

definition *eq-finder-rat* :: (*nat list* \times (*var* \Rightarrow *rat*)) *option* **where**
`eq-finder-rat = (case init-eq-finder-rat of None \Rightarrow None`
`| Some (s, ineqs, sineqs, eqs) \Rightarrow Some (`
`case eq-finder-main-rat s ineqs eqs of (ineq, eq, sol)`
`\Rightarrow (eq, sol)))`

context

fixes *eqs ineqs sineqs*:: *nat list*

assumes *init-cs*: *init-constraints cs* = (*ics, ineqs, sineqs, eqs*)

begin

definition *equiv-to-cs* **where**

`equiv-to-cs eq = (\forall v. v \models_{cs} set cs = (set (index-of ineqs sineqs eq), v) \models_{ics} set ics)`

definition *strict-ineq-sat* *ineq eq v* = ((set (index-of ineqs sineqs eq) \cup TmpSIneq ' set ineq, v) \models_{ics} set ics)

lemma *init-eq-finder-rat*: *init-eq-finder-rat* = None \implies \nexists v. v \models_{cs} set cs

init-eq-finder-rat = Some (s, ineq, sineq, eq) \implies

checked-simplex ics (set (index-of ineqs sineqs eq)) s

\wedge eq = eqs \wedge ineq = ineqs \wedge sineq = sineqs

\wedge *equiv-to-cs* eq

\wedge *distinct* (*ineq* @ *sineq* @ *eq*)
 \wedge *set* (*ineq* @ *sineq* @ *eq*) = {0 ..< length cs}
 <proof>

lemma *eq-finder-main-rat*: **fixes** *Ineq Eq*
assumes *checked-simplex ics* (*set* (*index-of ineqs sineqs eq*)) *s*
and *set ineq* \subseteq *set ineqs*
and *set eqs* \subseteq *set eq* \wedge *set eq* \cup *set ineq* = *set eqs* \cup *set ineqs*
and *eq-finder-main-rat s ineq eq* = (*Ineq*, *Eq*, *v-sol*)
and *equiv-to-cs eq*
and *distinct* (*ineq* @ *eq*)
shows *set Ineq* \subseteq *set ineqs* *set eqs* \subseteq *set Eq* *set Ineq* \cup *set Eq* = *set eqs* \cup *set ineqs*

and *equiv-to-cs Eq*
and *strict-ineq-sat Ineq Eq v-sol*
and *distinct* (*Ineq* @ *Eq*)
 <proof>

lemma *eq-finder-rat-in-ctxt*: *eq-finder-rat* = *None* $\implies \nexists v. v \models_{cs} \text{set } cs$
eq-finder-rat = *Some* (*eq-idx*, *v-sol*) $\implies \{i . i < \text{length } cs \wedge \text{is-equality } (cs ! i)\}$
 \subseteq *set eq-idx* \wedge
set eq-idx \subseteq {0 ..< length cs} \wedge
distinct eq-idx (**is** - \implies ?main1)
eq-finder-rat = *Some* (*eq-idx*, *v-sol*) \implies
set feq = *make-equality* '(!) *cs* ' *set eq-idx* \implies
set fineq = (!) *cs* ' ({0 ..< length cs} - *set eq-idx*) \implies
 $(\forall v. v \models_{cs} \text{set } cs \longleftrightarrow v \models_{cs} (\text{set } feq \cup \text{set } fineq)) \wedge$
Ball (*set feq*) *is-equality* \wedge *Ball* (*set fineq*) *is-no-equality* \wedge
 $(v\text{-sol} \models_{cs} (\text{set } feq \cup \text{make-strict 'set } fineq))$ (**is** - \implies - \implies - \implies ?main2)
 <proof>

end
end

lemma *eq-finder-rat*:
eq-finder-rat cs = *None* $\implies \nexists v. v \models_{cs} \text{set } cs$ (**is** ?p1 \implies ?g1)
eq-finder-rat cs = *Some* (*eq-idx*, *v-sol*) \implies
 $\{i . i < \text{length } cs \wedge \text{is-equality } (cs ! i)\} \subseteq$ *set eq-idx* \wedge
set eq-idx \subseteq {0 ..< length cs} \wedge
distinct eq-idx (**is** ?p2 \implies ?g2)
eq-finder-rat cs = *Some* (*eq-idx*, *v-sol*) \implies
set eq = *make-equality* '(!) *cs* ' *set eq-idx* \implies
set ineq = (!) *cs* ' ({0 ..< length cs} - *set eq-idx*) \implies
 $(\forall v. v \models_{cs} \text{set } cs \longleftrightarrow v \models_{cs} (\text{set } eq \cup \text{set } ineq)) \wedge$
Ball (*set eq*) *is-equality* \wedge *Ball* (*set ineq*) *is-no-equality* \wedge
 $(v\text{-sol} \models_{cs} (\text{set } eq \cup \text{make-strict 'set } ineq))$
is ?p2 \implies ?p3 \implies ?p4 \implies ?g3

<proof>

hide-fact *eq-finder-rat-in-ctxt*

end

5.3 Algorithm to Detect Implicit Equalities in \mathbb{Z}

Use the rational equality finder to identify integer equalities.

Basically, this is just a conversion between the different types of constraints.

theory *Linear-Diophantine-Eq-Finder*

imports

Linear-Polynomial-Impl

Equality-Detection-Impl

Diophantine-Tightening

begin

definition *linear-poly-of-lpoly* :: $(\text{int}, \text{var})\text{lpoly} \Rightarrow \text{linear-poly}$ **where**

linear-poly-of-lpoly $p = (\text{let } cxs = \text{map } (\lambda v. (v, \text{coeff-l } p v)) (\text{vars-l-list } p)$
in sum-list (map (lambda (x,c). lp-monom (of-int c) x) cxs))

lemma *linear-poly-of-lpoly-impl*[code]:

linear-poly-of-lpoly (lpoly-of p) = (let cxs = vars-coeffs-impl p
in sum-list (map (lambda (x,c). lp-monom (of-int c) x) cxs))
<proof>

lemma *valuate-sum-list*: $\text{valuate } (\text{sum-list } ps) \alpha = \text{sum-list } (\text{map } (\lambda p. \text{valuate } p \alpha) ps)$

<proof>

lemma *linear-poly-of-lpoly*: $\text{rat-of-int } (\text{eval-l } \alpha p) = \text{of-int } (\text{constant-l } p) + \text{valuate } (\text{linear-poly-of-lpoly } p) (\lambda x. \text{of-int } (\alpha x))$

<proof>

definition *dleq-to-constraint* :: $\text{var dleq} \Rightarrow \text{constraint}$ **where**

dleq-to-constraint p = EQ (linear-poly-of-lpoly p) (of-int (- constant-l p))

lemma *dleq-to-constraint*: $\text{satisfies-dleq } \alpha e \iff \text{satisfies-constraint } (\lambda x. \text{rat-of-int } (\alpha x)) (\text{dleq-to-constraint } e)$

<proof>

definition *dlineq-to-constraint* :: $\text{var dlineq} \Rightarrow \text{constraint}$ **where**

dlineq-to-constraint p = LEQ (linear-poly-of-lpoly p) (of-int (- constant-l p))

lemma *dlineq-to-constraint*: $\text{satisfies-dlineq } \alpha e \iff$

$\text{satisfies-constraint } (\lambda x. \text{rat-of-int } (\alpha x)) (\text{dlineq-to-constraint } e)$

<proof>

definition *eq-finder-int* :: var dlineq list \Rightarrow
 (var dleq list \times var dlineq list) option **where**
eq-finder-int ineqs = (case
 eq-finder-rat (map *dlineq-to-constraint ineqs*) of
 None \Rightarrow None
 | Some (idx-eq, -) \Rightarrow let I = set idx-eq;
 ics = zip [0..*length ineqs*] ineqs
 in case List.partition ($\lambda (i,c). i \in I$) ics
 of (eqs2, ineqs2) \Rightarrow Some (map snd eqs2, map snd ineqs2))

lemma *classify-dlineq-to-constraint[simp]*:
 \neg is-strict (*dlineq-to-constraint c*)
 \neg is-equality (*dlineq-to-constraint c*)
 is-nstrict (*dlineq-to-constraint c*)
 <proof>

lemma *init-constraints-ineqs*:
init-constraints (map *dlineq-to-constraint ineqs*) =
 (let idx = [0..*length ineqs*];
 ics' = zip idx
 (map *dlineq-to-constraint ineqs*);
 ics = concat (map *index-constraint ics'*)
 in (ics, idx, [], []))
 <proof>

lemmas *eq-finder-int-code[code]* =
eq-finder-int-def[unfolded eq-finder-rat-def init-eq-finder-rat-def, unfolded init-constraints-ineqs]

lemma *eq-finder-int: assumes*
res: *eq-finder-int ineqs* = *res*
shows *res* = None $\Rightarrow \nexists \alpha. \alpha \models_{dio} (\{\}, set\ ineqs)$
res = Some (eqs, ineqs') $\Rightarrow \alpha \models_{dio} (\{\}, set\ ineqs) \iff \alpha \models_{dio} (set\ eqs, set\ ineqs')$
res = Some (eqs, ineqs') $\Rightarrow \exists \alpha. \alpha \models_{cs} (make-strict\ 'dlineq-to-constraint'\ set\ ineqs')$
res = Some (eqs, ineqs') $\Rightarrow length\ ineqs = length\ eqs + length\ ineqs'$
 <proof>

end

6 A Combined Preprocessor

We combine equality detection, equality elimination and tightening in one function that eliminates all explicit and implicit equations from a list of inequalities and equalities, to either detect unsat or to return an equivalent list of inequalities which all can be satisfied strictly in the rational numbers.

theory *Dio-Preprocessor*
imports

Linear-Polynomial-Impl
Linear-Diophantine-Solver-Impl
Diophantine-Tightening
Linear-Diophantine-Eq-Finder

begin

Combine equality elimination and tightening in one algorithm

definition *dio-elim-equations-and-tighten* :: *var dleq list* \Rightarrow *var dlineq list* \Rightarrow
(var dlineq list \times *((int,var)assign* \Rightarrow *(int,var)assign*)) *option* **where**
dio-elim-equations-and-tighten eqs ineqs = *(case equality-elim-for-inequalities fresh-vars-nat*
eqs ineqs
of None \Rightarrow *None*
| Some (ineqs2, adj) \Rightarrow *map-option* (λ *ineqs3. (ineqs3, adj)*) (*tighten-ineqs*
ineqs2))

lemma *dio-elim-equations-and-tighten: assumes*

res: dio-elim-equations-and-tighten eqs ineqs = res

shows *res = None* \Longrightarrow $\nexists \alpha. \alpha \models_{dio} (\text{set } eqs, \text{set } ineqs)$

res = Some (ineqs', adj) $\Longrightarrow \alpha \models_{dio} (\{\}, \text{set } ineqs')$ $\Longrightarrow \beta = \text{adj } \alpha \Longrightarrow \beta \models_{dio}$
(set eqs, set ineqs)

res = Some (ineqs', adj) $\Longrightarrow \nexists \alpha. \alpha \models_{dio} (\{\}, \text{set } ineqs')$ $\Longrightarrow \nexists \alpha. \alpha \models_{dio} (\text{set}$
eqs, set ineqs)

res = Some (ineqs', adj) $\Longrightarrow \text{length } ineqs' \leq \text{length } ineqs$

<proof>

Now all three preprocessing steps are combined.

Since after an equality elimination the resulting inequalities might be tightened, it can happen that after the tightening new equalities are implied; therefore the whole process is performed recursively

function *dio-preprocess-main* :: *(int, var) lpoly list* \Rightarrow *((int, var) lpoly list* \times
((int,var)assign \Rightarrow *(int,var)assign*)) *option* **where**
dio-preprocess-main ineqs = *(case eq-finder-int ineqs of None* \Rightarrow *None*
| Some (eqs, ineqs') \Rightarrow *(case eqs of []* \Rightarrow *Some (ineqs', id)*
| - \Rightarrow *(case dio-elim-equations-and-tighten eqs ineqs' of None* \Rightarrow *None*
| Some (ineqs'', adj) \Rightarrow *map-option (map-prod id* (λ *adj'. adj o adj')*))
(dio-preprocess-main ineqs''))))
<proof>

termination

<proof>

declare *dio-preprocess-main.simps*[*simp del*]

lemma *dio-preprocess-main: assumes*

res: dio-preprocess-main ineqs = res

shows *res = None* $\Longrightarrow \nexists \alpha. \alpha \models_{dio} (\{\}, \text{set } ineqs)$

res = Some (ineqs', adj) $\Longrightarrow \alpha \models_{dio} (\{\}, \text{set } ineqs')$ $\Longrightarrow (\text{adj } \alpha) \models_{dio} (\{\}, \text{set}$
ineqs)

```

    res = Some (ineqs', adj)  $\implies$   $\nexists$   $\alpha. \alpha \models_{dio} (\{\}, set\ ineqs')$   $\implies$   $\nexists$   $\alpha. \alpha \models_{dio} (\{\}, set\ ineqs)$ 
    res = Some (ineqs', adj)  $\implies$   $\exists$   $\alpha. \alpha \models_{cs} (make\ strict\ 'dlineq\ to\ constraint'\ 'set\ ineqs')$ 
    <proof>

```

The final preprocessing function just does some initial round of equality elimination and tightening before invoking the main algorithm which tries to detect and eliminate further implicit equalities.

```

definition dio-preprocess :: var dleq list  $\Rightarrow$  var dlineq list  $\Rightarrow$  (var dlineq list  $\times$ 
((int,var)assign  $\Rightarrow$  (int,var)assign)) option where
  dio-preprocess eqs ineqs = (case dio-elim-equations-and-tighten eqs ineqs of None
 $\Rightarrow$  None
    | Some (ineqs', adj)  $\Rightarrow$  map-option (map-prod id ( $\lambda$  adj'. adj o adj'))
    (dio-preprocess-main ineqs'))

```

The *dio-preprocess* algorithm eliminates all explicit and implicit equalities; in the negative outcome (None) we see (1) that the input constraints are unsat; and in the positive case (Some) (2) the resulting inequalities are equisatisfiable to the input constraints, (3) the solutions can be transformed in one direction via an adjuster adj, and (4) all resulting inequalities can be satisfied strictly using rational numbers, so no further equalities can be deduced using rational arithmetic reasoning.

```

lemma dio-preprocess: assumes res: dio-preprocess eqs ineqs = res
shows res = None  $\implies$   $\nexists$   $\alpha. \alpha \models_{dio} (set\ eqs, set\ ineqs)$ 
  res = Some (ineqs', adj)  $\implies$  ( $\exists$   $\alpha. \alpha \models_{dio} (\{\}, set\ ineqs')$ )  $\longleftrightarrow$  ( $\exists$   $\alpha. \alpha \models_{dio} (set\ eqs, set\ ineqs)$ )
  res = Some (ineqs', adj)  $\implies$   $\alpha \models_{dio} (\{\}, set\ ineqs')$   $\implies$  (adj  $\alpha$ )  $\models_{dio} (set\ eqs, set\ ineqs)$ 
  res = Some (ineqs', adj)  $\implies$   $\exists$   $\alpha. \alpha \models_{cs} (make\ strict\ 'dlineq\ to\ constraint'\ 'set\ ineqs')$ 
  <proof>

```

end

7 Examples

```

theory Dio-Preprocessing-Examples
imports
  Dio-Preprocessor
begin

```

Encoding of an example task of <https://adventofcode.com/2025/day/10>, part 2.

The aim is to find the minimum value x0 for some satisfying assignment. Here, the assignment needs to be over natural numbers, so we add constraints

– $x_i \leq 0$. There are also implicit upper limits for each variable that are extracted from the equations.

definition *aoc-2025-10-2* :: var dleq list \times var dlineq list **where**

```

aoc-2025-10-2 = (let xs = map var-l [0 ..< 11] in case xs of
  [x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10] =>
  ([x1 + x4 + x5 + x6 + x8 - const-l 35,
    x3 + x4 + x5 + x7 - const-l 44,
    x1 + x2 + x3 + x5 + x6 + x7 + x8 + x10 - const-l 107,
    x2 + x3 + x4 + x5 + x8 + x9 + x10 - const-l 74,
    x4 + x6 + x10 - const-l 41,
    x3 + x4 + x5 + x6 + x8 - const-l 44,
    x1 + x4 + x7 - const-l 31,
    x1 + x3 + x4 + x5 + x6 + x7 + x9 - const-l 81,
    x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 - x0
  ], map (\lambda xi. - xi) (tl xs) @
  [ x1 - const-l 31,
    x2 - const-l 74,
    x3 - const-l 44,
    x4 - const-l 31,
    x5 - const-l 35,
    x6 - const-l 35,
    x7 - const-l 31,
    x8 - const-l 35,
    x9 - const-l 74,
    x10 - const-l 74]))

```

Preprocessing reduces the number of variables from 11 down to 2

lemma *case aoc-2025-10-2 of (eqs, ineqs) => case dio-preprocess eqs ineqs of*

```

Some (ineqs1, sol) => let alpha = (undefined(8 := a))(11 := b)
  in (map (eval-l alpha) ineqs1, sol alpha 0)
= (— new inequalities ... <= 0
  [ - a - 2 * b - 17,
    19 + 5 * a + 6 * b,
    - a - 2 * b - 22,
    3 + a + b,
    1 + 2 * a + 3 * b,
    1 + a + 3 * b,
    - a,
    b - 3,
    - 3 * a - 4 * b - 45,
    2 + a + 2 * b,
    - 5 * a - 6 * b - 93,
    a + 2 * b,
    - a - b - 34,
    - 2 * a - 3 * b - 36,
    - a - 3 * b - 32,
    a - 35,
    - 71 - b,
    3 * a + 4 * b - 29],

```

— new expression to be minimized
 $107 - a - 2 * b$
 ⟨proof⟩

After preprocessing, a brute-force approach to determine the minimum value $x0 = 114$ is possible: from $-a \leq 0$ and $a - 35 \leq 0$ we know $0 \leq a \wedge a \leq 35$, from $-3 + b \leq 0$ and $-71 - b \leq 0$ we get $-71 \leq b \wedge b \leq 3$.

lemma $114 = \text{min-list } [107 - a - 2 * b . a \leftarrow [0..35], b \leftarrow [-71 .. 3],$
 $- a - 2 * b - 17 \leq 0,$
 $19 + 5 * a + 6 * b \leq 0,$
 $- a - 2 * b - 22 \leq 0,$
 $3 + a + b \leq 0,$
 $1 + 2 * a + 3 * b \leq 0,$
 $1 + a + 3 * b \leq 0,$
 $- a \leq 0,$
 $b - 3 \leq 0,$
 $- 3 * a - 4 * b - 45 \leq 0,$
 $2 + a + 2 * b \leq 0,$
 $- 5 * a - 6 * b - 93 \leq 0,$
 $a + 2 * b \leq 0,$
 $- a - b - 34 \leq 0,$
 $- 2 * a - 3 * b - 36 \leq 0,$
 $- a - 3 * b - 32 \leq 0,$
 $a - 35 \leq 0,$
 $- 71 - b \leq 0,$
 $3 * a + 4 * b - 29 \leq 0]$
 ⟨proof⟩

Inequalities where branch-and-bound algorithm is not terminating without setting global bounds

definition *example-3-x-min-y* :: var dlineq list **where**
example-3-x-min-y = (let $x = \text{var-l } 1; y = \text{var-l } 2$ in
 $[\text{const-l } 1 - \text{smult-l } 3 x + \text{smult-l } 3 y,$
 $\text{smult-l } 3 x - \text{smult-l } 3 y - \text{const-l } 2])$

Preprocessing can detect unsat

lemma *case dio-preprocess* [] *example-3-x-min-y* of None \Rightarrow True | Some - \Rightarrow False
 ⟨proof⟩

Griggio, example 1, unsat detection by preprocessing

definition *griggio-example-1-eqs* :: var dleq list **where**
griggio-example-1-eqs = (let $x1 = \text{var-l } 1; x2 = \text{var-l } 2; x3 = \text{var-l } 3$ in
 $[\text{smult-l } 3 x1 + \text{smult-l } 3 x2 + \text{smult-l } 14 x3 - \text{const-l } 4,$
 $\text{smult-l } 7 x1 + \text{smult-l } 12 x2 + \text{smult-l } 31 x3 - \text{const-l } 17])$

lemma *case dio-preprocess griggio-example-1-eqs* [] of None \Rightarrow True | Some - \Rightarrow False

<proof>

Griggio, example 2, unsat detection by preprocessing

definition *griggio-example-2-eqs* :: *var dleq list where*

griggio-example-2-eqs = (let *x1* = var-l 1; *x2* = var-l 2; *x3* = var-l 3; *x4* = var-l 4 in
[smult-l 2 *x1* - smult-l 5 *x3*,
x2 - smult-l 3 *x4*])

definition *griggio-example-2-ineqs* :: (*int,var*) *lpoly list where*

griggio-example-2-ineqs = (let *x1* = var-l 1; *x2* = var-l 2; *x3* = var-l 3 in
[- smult-l 2 *x1* - *x2* - *x3* + const-l 7,
smult-l 2 *x1* + *x2* + *x3* - const-l 8])

lemma *case dio-preprocess griggio-example-2-eqs griggio-example-2-ineqs*

of None \Rightarrow *True* | *Some* - \Rightarrow *False*

<proof>

Termination proof of binary logarithm program $n := 0; \text{while } (x > 1) \{x := x \text{ div } 2; n := n + 1\}$

definition *example-log-transition-formula* :: (*int,var*) *lpoly list*

where *example-log-transition-formula* = (let *x* = var-l 1; *x'* = var-l 2; *n* = var-l 3; *n'* = var-l 4
in [const-l 1 - *x*,
n' - *n*,
n - *n'*,
smult-l 2 *x'* - *x*,
x - smult-l 2 *x'* - const-l 1])

x is decreasing in each iteration

value (*code*) let *x* = var-l 1; *x'* = var-l 2 in dio-preprocess [] ((*x* - *x'*) # *example-log-transition-formula*)

x is bounded by -2

value (*code*) let *x* = var-l 1 in dio-preprocess [] ((*x* + const-l 2) # *example-log-transition-formula*)

end

References

- [1] M. Bromberger and C. Weidenbach. New techniques for linear arithmetic: cubes and equalities. *Formal Methods Syst. Des.*, 51(3):433–461, 2017.
- [2] A. Griggio. A practical approach to satisfiability modulo linear integer arithmetic. *J. Satisf. Boolean Model. Comput.*, 8(1/2):1–27, 2012.

- [3] F. Maric, M. Spasic, and R. Thiemann. An incremental simplex algorithm with unsatisfiable core generation. *Arch. Formal Proofs*, 2018, 2018.
- [4] W. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In J. L. Martin, editor, *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pages 4–13. ACM, 1991.