A Preprocessor for Linear Diophantine Equalities and Inequalities

René Thiemann

University of Innsbruck, Austria

June 19, 2024

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

1.1 An Abstract Type for Multivariate Linear Polynomials

theory *Linear-Polynomial*
imports
Main
begin

typedef (**overloaded**) $('a :: zero,'v)$ *lpoly* $= \{ c :: 'v\ option \Rightarrow 'a.\ finite\ \{v.\ c\ v \neq 0\}\}$
by (*intro exI*[*of - λ -. 0*], *auto*)

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** $\lambda\ c\ x.\ - c\ x$ **by** *auto*

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 *goal-cases*
case (*1 c1 c2*)
have $\{v.\ c1\ v - c2\ v \neq 0\} \subseteq \{v.\ c1\ v \neq 0\} \cup \{v.\ c2\ v \neq 0\}$ **by** *auto*
from *finite-subset*[*OF this*] *1* **show** *?case* **by** *auto*
qed

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 *goal-cases*
case (*1 c1 c2*)
have $\{v.\ c1\ v + c2\ v \neq 0\} \subseteq \{v.\ c1\ v \neq 0\} \cup \{v.\ c2\ v \neq 0\}$ **by** *auto*
from *finite-subset*[*OF this*] *1* **show** *?case* **by** *auto*
qed

lift-definition *zero-lpoly* :: $('a, 'b)$ *lpoly* **is** $\lambda\ c.\ 0$ **by** *auto*

instance by (*intro-classes*; *transfer*, *auto simp*: *ac-simps*)

end

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

lemma *finite-vars-l*[*simp*,*intro*]: *finite* (*vars-l p*)
proof (*transfer*, *goal-cases*)
  case (*1 p*)
  show *?case* **by** (*rule finite-subset*[*OF - finite-imageI*[*OF 1*, *of the*]], *force*)
qed

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

lemma *vars-l-var*[*simp*]: *vars-l* (*var-l x*) $= \{x\}$ **by** *transfer auto*

lemma *vars-l-plus*: *vars-l* $(p1 + p2) \subseteq$ *vars-l p1* $\cup$ *vars-l p2*
  by (*transfer*, *auto*)

lemma *vars-l-minus*: *vars-l* $(p1 - p2) \subseteq$ *vars-l p1* $\cup$ *vars-l p2*
  by (*transfer*, *auto*)

lemma *vars-l-uminus*[*simp*]: *vars-l* $(- p) =$ *vars-l p*
  by (*transfer*, *auto*)

lemma *vars-l-zero*[*simp*]: *vars-l* $0 = \{\}$
  by (*transfer*, *auto*)

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

lemma *eval-l-mono*: **assumes** *finite V vars-l p* $\subseteq V$
  shows *eval-l* $\alpha\ p =$ *constant-l p* $+$ *sum* $(\lambda\ x.$ *coeff-l p x* $* \alpha\ x)\ V$
proof $-$
  define W **where** $W = V -$ *vars-l p*
  have [*simp*]: $(\sum x{\in}W.$ *coeff-l p x* $* \alpha\ x) = 0$
    by (*rule sum.neutral*, *unfold W-def*, *transfer*, *auto*)
  have V: $V = W \cup$ *vars-l p* $W \cap$ *vars-l p* $= \{\}$ **using** *assms* **unfolding** *W-def*
by *auto*
  show *?thesis* **unfolding** *eval-l-def* **using** *assms* **unfolding** V
    by (*subst sum.union-disjoint*[*OF - - V(2)*], *auto*)
qed

lemma *eval-l-cong*: **assumes** $\bigwedge\ x.\ x \in$ *vars-l p* $\Longrightarrow \alpha\ x = \beta\ x$
  shows *eval-l* $\alpha\ p =$ *eval-l* $\beta\ p$
  unfolding *eval-l-mono*[*OF finite-vars-l subset-refl*]

by (*intro arg-cong*[*of - -* λ *x. - + x*] *sum.cong refl*, *insert assms*, *auto*)

lemma *eval-l-0*[*simp*]: *eval-l* α *0* = *0* **unfolding** *eval-l-def*
by (*transfer*, *auto*)

lemma *eval-l-plus*[*simp*]: *eval-l* α (*p1* + *p2*) = *eval-l* α *p1* + *eval-l* α *p2*
proof −
have *fin*: *finite* (*vars-l p1* ∪ *vars-l p2*) **by** *auto*
show *?thesis*
apply (*subst* (*1 2 3*) *eval-l-mono*[*OF fin*])
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule vars-l-plus*)
subgoal by (*transfer*, *auto simp*: *sum.distrib algebra-simps*)
done
qed

lemma *eval-l-minus*[*simp*]: *eval-l* α (*p1* − *p2*) = *eval-l* α *p1* − *eval-l* α *p2*
proof −
have *fin*: *finite* (*vars-l p1* ∪ *vars-l p2*) **by** *auto*
show *?thesis*
apply (*subst* (*1 2 3*) *eval-l-mono*[*OF fin*])
subgoal by *auto*
subgoal by *auto*
subgoal by (*rule vars-l-minus*)
subgoal by (*transfer*, *auto simp*: *sum-subtractf algebra-simps*)
done
qed

lemma *eval-l-uminus*[*simp*]: *eval-l* α (− *p*) = − *eval-l* α *p*
unfolding *eval-l-def*
by (*transfer*, *auto simp*: *sum-negf*)

lemma *eval-l-var*[*simp*]: *eval-l* α (*var-l x*) = α *x*
apply (*subst eval-l-mono*[*of* {*x*}])
apply *force*
apply *force*
by (*transfer*, *auto*)

lift-definition *substitute-l* :: *'v* ⇒ (*'a* :: *comm-ring*, *'v*) *lpoly* ⇒ (*'a*,*'v*) *lpoly* ⇒ (*'a*,*'v*) *lpoly* **is**
λ *x p q y*. (*q*(*Some x* := *0*)) *y* + *q* (*Some x*) ∗ *p y*
proof *goal-cases*
case (*1 x p1 p2*)
show *?case*
apply (*rule finite-subset*[*of -* {*v. p1 v* ≠ *0*} ∪ {*v. p2 v* ≠ *0*}])
using *1* **by** *auto*
qed

lemma *vars-substitute-l*: *vars-l* (*substitute-l x p q*) ⊆ *vars-l p* ∪ (*vars-l q* − {*x*})
  by (*transfer*, *auto*)

lemma *substitute-l-id*: *x* ∉ *vars-l q* ⟹ *substitute-l x p q* = *q*
  by *transfer auto*

lemma *eval-substitute-l*: *eval-l* α (*substitute-l x p q*) = *eval-l* (α(*x* := *eval-l* α *p*)) *q*
proof −
  have *fin*: *finite* (*insert x* (*vars-l p* ∪ *vars-l q*))
    and *fin2*: *finite* (*vars-l p* ∪ *vars-l q*) **by** *auto*
  define *V* **where** *V* = *vars-l p* ∪ *vars-l q* − {*x*}
  have *V*: *finite V x* ∉ *V* **unfolding** *V-def* **by** *auto*
  show *?thesis*
    apply (*subst* (*1 2 3*) *eval-l-mono*[*OF fin*])
    subgoal by *auto*
    subgoal by *auto*
    subgoal using *vars-substitute-l*[*of x p q*] **by** *auto*
    apply (*unfold sum.insert-remove*[*OF fin2*])
    apply (*unfold V-def*[*symmetric*])
    using *V*
    apply (*transfer*)
    apply (*simp add*: *algebra-simps sum.distrib sum-distrib-left*)
    apply (*intro sum.cong*)
     apply (*auto simp*: *ac-simps*)
    done
qed

lift-definition *fun-of-lpoly* :: (′*a* :: *zero*,′*v*) *lpoly* ⇒ ′*v option* ⇒ ′*a* **is** λ *x*. *x* **.**

lift-definition *smult-l* :: ′*a* :: *comm-ring* ⇒ (′*a*,′*v*)*lpoly* ⇒ (′*a*,′*v*)*lpoly* **is**
  λ *y c z*. *y* ∗ *c z*
proof (*goal-cases*)
  case *1*
  show *?case* **by** (*rule finite-subset*[*OF - 1*], *auto*)
qed

lemma *coeff-smult-l*[*simp*]: *coeff-l* (*smult-l c p*) *x* = *c* ∗ *coeff-l p x*
  by *transfer auto*

lemma *constant-smult-l*[*simp*]: *constant-l* (*smult-l c p*) = *c* ∗ *constant-l p*
  by *transfer auto*

lemma *eval-smult-l*[*simp*]: *eval-l* α (*smult-l c p*) = *c* ∗ *eval-l* α *p*
  apply (*subst* (*1 2*) *eval-l-mono*[*of vars-l p*])
  subgoal by *simp*
  subgoal by *simp*

subgoal by *transfer auto*
unfolding *eval-l-def coeff-smult-l*
by (*auto simp*: *algebra-simps sum-distrib-left*)

lift-definition *const-l* :: $'a$ :: *zero* ⇒ ($'a$,$'v$) *lpoly* **is** λ *c*. (λ *z*. *0*)(*None* := *c*)
by *auto*

lemma *eval-l-const-l-constant*: *eval-l* α (*const-l* (*constant-l p*)) = *constant-l p*
unfolding *eval-l-def*
by *transfer auto*

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

lemma *eval-substitute-all-l*: *eval-l* α (*substitute-all-l* σ *p*) = *eval-l* (λ *x*. *eval-l* α (σ *x*)) *p*
proof −
define *xs* **where** *xs* = *vars-l p*
have *fin*: *finite xs* **unfolding** *xs-def* **by** *auto*
show *?thesis*
unfolding *substitute-all-l-def*
unfolding *eval-l-mono*[*OF finite-vars-l subset-refl*, *of - p*]
unfolding *eval-l-plus eval-l-const-l-constant*
unfolding *xs-def*[*symmetric*] **using** *fin*
proof (*intro arg-cong*[*of - - λ x. - + x*], *induct xs rule*: *finite-induct*)
case ∗: (*insert x xs*)
note *IH* = ∗(*3*)[*OF* ∗(*1*)]
note *sum* = *sum.insert*[*OF* ∗(*1*−*2*)]
show *?case* **unfolding** *sum eval-l-plus IH eval-smult-l* **by** *simp*
qed *simp*
qed

lift-definition *sdiv-l* :: (*int*, $'v$) *lpoly* ⇒ *int* ⇒ (*int*, $'v$) *lpoly* **is** λ *c q x*. *c x div q*
proof (*goal-cases*)
case *1*
show *?case* **by** (*rule finite-subset*[*OF - 1*], *auto*)
qed

definition *vars-l-list p* = *sorted-list-of-set* (*vars-l p*)

lemma *vars-l-list*[*simp*]: *set* (*vars-l-list p*) = *vars-l p*
unfolding *vars-l-list-def* **by** *simp*

definition *min-var* :: ($'a$ :: {*linorder*, *ordered-ab-group-add-abs*}, $'v$ :: *linorder*) *lpoly* ⇒ $'v$ **where**
min-var p = (*let*
xcs = *map* (λ *x*. (*x*,*coeff-l p x*)) (*vars-l-list p*);

axcs = *map* (*map-prod id abs*) *xcs*;
m = *min-list* (*map snd axcs*)
in (*case filter* (λ *xa*. *snd xa* = *m*) *axcs of*
(*x*,*a*) # - ⇒ *x*))

lemma *min-var*: *vars-l p* ≠ {} ⟹ *coeff-l p* (*min-var p*) ≠ *0*
x ∈ *vars-l p* ⟹ *abs* (*coeff-l p* (*min-var p*)) ≤ *abs* (*coeff-l p x*)
proof −
let *?m* = *min-var p*
define *xcs* **where** *xcs* = *map* (λ *x*. (*x*,*coeff-l p x*)) (*vars-l-list p*)
define *axcs* **where** *axcs* = *map* (*map-prod id abs*) *xcs*
define *m* **where** *m* = *min-list* (*map snd axcs*)
define *fxs* **where** *fxs* = *filter* (λ *xa*. *snd xa* = *m*) *axcs*
{
fix *x*
assume *x*: *x* ∈ *vars-l p*
let *?c* = *coeff-l p x*
from *x* **have** *cx*: *?c* ≠ *0* **by** *transfer auto*
from *x* **have** (*x*, *?c*) ∈ *set xcs* **unfolding** *xcs-def* **by** *force*
hence *ax*: (*x*, *abs ?c*) ∈ *set axcs* **unfolding** *axcs-def* **by** *force*
hence *map snd axcs* ≠ [] *abs ?c* ∈ *set* (*map snd axcs*) **by** *force+*
with *min-list-Min*[*OF this*(*1*), *folded m-def*]
have *m*: *m* = *Min* (*set* (*map snd axcs*)) *m* ∈ *set* (*map snd axcs*) *m* ≤ *abs ?c*
by *auto*
from *m*(*2*) **have** *m* ∈ *snd* ‘ *set fxs* **unfolding** *fxs-def* **by** *force*
then obtain *y m′ xs* **where** *fxs*: *fxs* = ((*y*,*m′*) # *xs*)
by (*cases fxs*, *auto simp*: *fxs-def*)
hence (*y*,*m′*) ∈ *set fxs* **by** *auto*
from *this*[*unfolded fxs-def*] **have** *m′*: *m′* = *m* **by** *auto*
with *fxs* **have** *fxs*: *fxs* = ((*y*,*m*) # *xs*) **by** *auto*
have *m′*: *?m* = *y*
unfolding *min-var-def Let-def xcs-def*[*symmetric*]
unfolding *axcs-def*[*symmetric*]
unfolding *m-def*[*symmetric*]
unfolding *fxs-def*[*symmetric*]
unfolding *fxs* **by** *simp*
from *fxs* **have** (*y*,*m*) ∈ *set axcs* **unfolding** *fxs-def*
by (*metis Cons-eq-filter-iff in-set-conv-decomp*)
then obtain *c* **where** (*y*,*c*) ∈ *set xcs* **and** *mc*: *m* = *abs c* **unfolding** *axcs-def*
by *auto*
hence *c*: *c* = *coeff-l p y* **and** *y*: *y* ∈ *vars-l p* **unfolding** *xcs-def* **by** *auto*
hence *c0*: *c* ≠ *0* **by** *transfer auto*
show *abs* (*coeff-l p ?m*) ≤ *abs* (*coeff-l p x*)
unfolding *m′* **using** *m*(*3*) **unfolding** *c mc* **.**
have *abs* (*coeff-l p ?m*) ≠ *0* **using** *c0* **unfolding** *c m′* **by** *auto*
}
thus *vars-l p* ≠ {} ⟹ *coeff-l p* (*min-var p*) ≠ *0* **by** *auto*
qed

definition *gcd-coeffs-l* :: (*'a* :: *Gcd*, *'v*)*lpoly* ⇒ *'a* **where**
gcd-coeffs-l p = *Gcd* (*coeff-l p* ‘ *vars-l p*)

lift-definition *change-const* :: *'a* :: *zero* ⇒ (*'a*,*'v*)*lpoly* ⇒ (*'a*,*'v*)*lpoly* **is** λ *x c*. *c*(*None* := *x*)
proof *goal-cases*
case (*1 x c*)
hence *f*: *finite* ((*insert None*) {*v*. *c v* ≠ *0*}) **by** *auto*
show *?case*
by (*rule finite-subset*[*OF* - *f*], *auto*)
qed

lemma *lpoly-fun-of-eqI*: **assumes** ⋀ *x*. *fun-of-lpoly p x* = *fun-of-lpoly q x*
shows *p* = *q*
using *assms* **by** *transfer auto*

lift-definition *reorder-nontriv-var* :: *'v* ⇒ (*int*,*'v*) *lpoly* ⇒ *'v* ⇒ (*int*,*'v*) *lpoly* **is**
λ *x c y*. (λ *z*. *c z div c* (*Some x*))(*Some x* := *1*, *Some y* := −*1*)
proof (*goal-cases*)
case (*1 x c y*)
from *1* **have** *fin*: *finite* (*insert* (*Some y*) (*insert* (*Some x*) ({*v*. *c v* ≠ *0*}))) **by** *auto*
show *?case* **by** (*rule finite-subset*[*OF* - *fin*], *auto*)
qed

lemma *coeff-l-reorder-nontriv-var*: *coeff-l* (*reorder-nontriv-var x p y*)
= (λ *z*. *coeff-l p z div coeff-l p x*)(*x* := *1*, *y* := −*1*)
by (*transfer*, *auto simp*: *Let-def*)

lemma *vars-reorder-non-triv*: *vars-l* (*reorder-nontriv-var x p y*) ⊆ *insert x* (*insert y* (*vars-l p*))
by (*transfer*, *auto simp*: *Let-def*)

end

1.2 An Implementation of Linear Polynomials as Ordered Association Lists

theory *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*)}
by (*intro exI*[*of* - (*0*,[])], *auto*)

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*)
by (*intro ext*, *auto simp*: *lookup-0-def*)

lemma *lookup-0-single*[*simp*]: *lookup-0* [(*x*,*c*)] = (λ *y*. *0*)(*x* := *c*)
by (*intro ext*, *auto simp*: *lookup-0-def*)

lemma *finite-lookup-0*[*simp*, *intro*]: *finite* {*x* . *lookup-0 xs x* ≠ *0*}
unfolding *lookup-0-def*
by (*rule finite-subset*[*OF* - *finite-set*, *of* - *map fst xs*],
force split: *option.splits dest*!: *map-of-SomeD*)

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*
apply *clarsimp*
subgoal for *c vcs*
apply (*rule finite-subset*[*of* - *insert None* (*Some* ‘ {*x*. *lookup-0 vcs x* ≠ *0*})])
subgoal apply (*clarsimp split*: *option.splits*)
subgoal for *x* **by** (*cases x*, *auto*)
done
subgoal by *simp*
done
done

code-datatype *lpoly-of*

lift-definition *zero-lpoly-impl* :: ($'a$ :: *zero*, $'v$ :: *linorder*) *lpoly-impl* **is**
(*0*,[]) **by** *auto*

lemma *zero-lpoly-impl*[*code*]: *0* = *lpoly-of zero-lpoly-impl*
by (*transfer*, *auto split*: *option.splits*)

lift-definition *const-lpoly-impl* :: $'a$ ⇒ ($'a$ :: *zero*, $'v$ :: *linorder*) *lpoly-impl* **is**
λ *c*. (*c*,[]) **by** *auto*

lemma *const-lpoly-impl*[*code*]: *const-l c* = *lpoly-of* (*const-lpoly-impl c*)
by (*transfer*, *auto split*: *option.splits*)

lift-definition *constant-lpoly-impl* :: ($'a$ :: *zero*, $'v$ :: *linorder*) *lpoly-impl* ⇒ $'a$ **is**
fst **.**

lemma *constant-lpoly-impl*[*code*]: *constant-l* (*lpoly-of p*) = *constant-lpoly-impl p*
  by (*transfer*, *auto*)

lift-definition *var-lpoly-impl* :: ′*v* :: *linorder* ⇒ (′*a* :: {*comm-monoid-mult*,*zero-neq-one*},
′*v*) *lpoly-impl* **is**
  λ *x*. (*0*, [(*x*,*1*)]) **by** *auto*

lemma *var-lpoly-impl*[*code*]: *var-l x* = *lpoly-of* (*var-lpoly-impl x*)
  by *transfer* (*auto split*: *option.splits*)

lift-definition *uminus-lpoly-impl* :: (′*a* :: *ab-group-add*, ′*v* :: *linorder*) *lpoly-impl*
⇒ (′*a*,′*v*) *lpoly-impl* **is**
  λ (*c*, *vcs*). (*uminus c*, *map* (*map-prod id uminus*) *vcs*)
  by *force*

lemma *uminus-lpoly-impl*[*code*]: − *lpoly-of p* = *lpoly-of* (*uminus-lpoly-impl p*)
 by *transfer* (*force split*: *option.split simp*: *map-of-eq-None-iff lookup-0-def eq-key-imp-eq-value*)

fun *merge-coeffs-main* :: (′*a* :: *zero* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*v* :: *linorder* × ′*a*) *list* ⇒ (′*v*
× ′*a*)*list* ⇒ (′*v* × ′*a*)*list* **where**
  merge-coeffs-main f ((*x*,*c*) # *xs*) ((*y*,*d*) # *ys*) = (
    if x = *y then* (*x*,*f c d*) # *merge-coeffs-main f xs ys*
    else if x < *y then* (*x*,*f c 0*) # *merge-coeffs-main f xs* ((*y*,*d*) # *ys*)
    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* (λ *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*))
  ∧ *distinct* (*map fst* (*merge-coeffs-main f vxs vys*))
  ∧ *fst* ‘ *set* (*merge-coeffs-main f vxs vys*) = *fst* ‘ *set vxs* ∪ *fst* ‘ *set vys*
  ∧ *lookup-0* (*merge-coeffs-main f vxs vys*) *x* = *f* (*lookup-0 vxs x*) (*lookup-0 vys x*)
  using *assms*
proof (*induction f vxs vys rule*: *merge-coeffs-main.induct*)
  case (*1 f x c xs y d ys*)
  let *?lhs* = *merge-coeffs-main f* ((*x*, *c*) # *xs*) ((*y*, *d*) # *ys*)
  consider (*eq*) *x* = *y* | (*lt*) *x* ≠ *y x* < *y* | (*gt*) *x* ≠ *y* ¬ *x* < *y* **by** *linarith*
  thus *?case*
  proof *cases*
    case *eq*
    from *eq 1.prems* **have** *sorted* (*map fst xs*) *distinct* (*map fst xs*)
      sorted (*map fst ys*) *distinct* (*map fst ys*) *f 0 0* = *0* **by** *auto*
    note *IH* = *1.IH*(*1*)[*OF eq this*]
    from *eq* **have** *res*: *?lhs* = (*x*, *f c d*) # *merge-coeffs-main f xs ys* **by** *auto*
    from *eq 1.prems IH* **show** *?thesis* **unfolding** *res* **using** *IH*
      apply (*intro conjI*)
      subgoal by *auto*

subgoal by *auto*
subgoal by *auto*
subgoal by (*force simp*: *lookup-0-def map-of-eq-None-iff split*: *option.split dest*: *eq-key-imp-eq-value*)
done
next
case *lt*
from *lt 1.prems* **have** *sorted* (*map fst xs*) *distinct* (*map fst xs*)
sorted (*map fst* ((*y*, *d*) # *ys*)) *distinct* (*map fst* ((*y*, *d*) # *ys*)) *f 0 0* = *0* **by** *auto*
note *IH* = *1.IH*(*2*)[*OF lt this*]
from *lt* **have** *res*: *?lhs* = (*x*, *f c 0*) # *merge-coeffs-main f xs* ((*y*, *d*) # *ys*) **by** *auto*
from *lt 1.prems IH* **show** *?thesis* **unfolding** *res* **using** *IH*
apply (*intro conjI*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*force simp*: *lookup-0-def map-of-eq-None-iff split*: *option.split dest*: *eq-key-imp-eq-value*)
done
next
case *gt*
from *gt 1.prems* **have** *sorted* (*map fst* ((*x*, *c*) # *xs*)) *distinct* (*map fst* ((*x*, *c*) # *xs*))
sorted (*map fst ys*) *distinct* (*map fst ys*) *f 0 0* = *0* **by** *auto*
note *IH* = *1.IH*(*3*)[*OF gt this*]
from *gt* **have** *res*: *?lhs* = (*y*, *f 0 d*) # *merge-coeffs-main f* ((*x*, *c*) # *xs*) *ys* **by** *auto*
from *gt 1.prems IH* **show** *?thesis* **unfolding** *res* **using** *IH*
apply (*intro conjI*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*force simp*: *lookup-0-def map-of-eq-None-iff split*: *option.split dest*: *eq-key-imp-eq-value*)
done
qed
next
case (*2 f ys*)
then show *?case*
apply (*intro conjI*)
subgoal by *force*
subgoal by *force*
subgoal by *force*
by (*force simp*: *map-of-eq-None-iff lookup-0-def split*: *option.split dest*: *eq-key-imp-eq-value*)
next
case (*3 f v va*)
then show *?case*

apply (*intro conjI*)
subgoal by *force*
subgoal by *force*
subgoal by *force*
by (*force simp*: *map-of-eq-None-iff lookup-0-def split*: *option.split dest*: *eq-key-imp-eq-value*)
qed

definition *filter-0* **where** *filter-0* = *filter* (λ *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*)
subgoal
apply (*intro ext*)
apply (*clarsimp simp*: *lookup-0-def filter-0-def split*: *option.split*)
apply (*intro conjI impI allI*)
subgoal for *x*
by (*smt* (*verit*, *ccfv-SIG*) *eq-snd-iff map-of-SomeD mem-Collect-eq not-None-eq set-filter weak-map-of-SomeI*)
subgoal for *x y* **by** (*force dest*: *map-of-SomeD simp*: *map-of-eq-None-iff*)
subgoal for *x y z* **using** *assms*
by (*metis* (*no-types*, *lifting*) *eq-key-imp-eq-value map-of-SomeD mem-Collect-eq set-filter*)
done
subgoal using *assms*(*1*) **unfolding** *filter-0-def* **by** (*rule distinct-map-filter*)
subgoal using *assms*(*2*) **unfolding** *filter-0-def* **by** (*rule sorted-filter*)
subgoal unfolding *filter-0-def* **by** *auto*
done

definition *merge-coeffs* :: (*'a* :: *zero* ⇒ *'a* ⇒ *'a*) ⇒ (*'v* :: *linorder* × *'a*) *list* ⇒ (*'v* × *'a*)*list* ⇒ (*'v* × *'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 −
let *?m* = *merge-coeffs-main f vxs vys*
from *merge-coeffs-main*[*OF assms*(*1*−*4*), *of f*, *OF assms*(*5*)]
have *distinct* (*map fst ?m*) *sorted* (*map fst ?m*) *lookup-0 ?m x* = *f* (*lookup-0 vxs x*) (*lookup-0 vys x*)
by *auto*
from *filter-0*[*OF this*(*1*−*2*)] *this*(*3*)

show *?A ?B ?C ?D*
unfolding *merge-coeffs-def*[*symmetric*] **by** *auto*
qed

lift-definition *minus-lpoly-impl* :: ($'a$ :: *ab-group-add*, $'v$ :: *linorder*) *lpoly-impl* ⇒ ($'a$,$'v$) *lpoly-impl* ⇒ ($'a$,$'v$) *lpoly-impl* **is**
λ (*c*, *vxs*) (*d*, *vys*). (*c* − *d*, *merge-coeffs minus vxs vys*)
apply *clarsimp*
subgoal for *vxs vys*
using *merge-coeffs*[*of vxs vys minus*] **by** *auto*
done

lemma *minus-lpoly-impl*[*code*]: *lpoly-of p* − *lpoly-of q* = *lpoly-of* (*minus-lpoly-impl p q*)
apply *transfer*
apply *clarsimp*
apply (*intro ext*)
subgoal for *a vxs b vys x*
using *merge-coeffs*[*of vxs vys minus*]
by (*cases x*, *auto*)
done

lift-definition *plus-lpoly-impl* :: ($'a$ :: *ab-group-add*, $'v$ :: *linorder*) *lpoly-impl* ⇒ ($'a$,$'v$) *lpoly-impl* ⇒ ($'a$,$'v$) *lpoly-impl* **is**
λ (*c*, *vxs*) (*d*, *vys*). (*c* + *d*, *merge-coeffs plus vxs vys*)
apply *clarsimp*
subgoal for *vxs vys*
using *merge-coeffs*[*of vxs vys plus*] **by** *auto*
done

lemma *plus-lpoly-impl*[*code*]: *lpoly-of p* + *lpoly-of q* = *lpoly-of* (*plus-lpoly-impl p q*)
apply *transfer*
apply *clarsimp*
apply (*intro ext*)
subgoal for *a vxs b vys x*
using *merge-coeffs*[*of vxs vys plus*]
by (*cases x*, *auto*)
done

lift-definition *map-lpoly-impl* :: ($'a$ :: *zero* ⇒ $'a$) ⇒ ($'a$,$'v$ :: *linorder*)*lpoly-impl* ⇒ ($'a$,$'v$)*lpoly-impl* **is**
λ *f* (*c*,*vcs*). (*f c*, *filter-0* (*map* (*map-prod id f*) *vcs*))
by *clarsimp* (*intro conjI filter-0*, *auto simp*: *filter-0-def*)

lemma *map-lpoly-impl*: *f 0* = *0* ⟹ *fun-of-lpoly* (*lpoly-of* (*map-lpoly-impl f p*)) = (λ *x*. *f* (*fun-of-lpoly* (*lpoly-of p*) *x*))
apply (*intro ext*)
apply *transfer*

apply *clarsimp*
subgoal for *x f c vcs*
apply (*cases x*)
subgoal by *simp*
subgoal for *y*
apply (*simp add*: *filter-0*)
by (*force simp*: *lookup-0-def map-of-eq-None-iff dest*: *eq-key-imp-eq-value split*: *option.split*)
done
done

definition *sdiv-lpoly-impl p x* = *map-lpoly-impl* (λ *y*. *y div x*) *p*

lemma *sdiv-lpoly-impl*[*code*]: *sdiv-l* (*lpoly-of p*) *x* = *lpoly-of* (*sdiv-lpoly-impl p x*)
apply (*intro lpoly-fun-of-eqI*)
apply (*unfold sdiv-lpoly-impl-def*, *subst map-lpoly-impl*, *force*)
by *transfer auto*

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*)
apply (*intro lpoly-fun-of-eqI*)
apply (*unfold smult-lpoly-impl-def*, *subst map-lpoly-impl*, *force*)
by *transfer auto*

instantiation *lpoly* :: (*type*,*type*)*equal* **begin**
definition *equal-lpoly* :: (′*a*, ′*b*) *lpoly* ⇒ (′*a*, ′*b*) *lpoly* ⇒ *bool* **where** *equal-lpoly* = (=)
instance
by (*intro-classes*, *auto simp*: *equal-lpoly-def*)
end

instantiation *lpoly-impl* :: (*zero*,*linorder*)*equal* **begin**
lift-definition *equal-lpoly-impl* :: (′*a*, ′*b*) *lpoly-impl* ⇒ (′*a*, ′*b*) *lpoly-impl* ⇒ *bool*
is λ (*c*,*xs*) (*d*,*ys*). *c* = *d* ∧ *xs* = *ys* **.**
instance
by (*intro-classes*, *transfer*, *auto*)
end

lift-definition *vars-coeffs-impl* :: (′*a* :: *zero*, ′*v* :: *linorder*) *lpoly-impl* ⇒ (′*v* × ′*a*) *list* **is** *snd* **.**

lemma *vars-coeffs-impl*:
set (*vars-coeffs-impl p*) = (λ *v*. (*v*, *coeff-l* (*lpoly-of p*) *v*)) ‘ *vars-l* (*lpoly-of p*) (**is** *?A*)
distinct (*map fst* (*vars-coeffs-impl p*)) (**is** *?B*)
sorted (*map fst* (*vars-coeffs-impl p*)) (**is** *?C*)
vars-l-list (*lpoly-of p*) = *map fst* (*vars-coeffs-impl p*) (**is** *?D*)

```
  vars-coeffs-impl p = map (λ v. (v, coeff-l (lpoly-of p) v)) (vars-l-list (lpoly-of p))
(is ?E)
proof –
  show ?A ?B ?C
  proof (atomize(full), transfer, goal-cases)
    case (1 p)
    define vcs where vcs = snd p
    with 1 have sort: sorted (map fst vcs) and
      dist: distinct (map fst vcs) and
      non0: ∀ y∈set vcs. snd y ≠ 0 by auto
    let ?set = (λx. (x, lookup-0 vcs x)) ‘ {x. lookup-0 vcs x ≠ 0}
    {
      fix x c
      {
        assume x: (x,c) ∈ set vcs
        with non0 have c: c ≠ 0 by auto
        with dist x have lookup-0 vcs x = c unfolding lookup-0-def by simp
        hence (x,c) ∈ ?set using c by auto
      }
      moreover
      {
        assume (x,c) ∈ ?set
        hence look: lookup-0 vcs x = c and c: c ≠ 0 by auto
        hence (x,c) ∈ set vcs unfolding lookup-0-def
          by (cases map-of vcs x; force dest: map-of-SomeD)
      }
      ultimately have (x,c) ∈ set vcs ⟷ (x,c) ∈ ?set by auto
    }
    with 1 show ?case unfolding vcs-def by auto
  qed
  show ?D unfolding vars-l-list-def using ‹?A› ‹?B› ‹?C›
   by (metis (no-types, lifting) fst-eqD image-set list.map-comp list.map-ident-strong
o-def sorted-distinct-set-unique sorted-list-of-set.distinct-sorted-key-list-of-set sorted-list-of-set.sorted-sorted-ke
vars-l-list vars-l-list-def)
  show ?E using ‹?A› ‹?B› ‹?C› ‹?D›
    by (smt (verit, ccfv-SIG) fst-conv image-iff list.map-comp list.map-ident-strong
o-def)
qed

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

declare eval-l-def[code del]

lemma eval-lpoly-impl[code]: eval-l α (lpoly-of p) =
  constant-lpoly-impl p + (∑ (x, c) ← vars-coeffs-impl p. c ∗ α x)
  unfolding eval-l-def constant-lpoly-impl
  unfolding vars-coeffs-impl(5)
  unfolding vars-l-list[symmetric]
  apply (subst sum.distinct-set-conv-list)
```

```
  subgoal unfolding vars-l-list-def by simp
  subgoal unfolding map-map o-def split ..
  done

declare substitute-all-l-def[code del]

lemma substitute-all-impl[code]: substitute-all-l σ (lpoly-of p) =
  const-l (constant-lpoly-impl p) + (∑ (x, c) ← vars-coeffs-impl p. smult-l c (σ x))

  unfolding substitute-all-l-def constant-lpoly-impl
  unfolding vars-coeffs-impl(5)
  unfolding vars-l-list[symmetric]
  apply (subst sum.distinct-set-conv-list)
  subgoal unfolding vars-l-list-def by simp
  subgoal unfolding map-map o-def split ..
  done

lemma equal-lpoly-impl[code]: HOL.equal (lpoly-of p) (lpoly-of q) = (p = q)
proof (unfold equal-lpoly-def, standard)
  assume *: lpoly-of p = lpoly-of q
  hence vars-coeffs-impl p = vars-coeffs-impl q
    unfolding vars-coeffs-impl(5) by simp
  moreover from * have constant-l (lpoly-of p) = constant-l (lpoly-of q) by simp
  from this[unfolded constant-lpoly-impl]
  have constant-lpoly-impl p = constant-lpoly-impl q .
  ultimately show p = q by transfer auto
qed auto

fun update-main :: 'v :: linorder ⇒ 'a :: zero ⇒ ('v × 'a) list ⇒ ('v × '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 ` set vcs) ((≠) 0)
  and vcs' = update-main x a vcs
  and a: a ≠ 0
shows sorted (map fst vcs') distinct (map fst vcs') Ball (snd ` set vcs') ((≠) 0)
  fst ` set vcs' = insert x (fst ` set vcs)
  lookup-0 vcs' z = ((lookup-0 vcs)(x := a)) z
  using assms(1−4)
proof (atomize(full), induct vcs arbitrary: vcs')
  case Nil
  thus ?case using a by auto
next
  case (Cons p vcs vcs1)
  obtain y b where p: p = (y,b) by force
  note Cons = Cons[unfolded p list.simps fst-conv]
```

```
  consider (gt) x > y | (lt) x < y | (eq) x = y by fastforce
  thus ?case
  proof cases
    case gt
    define vcs2 where vcs2 = update-main x a vcs
    from gt Cons have vcs1: vcs1 = (y, b) # vcs2 unfolding vcs2-def by auto
    from Cons(2−) have ∗:
      sorted (map fst vcs)
      distinct (map fst vcs)
      ∀ y∈snd ‘ set vcs. 0 ≠ y by auto
    from Cons(1)[OF ∗ vcs2-def] Cons(2−4) a gt
    show ?thesis unfolding p vcs1 by (auto simp: lookup-0-def)
  next
    case lt
    with Cons have vcs1: vcs1 = (x,a) # (y,b) # vcs by auto
    from Cons(2−4) a lt
    show ?thesis unfolding p vcs1 by (auto simp: lookup-0-def)
  next
    case eq
    with Cons have vcs1: vcs1 = (x,a) # vcs by auto
    from Cons(2−4) a eq
    show ?thesis unfolding p vcs1 by (auto simp: lookup-0-def)
  qed
qed

fun update-main-0 :: 'v :: linorder ⇒ ('v × 'a) list ⇒ ('v × '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) ((≠) 0)
  and vcs' = update-main-0 x vcs
shows sorted (map fst vcs') distinct (map fst vcs') Ball (snd ‘ set vcs') ((≠) 0)
  fst ‘ set vcs' = fst ‘ set vcs − {x}
  lookup-0 vcs' z = ((lookup-0 vcs)(x := 0)) z
  using assms(1−4)
proof (atomize(full), induct vcs arbitrary: vcs')
  case Nil
  hence vcs': vcs' = [] by auto
  show ?case unfolding vcs' by auto
next
  case (Cons p vcs vcs1)
  obtain y b where p: p = (y,b) by force
  note Cons = Cons[unfolded p list.simps fst-conv]
  consider (gt) x > y | (lt) x < y | (eq) x = y by fastforce
  thus ?case
  proof cases
    case gt
```

define *vcs2* **where** *vcs2* = *update-main-0 x vcs*
from *gt Cons* **have** *vcs1*: *vcs1* = (*y*, *b*) # *vcs2* **unfolding** *vcs2-def* **by** *auto*
from *Cons*(*2−*) **have** ∗:
sorted (*map fst vcs*)
distinct (*map fst vcs*)
∀ *y*∈*snd* ‘ *set vcs*. *0* ≠ *y* **by** *auto*
from *Cons*(*1*)[*OF* ∗ *vcs2-def*] *Cons*(*2−4*) *gt*
show *?thesis* **unfolding** *p vcs1* **by** (*auto simp*: *lookup-0-def*)
next
case *lt*
with *Cons* **have** *vcs1*: *vcs1* = (*y*,*b*) # *vcs* **by** *auto*
from *Cons*(*2−4*) *lt*
show *?thesis* **unfolding** *p vcs1* **by** (*auto simp*: *lookup-0-def split*: *option.split*)
next
case *eq*
with *Cons* **have** *vcs1*: *vcs1* = *vcs* **by** *auto*
from *Cons*(*2−4*) *eq*
show *?thesis* **unfolding** *p vcs1* **by** (*force simp*: *lookup-0-def split*: *option.split*)
qed
qed

lift-definition *update-lpoly-impl* :: *′v* :: *linorder* ⇒ *′a* :: *zero* ⇒ (*′a*,*′v*)*lpoly-impl* ⇒ (*′a*,*′v*)*lpoly-impl* **is**
λ *x a* (*c*, *vs*). *if a* = *0 then* (*c*, *update-main-0 x vs*) *else* (*c*, *update-main x a vs*)
apply *clarsimp*
subgoal for *x a c vs d vcs*
proof *goal-cases*
case *1*
show *?case*
proof (*cases a* = *0*)
case *True*
hence *vcs*: *vcs* = *update-main-0 x vs* **and** *c*: *c* = *d* **using** *1* **by** *auto*
from *update-main-0*[*OF 1*(*2*) *1*(*3*) - *vcs*] *1*(*4*)
show *?thesis* **using** *c* **by** *auto*
next
case *False*
hence *vcs*: *vcs* = *update-main x a vs* **and** *c*: *c* = *d* **using** *1* **by** *auto*
from *update-main*[*OF 1*(*2*) *1*(*3*) - *vcs False*] *1*(*4*)
show *?thesis* **using** *c* **by** *auto*
qed
qed
done

lemma *update-lpoly-impl*: *fun-of-lpoly* (*lpoly-of* (*update-lpoly-impl x a p*)) = (*fun-of-lpoly* (*lpoly-of p*))(*Some x* := *a*)
apply (*transfer*, *clarsimp*, *intro conjI ext impI*)
subgoal for *x a z vs p*
using *update-main-0*(*5*)[*of vs* - *x*, *OF* - - - *refl*]

```
    by (cases p, auto)
  subgoal for x a z vs p
    using update-main(5)[of vs - x a, OF - - - refl]
    by (cases p, auto)
  done

lift-definition coeff-lpoly-impl :: ('a :: zero, 'v :: linorder)lpoly-impl ⇒ 'v ⇒ 'a is
  λ (c,p) x. lookup-0 p x .

lemma coeff-lpoly-impl[code]: coeff-l (lpoly-of p) x = coeff-lpoly-impl p x
  by (transfer, auto)

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)
  unfolding substitute-l-impl-def Let-def
 unfolding plus-lpoly-impl[symmetric] smult-lpoly-impl[symmetric] coeff-lpoly-impl[symmetric]
proof (intro lpoly-fun-of-eqI, goal-cases)
  case (1 y)
  show ?case using update-lpoly-impl[of x 0 q]
    by transfer auto
qed

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)
 unfolding reorder-nontriv-var-impl-def Let-def sdiv-lpoly-impl-def coeff-lpoly-impl[symmetric]
proof (intro lpoly-fun-of-eqI, goal-cases)
  case (1 z)
  show ?case unfolding update-lpoly-impl
    apply (subst map-lpoly-impl, force)
    by transfer auto
qed

declare min-var-def[code del]

lemmas min-var-impl = min-var-def[of lpoly-of p for p,
    folded vars-coeffs-impl(5)]

declare min-var-impl[code]

declare gcd-coeffs-l-def[code del]
```

lemma *Gcd-set*: *Gcd* (*set* (*xs* :: *'a* :: *semiring-Gcd list*)) = *gcd-list xs*
  unfolding *Gcd-set-eq-fold Gcd-fin.set-eq-fold*[*of xs*] **..**

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*
  unfolding *gcd-coeffs-l-def vars-coeffs-impl*(*5*) *map-map o-def snd-conv*
  unfolding *vars-l-list*[*symmetric*] *image-set Gcd-set Gcd-fin.set-eq-fold* **..**

lift-definition *change-const-impl* :: *'a* ⇒ (*'a* :: *zero*, *'v* :: *linorder*)*lpoly-impl* ⇒ (*'a*, *'v*)*lpoly-impl*
  is λ *c* (*d*,*vs*). (*c*,*vs*) **by** *auto*

lemma *change-const-impl*[*code*]: *change-const c* (*lpoly-of p*) = *lpoly-of* (*change-const-impl c p*)
  by (*intro lpoly-fun-of-eqI*, *transfer*, *auto*)

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* ⇒ *'v dleq* ⇒ *bool* **where**
  satisfies-dleq α *p* = (*eval-l* α *p* = *0*)

definition *satisfies-dlineq* :: (*int*,*'v*) *assign* ⇒ *'v dlineq* ⇒ *bool* **where**
  satisfies-dlineq α *p* = (*eval-l* α *p* ≤ *0*)

abbreviation *satisfies-eq-ineqs* :: (*int*,*'v*) *assign* ⇒ *'v dleq set* ⇒ *'v dlineq set* ⇒ *bool* (- $\models_{dio}$ '(-,-')) **where**
  satisfies-eq-ineqs α *eqs ineqs* ≡ *Ball eqs* (*satisfies-dleq* α) ∧ *Ball ineqs* (*satisfies-dlineq* α)

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

lemma *trivial-ineq-None*: *trivial-ineq c* = *None* ⟹ *vars-l c* ≠ {}
  unfolding *trivial-ineq-def* **unfolding** *vars-l-list*[*symmetric*] **by** *fastforce*

lemma *trivial-ineq-Some*: **assumes** *trivial-ineq c* = *Some b*
  shows *b* = *satisfies-dlineq* α *c*
proof −
  from *assms*[*unfolded trivial-ineq-def*] **have** *vars*: *vars-l c* = {} **and** *b*: *b* = (*constant-l c* $\leq$ *0*)
    by (*auto split*: *if-splits simp*: *vars-l-list-def*)
  show *?thesis* **unfolding** *satisfies-dlineq-def eval-l-def vars* **using** *b* **by** *auto*
qed

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

lemma *trivial-ineq-filter*: *trivial-ineq-filter cs* = *None* $\Longrightarrow$ ($\nexists$ α. $\alpha \models_{dio}$ ({}, *set cs*))
  trivial-ineq-filter cs = *Some ds* $\Longrightarrow$
    Ball (*set ds*) (λ *c*. *vars-l c* $\neq$ {}) $\wedge$
    ($\alpha \models_{dio}$ ({}, *set cs*) $\longleftrightarrow$ $\alpha \models_{dio}$ ({}, *set ds*)) $\wedge$
    length ds $\leq$ *length cs*
proof (*atomize*(*full*), *induct cs arbitrary*: *ds*)
  case *IH*: (*Cons c cs*)
  let *?t* = *trivial-ineq c*
  consider (*T*) *?t* = *Some True* | (*F*) *?t* = *Some False* | (*V*) *?t* = *None* **by** (*cases ?t*, *auto*)
  thus *?case*
  proof *cases*
    case *F*
    from *trivial-ineq-Some*[*OF F*] *F* **show** *?thesis* **by** *auto*
  next
    case *T*
    from *trivial-ineq-Some*[*OF T*] *T IH* **show** *?thesis* **by** *force*
  next
    case *V*
    from *trivial-ineq-None*[*OF V*] *V IH* **show** *?thesis* **by** *auto*
  qed
qed *simp*

lemma *trivial-lhe*: **assumes** *vars-l p* = {}
  shows *eval-l* α *p* = *constant-l p*
    satisfies-dleq α *p* $\longleftrightarrow$ *p* = *0*
proof −
  show *id*: *eval-l* α *p* = *constant-l p*
    by (*subst eval-l-mono*[*of* {}], *insert assms*, *auto*)
  show *satisfies-dleq* α *p* $\longleftrightarrow$ *p* = *0*
    unfolding *satisfies-dleq-def id* **using** *assms*

```
    apply (transfer)
    by (metis (mono-tags, lifting) Collect-empty-eq not-None-eq)
qed


end
```

3 Tightening

replace $p + c \le 0$ by $p \;/\; g + \lceil\; c \;/\; g \;\rceil \le 0$ 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 dlineq ⇒ 'v dlineq where
  tighten-ineq p = (let g = gcd-coeffs-l p;
       c = constant-l p
     in if g = 1 then p else let d = − ((−c) div g)
       in change-const d (sdiv-l p g))

lemma tighten-ineq: assumes vars-l p ≠ {}
  shows satisfies-dlineq α (tighten-ineq p) = satisfies-dlineq α p
proof (rule ccontr)
  assume contra: ¬ ?thesis
  let ?tp = tighten-ineq p
  define g where g = gcd-coeffs-l p
  define c where c = constant-l p
  note def = tighten-ineq-def[of p, unfolded Let-def, folded g-def, folded c-def]
  define d where d = − (− c div g)
  define mc where mc = −c
  define pg where pg = sdiv-l p g
  define f where f = (∑x∈vars-l pg. coeff-l pg x ∗ α x)
  from contra def have g1: (g = 1) = False by auto
  from def[unfolded this if-False, folded d-def pg-def]
  have tp: ?tp = change-const d pg by auto

  from assms have g0: g ≠ 0 unfolding g-def gcd-coeffs-l-def
    by (transfer, auto)
  have g ≥ 0 unfolding g-def gcd-coeffs-l-def by simp
  with g0 g1 have g: g > 0 by simp
  have p: p = change-const c (smult-l g pg) (is - = ?p)
  proof (intro lpoly-fun-of-eqI, goal-cases)
    case (1 x)
    show ?case
    proof (cases x)
      case None
```

thus *?thesis* **unfolding** *c-def* **by** *transfer auto*
next
case (*Some y*)
hence *fun-of-lpoly* (*change-const c* (*smult-l g pg*)) *x*
$=$ *g* $*$ (*fun-of-lpoly p x div g*) **unfolding** *pg-def* **by** *transfer auto*
also have $\ldots =$ *fun-of-lpoly p x*
proof (*rule dvd-mult-div-cancel*)
have *fun-of-lpoly p x* $\in$ *coeff-l p* ' *vars-l p* $\vee$ *fun-of-lpoly p x* $=$ *0* **unfolding** *Some*
by *transfer auto*
thus *g dvd fun-of-lpoly p x* **using** *g0* **unfolding** *g-def gcd-coeffs-l-def* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed
qed

have *coeff*: *coeff-l ?p x* $=$ *g* $*$ *coeff-l pg x* **for** *x* **by** *transfer auto*
have *coeff'*: *coeff-l ?tp x* $=$ *coeff-l pg x* **for** *x* **unfolding** *tp* **by** *transfer auto*

have *eval-l* α *p* $=$ *constant-l ?p* $+$ ($\sum$ *x*$\in$*vars-l ?p*. *coeff-l ?p x* $*$ α *x*) **unfolding** *p* **unfolding** *eval-l-def* **by** *auto*
also have *constant-l ?p* $=$ *c* **by** *transfer auto*
also have *vars-l ?p* $=$ *vars-l pg* **using** *g0* **by** *transfer auto*
finally have *evalp*: *eval-l* α *p* $=$ *c* $+$ *g* $*$ *f* **unfolding** *f-def coeff sum-distrib-left* **by** (*simp add*: *ac-simps*)

have *eval-l* α *?tp* $=$ *constant-l ?tp* $+$ ($\sum$ *x*$\in$*vars-l ?tp*. *coeff-l ?tp x* $*$ α *x*) **unfolding** *eval-l-def* **by** *auto*
also have *vars-l ?tp* $=$ *vars-l pg* **unfolding** *tp* **by** *transfer auto*
also have *constant-l ?tp* $=$ *d* **unfolding** *tp* **by** *transfer auto*
finally have *eval-tp*: *eval-l* α *?tp* $=$ *d* $+$ *f* **unfolding** *f-def coeff'* **by** *auto*

define *mo* **where** *mo* $=$ *mc mod g*
define *di* **where** *di* $=$ *mc div g*
have *mc*: *mc* $=$ *g* $*$ *di* $+$ *mo* **and** *mo*: *0* $\leq$ *mo mo* $<$ *g* **using** *g* **unfolding** *mo-def di-def* **by** *auto*

have *sat-p*: *satisfies-dlineq* α *p* $=$ ($g * f \leq -c$) **unfolding** *satisfies-dlineq-def evalp* **by** *auto*
have *satisfies-dlineq* α *?tp* $=$ ($f \leq -\, d$) **unfolding** *satisfies-dlineq-def eval-tp* **by** *auto*
also have $\ldots = (g * f \leq g * (-d))$ **using** *g*
by (*smt* (*verit*, *ccfv-SIG*) *mult-le-cancel-left-pos*)
finally have *?thesis* $\longleftrightarrow$ ($g * f \leq -c \longleftrightarrow g * f \leq g * (-d)$) **unfolding** *sat-p* **by** *auto*
also have $\ldots \longleftrightarrow$ *True* **unfolding** *d-def minus-minus mc-def*[*symmetric*] *di-def*[*symmetric*] **unfolding** *mc* **using** *mo*
by (*smt* (*verit*, *del-insts*) *int-distrib*(*4*) *mult-le-cancel-left1*)

finally show *False* **using** *contra* **by** *auto*
qed

definition *tighten-ineqs* :: *'v dlineq list* ⇒ *'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* $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ ({}, *set cs*)
tighten-ineqs cs = *Some ds* $\Longrightarrow$
($\alpha \models_{dio}$ ({}, *set cs*) $\longleftrightarrow \alpha \models_{dio}$ ({}, *set ds*)) $\wedge$
length ds $\leq$ *length cs*
proof (*atomize*(*full*), *goal-cases*)
case *1*
show *?case*
proof (*cases trivial-ineq-filter cs*)
case *None*
thus *?thesis* **unfolding** *tighten-ineqs-def* **using** *trivial-ineq-filter*(*1*)[*OF None*]
by *auto*
next
case (*Some cs'*)
from *Some* **have** *tighten-ineqs cs* = *Some* (*map tighten-ineq cs'*) **unfolding** *tighten-ineqs-def* **by** *auto*
with *trivial-ineq-filter*(*2*)[*OF Some*, *of* α]
show *?thesis* **using** *tighten-ineq*[*of* - α] **by** *auto*
qed
qed

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* ⇒ *int* × *'v dleq* **is**
λ *c*. (*Gcd* (*range c*), λ *x*. *c x div Gcd* (*range c*))
apply *simp*
subgoal by (*rule finite-subset*, *auto*)
done

```
lemma normalize-dleq-gcd: assumes normalize-dleq p = (g,q)
  and p ≠ 0
shows g = Gcd (insert (constant-l p) (coeff-l p ‘ vars-l p))
  and g ≥ 1
  and normalize-dleq q = (1,q)
  using assms
proof (atomize (full), transfer, goal-cases)
  case (1 p g q)
  let ?G = insert (p None) ((λx. p (Some x)) ‘ {x. p (Some x) ≠ 0})
  let ?g = Gcd (range p)
  have Gcd ?G = Gcd (insert 0 ?G) by auto
  also have insert 0 ?G = insert 0 (range p)
  proof −
    {
      fix y
      assume ∗: y ∈ insert 0 (range p) y ∉ insert 0 ?G
      then obtain z where y = p z by auto
      with ∗ have False by (cases z, auto)
    }
    thus ?thesis by auto
  qed
  also have Gcd . . . = Gcd (range p) by auto
  finally have eq: Gcd ?G = ?g .

  from 1 obtain x where px: p x ≠ 0 by auto
  then obtain y where y ∈ range p y ≠ 0 by auto
  hence g0: ?g ≠ 0 by auto
  moreover have ?g ≥ 0 by simp
  ultimately have g1: ?g ≥ 1 by linarith

  from 1 have gg: g = ?g by auto

  let ?gq = Gcd (range q)
  from 1 have q: q = (λx. p x div ?g) by auto
  have dvd: ?g dvd p x for x by auto
  define gp where gp = ?g
  define gq where gq = ?gq
  note hide = gp-def[symmetric] gq-def[symmetric]
  have ?gq ≥ 0 by simp
  then consider (0) ?gq = 0 | (1) ?gq = 1 | (large) ?gq ≥ 2 by linarith
  hence gq1: ?gq = 1
  proof cases
    case 0
    hence range q ⊆ {0} by simp
    moreover from px dvd[of x] have q x ≠ 0 unfolding q
      using dvd-div-eq-0-iff by blast
    ultimately show ?thesis by auto
  next
    case large
```

```
    hence gq0: ?gq ≠ 0 by linarith
    define prod where prod = ?gq * ?g
    {
      fix y
      have ?gq dvd q y by simp
      then obtain fq where qy: q y = ?gq * fq by blast
      from dvd[of y] obtain fp where py: p y = ?g * fp by blast
      have prod dvd p y using fun-cong[OF q, of y] py qy gq0 g0 unfolding hide
prod-def by auto
    }
    hence prod dvd Gcd (range p)
      by (simp add: dvd-Gcd-iff)
    from this[unfolded prod-def] g0 gq0 have ?gq dvd 1 by force
    hence abs ?gq = 1 by simp
    with large show ?thesis by simp
  qed simp

  show ?case unfolding gg gq1
    by (intro conjI g1 eq[symmetric], auto)
qed


lemma vars-l-normalize: normalize-dleq p = (g,q) ⟹ vars-l q = vars-l p
proof (transfer, goal-cases)
  case (1 c g q)
  {
    fix x
    assume c (Some x) ≠ 0
    moreover have Gcd (range c) dvd c (Some x) by simp
    ultimately have c (Some x) div Gcd (range c) ≠ 0 by fastforce
  }
  thus ?case using 1 by auto
qed


lemma eval-normalize-dleq: normalize-dleq p = (g,q) ⟹ eval-l α p = g * eval-l
α q
proof (subst (1 2) eval-l-mono[of vars-l p], goal-cases)
  case 1 show ?case by force
  case 2 thus ?case using vars-l-normalize by auto
  case 3 thus ?case by force
  case 4 thus ?case
  proof (transfer, goal-cases)
    case (1 c g d α)
    show ?case
    proof (cases range c ⊆ {0})
      case True
      hence c x = 0 for x using 1 by auto
```

thus *?thesis* **using** *1* **by** *auto*
next
case *False*
let *?g* = *Gcd* (*range c*)
from *False* **have** *gcd*: *?g* $\neq$ *0* **by** *auto*
hence *mult*: *c x div ?g* $*$ *?g* = *c x* **for** *x* **by** *simp*
let *?expr* = *c None div ?g* + ($\sum$*x* | *c* (*Some x*) $\neq$ *0*. *c* (*Some x*) *div ?g* $*$ α *x*)
have *?g* $*$ *?expr* = *?expr* $*$ *?g* **by** *simp*
also have ... = *c None* + ($\sum$*x* | *c* (*Some x*) $\neq$ *0*. *c* (*Some x*) $*$ α *x*)
unfolding *distrib-right mult sum-distrib-right*
by (*simp add*: *ac-simps mult*)
finally show *?thesis* **using** *1*(*3*) **by** *auto*
qed
qed
qed

lemma *gcd-unsat-detection*: **assumes** *g* = *Gcd* (*coeff-l p* ‘ *vars-l p*)
and $\neg$ *g dvd constant-l p*
shows $\neg$ *satisfies-dleq* α *p*
proof
assume *satisfies-dleq* α *p*
from *this*[*unfolded satisfies-dleq-def eval-l-def*]
have ($\sum$*x*$\in$*vars-l p*. *coeff-l p x* $*$ α *x*) = $-$ *constant-l p* **by** *auto*
hence ($\sum$*x*$\in$*vars-l p*. *coeff-l p x* $*$ α *x*) *dvd constant-l p* **by** *auto*
moreover have *g dvd* ($\sum$*x*$\in$*vars-l p*. *coeff-l p x* $*$ α *x*)
unfolding *assms* **by** (*rule dvd-sum*, *simp*)
ultimately show *False* **using** *assms* **by** *auto*
qed

lemma *substitute-l-in-equation*: **assumes** α *x* = *eval-l* α *p*
shows *eval-l* α (*substitute-l x p q*) = *eval-l* α *q*
satisfies-dleq α (*substitute-l x p q*) $\longleftrightarrow$ *satisfies-dleq* α *q*
proof $-$
show *eval-l* α (*substitute-l x p q*) = *eval-l* α *q*
unfolding *eval-substitute-l* **unfolding** *assms*(*1*)[*symmetric*] **by** *auto*
thus *satisfies-dleq* α (*substitute-l x p q*) $\longleftrightarrow$ *satisfies-dleq* α *q*
unfolding *satisfies-dleq-def* **by** *auto*
qed

type-synonym *'v dleq-sf* = *'v* $\times$ (*int*,*'v*)*lpoly*

fun *satisfies-dleq-sf*:: (*int*,*'v*) *assign* $\Rightarrow$ *'v dleq-sf* $\Rightarrow$ *bool* **where**
satisfies-dleq-sf α (*x*,*p*) = (α *x* = *eval-l* α *p*)

type-synonym *'v dleq-system* = *'v dleq-sf set* $\times$ *'v dleq set*

fun *satisfies-system* :: (*int*,*'v*) *assign* $\Rightarrow$ *'v dleq-system* $\Rightarrow$ *bool* **where**

satisfies-system α (S,E) = (*Ball* S (*satisfies-dleq-sf* α) $\wedge$ *Ball* E (*satisfies-dleq* α))

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

definition *reorder-for-var* **where**
reorder-for-var x p = (*if coeff-l* p x = *1 then* $-$ (p $-$ *var-l* x) *else* p + *var-l* x)

lemma *reorder-for-var*: **assumes** *abs* (*coeff-l* p x) = *1*
shows *satisfies-dleq* α p $\longleftrightarrow$ *satisfies-dleq-sf* α (x, *reorder-for-var* x p) (**is** *?prop1*)
vars-l (*reorder-for-var* x p) = *vars-l* p $-$ $\{x\}$ (**is** *?prop2*)
proof $-$
from *assms* **have** *coeff-l* p x = *1* $\vee$ *coeff-l* p x = $-$*1* **by** *auto*
hence *?prop1* $\wedge$ *?prop2*
proof
assume *1*: *coeff-l* p x = *1*
hence *res*: *reorder-for-var* x p = $-$ (p $-$ *var-l* x) **unfolding** *reorder-for-var-def* **by** *auto*
have *?prop2* **unfolding** *res vars-l-uminus* **using** *1* **by** *transfer auto*
moreover have *?prop1* **unfolding** *satisfies-dleq-def res satisfies-dleq-sf.simps* **by** *auto*
ultimately show *?thesis* **by** *auto*
next
assume *m1*: *coeff-l* p x = $-$*1*
hence *res*: *reorder-for-var* x p = p + *var-l* x **unfolding** *reorder-for-var-def* **by** *auto*
have *?prop2* **unfolding** *res* **using** *m1* **by** *transfer auto*
moreover have *?prop1* **unfolding** *satisfies-dleq-def res satisfies-dleq-sf.simps* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
thus *?prop1 ?prop2* **by** *blast+*
qed

lemma *reorder-nontriv-var-sat*: $\exists$ a. *satisfies-dleq* (α(y := a)) (*reorder-nontriv-var* x p y)
proof $-$
define X **where** X = *insert* x (*vars-l* p) $-$ $\{y\}$
have X: *finite* X $y \notin X$ *insert* x (*insert* y (*vars-l* p)) = *insert* y X **unfolding** *X-def* **by** *auto*
have *sum*: *sum* f (*insert* x (*insert* y (*vars-l* p))) = f y + *sum* f X **for** f :: - $\Rightarrow$ *int*
unfolding X **using** X(*1*$-$*2*) **by** *simp*
show *?thesis*
unfolding *satisfies-dleq-def*
apply (*subst eval-l-mono*[*of insert* x (*insert* y (*vars-l* p))])
apply *force*

apply (*rule vars-reorder-non-triv*)
apply (*unfold sum*)
apply (*subst* (*1*) *coeff-l-reorder-nontriv-var*)
apply (*subst sum.cong*[*OF refl*, *of - -* λ *z. coeff-l* (*reorder-nontriv-var x p y*) *z* $*$ α *z*])
subgoal using *X* **by** *auto*
subgoal by *simp algebra*
done
qed

lemma *reorder-nontriv-var*: **assumes** *a*: *a* = *coeff-l p x a* $\neq$ *0*
and *y*: *y* $\notin$ *vars-l p*
and *q*: *q* = *reorder-nontriv-var x p y*
and *e*: *e* = *reorder-for-var x q*
and *r*: *r* = *substitute-l x e p*
shows *fun-of-lpoly r* = (λ *z. fun-of-lpoly p z mod a*)(*Some x* := *0*, *Some y* := *a*)
constant-l r = *constant-l p mod a*
coeff-l r = (λ *z. coeff-l p z mod a*)(*x* := *0*, *y* := *a*)
proof –
from *a* **have** *xv*: *x* $\in$ *vars-l p* **by** (*transfer*, *auto*)
with *y* **have** *xy*: *x* $\neq$ *y* **by** *auto*
from *q* **have** *q*: *fun-of-lpoly q* = (λ*z. fun-of-lpoly p z div a*)(*Some x* := *1*, *Some y* := $-$ *1*)
unfolding *a* **by** *transfer*
hence *fun-of-lpoly e* = (λ*z.* $-$ (*fun-of-lpoly p z div a*))(*Some x* := *0*, *Some y* := *1*)
unfolding *e reorder-for-var-def* **using** *xy*
by (*transfer*, *auto*)
thus *main*: *fun-of-lpoly r* = (λ *z. fun-of-lpoly p z mod a*)(*Some x* := *0*, *Some y* := *a*)
unfolding *r* **using** *a xy y*
by (*transfer*, *auto simp*: *minus-mult-div-eq-mod*)
from *main* **show** *constant-l r* = *constant-l p mod a* **by** *transfer auto*
from *main* **show** *coeff-l r* = (λ *z. coeff-l p z mod a*)(*x* := *0*, *y* := *a*) **by** *transfer auto*
qed

inductive *griggio-equiv-step* :: *'v dleq-system* $\Rightarrow$ *'v dleq-system* $\Rightarrow$ *bool* **where**
griggio-solve: *abs* (*coeff-l p x*) = *1* $\Longrightarrow$ *e* = *reorder-for-var x p* $\Longrightarrow$
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*: *normalize-dleq p* = (*g*,*q*) $\Longrightarrow$ *g* $\geq$ *1* $\Longrightarrow$
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* $\cup$ $\bigcup$ (*vars-l* ‘ (*snd* ‘ *S* $\cup$ *E*))

```
lemma griggio-equiv-step: assumes griggio-equiv-step SE TF
  shows (satisfies-system α SE ⟷ satisfies-system α TF) ∧
        (invariant-system SE ⟶ invariant-system TF) ∧
        vars-system TF ⊆ vars-system SE
  using assms
proof induction
  case ∗: (griggio-solve p x e S E)
  from ∗(1) have xp: x ∈ vars-l p by transfer auto
  let ?E = insert p E
  let ?T = insert (x, e) (map-prod id (substitute-l x e) ‘ S)
  let ?F = substitute-l x e ‘ E
  note reorder = reorder-for-var[OF ∗(1), folded ∗(2)]
  from reorder(1)[of α]
  have satisfies-system α (S, ?E) = satisfies-system α (insert (x,e) S, E)
    unfolding satisfies-system.simps by auto
  also have . . . = satisfies-system α (?T, ?F)
  proof (cases α x = eval-l α e)
    case True
    from substitute-l-in-equation[OF this] show ?thesis by auto
  qed auto
  finally have equiv: satisfies-system α (S, ?E) = satisfies-system α (?T, ?F) .
  moreover {
    assume inv: invariant-system (S, ?E)
    have invariant-system (?T, ?F)
      unfolding invariant-system.simps
    proof (intro ballI)
      fix y
      assume y: y ∈ fst ‘ ?T
      from vars-substitute-l[of x e, unfolded reorder]
       have vars-subst: vars-l (substitute-l x e q) ⊆ vars-l p − {x} ∪ (vars-l q −
{x}) for q by auto
      from y have y: y = x ∨ x ≠ y ∧ y ∈ fst ‘ S by force
      thus y ∉ ⋃ (vars-l ‘ (snd ‘ ?T ∪ ?F)) ∧ (∃!f. (y, f) ∈ ?T)
      proof
        assume y: y = x
         hence y ∉ ⋃ (vars-l ‘ (snd ‘ ?T ∪ ?F)) using vars-subst reorder(2) by
auto
        moreover have ∃!f. (y, f) ∈ ?T unfolding y
        proof (intro ex1I[of - e])
          fix f
          assume xf: (x, f) ∈ ?T
          show f = e
          proof (rule ccontr)
            assume f ≠ e
            with xf have x ∈ fst ‘ S by force
            from inv[unfolded invariant-system.simps, rule-format, OF this]
            have x ∉ vars-l p by auto
            with ∗(1) show False by transfer auto
```

qed
qed *force*
ultimately show *?thesis* **by** *auto*
next
assume $x \neq y \wedge y \in fst$ ‘ S
hence *xy*: $x \neq y$ **and** *y*: $y \in fst$ ‘ S **by** *auto*
from *inv*[*unfolded invariant-system.simps*, *rule-format*, *OF y*]
have *nmem*: $y \notin \bigcup$ (*vars-l* ‘ (*snd* ‘ $S \cup$ *insert p E*)) **and** *unique*: ($\exists!f$. (y, f) $\in S$) **by** *auto*
from *unique* **have** $\exists!f$. (y, f) $\in$ *?T* **using** *xy* **by** *force*
moreover from *nmem reorder*(*2*) **have** $y \notin$ *vars-l e* **by** *auto*
with *nmem vars-substitute-l*[*of x e*]
have $y \notin \bigcup$ (*vars-l* ‘ (*snd* ‘ *?T* $\cup$ *?F*)) **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
qed
}
moreover
have *vars-system* (*?T*, *?F*) $\subseteq$ *vars-system* (*S*, *?E*)
using *reorder*(*2*) *vars-substitute-l*[*of x e*] *xp* **unfolding** *vars-system.simps*
by (*auto simp*: *rev-image-eqI*) *blast*
ultimately show *?case* **by** *auto*
next
case $*$: (*griggio-normalize p g q S E*)
from *vars-l-normalize*[*OF* $*$(*1*)] **have** *vars*[*simp*]: *vars-l q* = *vars-l p* **by** *auto*
from *eval-normalize-dleq*[*OF* $*$(*1*)] $*$(*2*)
have *sat*[*simp*]: *satisfies-dleq* α *p* = *satisfies-dleq* α *q* **unfolding** *satisfies-dleq-def* **by** *auto*
show *?case* **by** *simp*
next
case *griggio-trivial*
show *?case* **by** (*simp add*: *satisfies-dleq-def*)
qed

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

lemma *griggio-unsat*: **assumes** *griggio-unsat p*
shows $\neg$ *satisfies-system* α (*S*, *insert p E*)
using *assms*
proof *induction*
case (*griggio-gcd-unsat p*)
from *gcd-unsat-detection*[*OF refl this*]
show *?case* **by** *auto*
next
case (*griggio-constant-unsat p*)
hence *eval-l* α $p \neq 0$ **for** α

```
    unfolding eval-l-def
  proof (transfer, goal-cases)
    case (1 p α)
    from 1(3) obtain x where p x ≠ 0 by auto
    with 1 show ?case by (cases x, auto)
  qed
  thus ?case by (auto simp: satisfies-dleq-def)
qed

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

definition solution-subst :: 'v dleq-sf list ⇒ ('v ⇒ (int,'v)lpoly) where
  solution-subst S x = (case map-of S x of Some p ⇒ p | None ⇒ 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
     ∧ vars-system SF ⊆ V ∪ X
     ∧ V ∩ X = {}
     ∧ (∀ α. (satisfies-system α SF ⟶ Ball E (satisfies-dleq α))
          ∧ (Ball E (satisfies-dleq α) ⟶ (∃ β. satisfies-system β SF ∧ (∀ x. x ∉
X ⟶ α x = β x)))))
| invariant-state None = (∀ α. ¬ Ball E (satisfies-dleq α))


inductive-set griggio-step :: ('v dleq-system × 'v set) option rel where
  griggio-eq-step: griggio-equiv-step SF TG ⟹ (Some (SF,X), Some (TG, X)) ∈
griggio-step
| griggio-fail-step: griggio-unsat p ⟹ (Some ((S,insert p F),X), None) ∈ grig-
gio-step
| griggio-complex-step: coeff-l p x ≠ 0
    ⟹ q = reorder-nontriv-var x p y
    ⟹ e = reorder-for-var x q
    ⟹ y ∉ V ∪ X
    ⟹ (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))
       ∈ griggio-step

lemma griggio-step: assumes (A,B) ∈ griggio-step
  and invariant-state A
shows invariant-state B
  using assms
proof (induct rule: griggio-step.induct)
```

case $*$: (*griggio-eq-step SF TG X*)
from *griggio-equiv-step*[*OF* $*$(*1*)] $*$(*2*)
show *?case* **by** *auto*
next
case $*$: (*griggio-fail-step p S F X*)
from *griggio-unsat*[*OF* $*$(*1*)]
have $\neg$ *satisfies-system* α (*S*, *insert p F*) **for** α **by** *auto*
with $*$(*2*)[*unfolded invariant-state.simps*] **have** $\neg$ *Ball E* (*satisfies-dleq* α) **for** α
by *blast*
then show *?case* **by** *auto*
next
case $*$: (*griggio-complex-step p x q y e X S F*)
have *sat*: $\exists\, a.$ *satisfies-dleq* (α($y := a$)) *q* **for** α
using *reorder-nontriv-var-sat*[*of - y x p*] $*$(*2*) **by** *auto*
have *invariant-state* (*Some* ((*S*, *insert p F*), *X*)) **by** *fact*
note *inv* = *this*[*unfolded invariant-state.simps*]
let *?F* = *insert q* (*insert p F*)
let *?Y* = *insert y X*
let *?T* = *insert* (*x*, *e*) (*map-prod id* (*substitute-l x e*) ‘ *S*)
let *?G* = *substitute-l x e* ‘ *insert p F*
define *SF* **where** *SF* = (*S*,*?F*)
define *TG* **where** *TG* = (*?T*,*?G*)
define *Y* **where** *Y* = *?Y*
from *inv* $*$ **have** *y*: $y \notin$ *vars-system* (*S*, *insert p F*) **by** *blast*
have *inv'*: *invariant-state* (*Some* ((*S*, *?F*), *?Y*))
unfolding *invariant-state.simps*
proof (*intro allI conjI impI*)
from *inv* ‹$y \notin V \cup X$›
show $V \cap$ *insert y X* = {} **by** *auto*
from $*$(*1*) **have** *xp*: $x \in$ *vars-l p* **by** *transfer auto*
with *vars-reorder-non-triv*[*of x p y*, *folded* $*$(*2*)]
have *vq*: *vars-l q* $\subseteq$ *insert y* (*vars-l p*) **by** *auto*
from *inv* **have** *vSF*: *vars-system* (*S*, *insert p F*) $\subseteq V \cup X$ **by** *auto*
with *vq* **show** *vars-system* (*S*, *insert q* (*insert p F*)) $\subseteq V \cup$ *insert y X* **by** *auto*
{
fix α
assume *satisfies-system* α (*S*, *insert q* (*insert p F*))
hence *satisfies-system* α (*S*, *insert p F*) **by** *auto*
with *inv* **show** *Ball E* (*satisfies-dleq* α) **by** *blast*
}
{
fix α
assume *Ball E* (*satisfies-dleq* α)
with *inv* **obtain** β **where** *sat2*: *satisfies-system* β (*S*, *insert p F*)
and *eq*: $\bigwedge z.\ z \notin X \Longrightarrow \alpha\ z = \beta\ z$ **by** *blast*
from *sat*[*of* β] **obtain** *a* **where** *sat3*: *satisfies-dleq* (β($y := a$)) *q* **by** *auto*
let *?*β = β($y := a$)
show $\exists\, \beta.$ *satisfies-system* β (*S*, *?F*) $\wedge$ ($\forall z.\ z \notin$ *?Y* $\longrightarrow \alpha\ z = \beta\ z$)
proof (*intro exI*[*of - ?*β] *conjI allI impI*)

```
        show z ∉ ?Y ⟹ α z = ?β z for z
          using eq[of z] by auto
      have satisfies-system ?β (S, ?F) = satisfies-system ?β (S, insert p F) using
sat3 by auto
        also have ... = satisfies-system β (S, insert p F)
          unfolding satisfies-system.simps
        proof (intro arg-cong2[of - - - - conj] ball-cong refl)
          fix r
          assume r ∈ insert p F
          with y have y ∉ vars-l r by auto
          thus satisfies-dleq ?β r = satisfies-dleq β r
            unfolding satisfies-dleq-def
            by (subst eval-l-cong[of - ?β β], auto)
        next
          fix zr
          assume zr ∈ S
          then obtain z r where zr: zr = (z,r) and (z,r) ∈ S by (cases zr, auto)
          hence insert z (vars-l r) ⊆ V ∪ X using vSF by force
          with ∗(4) have z ≠ y and y ∉ vars-l r by auto
          thus satisfies-dleq-sf ?β zr = satisfies-dleq-sf β zr
            unfolding satisfies-dleq-sf.simps zr
            by (subst eval-l-cong[of - ?β β], auto)
        qed
        also have ... by fact
        finally show satisfies-system ?β (S, ?F) .
      qed
    }
    from inv have invariant-system (S, insert p F) by auto
    with y vq
    show invariant-system (S, ?F) by auto
  qed
  have step: griggio-equiv-step (S, ?F) (?T, ?G)
  proof (intro griggio-equiv-step.intros(1) ∗(3))
    show |coeff-l q x| = 1 unfolding ∗(2) coeff-l-reorder-nontriv-var by simp
  qed
  from griggio-equiv-step[OF this] inv'
  show ?case unfolding SF-def[symmetric] TG-def[symmetric] Y-def[symmetric]
by auto
qed

context
  assumes VE: ⋃ (vars-l ‘ E) ⊆ V
begin

lemma griggio-steps: assumes (Some (({},E),{}), SFO) ∈ griggio-step^∗ (is (?I,-)
∈ -)
  shows invariant-state SFO
proof −
  define I where I = ?I
```

have *inv*: *invariant-state I* **unfolding** *I-def* **using** *VE* **by** *auto*
from *assms*[*folded I-def*]
show *?thesis*
proof (*induct*)
case *base*
then show *?case* **using** *inv* **.**
next
case *step*
then show *?case* **using** *griggio-step*[*OF step*(*2*)] **by** *auto*
qed
qed

lemma *griggio-fail*: **assumes** (*Some* (({},*E*),{}), *None*) $\in$ *griggio-step*^$*$
shows $\nexists$ α. $\alpha \models_{dio}$ (*E*, {})
proof −
from *griggio-steps*[*OF assms*] **show** *?thesis* **by** *auto*
qed

lemma *griggio-success*: **assumes** (*Some* (({},*E*),{}), *Some* ((*S*,{}),*X*)) $\in$ *griggio-step*^$*$
and β: β = *adjust-assign S-list* α *set S-list* = *S*
shows $\beta \models_{dio}$ (*E*, {})
proof −
obtain *LV RV* **where** *LV*: *LV* = *fst* ‘ *S*
and *RV*: *RV* = $\bigcup$ (*vars-l* ‘ *snd* ‘ *S*)
by *auto*
have *id*: *satisfies-system* β (*S*, {}) = *Ball S* (*satisfies-dleq-sf* β) **for** β
by *auto*
have *id2*: *vars-system* (*S*, {}) = *LV* $\cup$ *RV*
by (*auto simp*: *LV RV*)
have *id3*: *invariant-system* (*S*, {}) = (*LV* $\cap$ *RV* = {} $\wedge$ ($\forall$ *x*$\in$*LV*. $\exists$!*e*. (*x*, *e*) $\in$ *S*))
by (*auto simp*: *LV RV*)
from *griggio-steps*[*OF assms*(*1*)]
have *invariant-state* (*Some* ((*S*, {}), *X*)) **.**
note *inv* = *this*[*unfolded invariant-state.simps id id2 id3*]
from *inv* **have** *Ball S* (*satisfies-dleq-sf* β) $\Longrightarrow$ *Ball E* (*satisfies-dleq* β)
by *auto*
moreover {
fix *x e*
assume *xe*: (*x*,*e*) $\in$ *S*
hence *x*: *x* $\in$ *LV* **by** (*force simp*: *LV*)
with *inv xe* **have** $\exists$! *e*. (*x*,*e*) $\in$ *S* **by** *force*
with *xe* **have** *map-of S-list x* = *Some e* **unfolding** β(*2*)[*symmetric*]
by (*metis map-of-SomeD weak-map-of-SomeI*)
hence β *x* = *eval-l* α *e* **unfolding** β *adjust-assign-def* **by** *simp*
also have ... = *eval-l* β *e*
proof (*rule eval-l-cong*)

fix y
assume $y \in$ *vars-l e*
with *xe* **have** $y \in RV$ **unfolding** RV **by** *force*
with *inv* **have** $y \notin LV$ **by** *auto*
thus $\alpha\ y = \beta\ y$ **unfolding** $\beta(2)$[*symmetric*] $\beta(1)$ *adjust-assign-def LV*
by (*force split*: *option.splits dest*: *map-of-SomeD*)
qed
finally have *satisfies-dleq-sf* β (x,e) **by** *auto*
}
ultimately show *?thesis* **by** *force*
qed

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*^*
and σ: σ = *solution-subst S-list*
and *S-list*: *set S-list* $= S$
shows

$f\ (\lambda\ i.\ \textit{eval-l}\ \alpha\ (\textit{substitute-all-l}\ \sigma\ (P\ i))) \Longrightarrow$
$\beta = \textit{adjust-assign S-list}\ \alpha \Longrightarrow$
$f\ (\lambda\ i.\ \textit{eval-l}\ \beta\ (P\ i)) \wedge \beta \models_{dio} (E, \{\})$

$f\ (\lambda\ i.\ \textit{eval-l}\ \alpha\ (P\ i)) \wedge \alpha \models_{dio} (E, \{\}) \Longrightarrow$
$(\bigwedge\ i.\ \textit{vars-l}\ (P\ i) \subseteq V) \Longrightarrow$
$\exists\ \gamma.\ f\ (\lambda\ i.\ \textit{eval-l}\ \gamma\ (\textit{substitute-all-l}\ \sigma\ (P\ i)))$

proof −
assume *sol*: $f\ (\lambda\ i.\ \textit{eval-l}\ \alpha\ (\textit{substitute-all-l}\ \sigma\ (P\ i)))$
and β: β = *adjust-assign S-list* α
from *griggio-success*[*OF assms*(*1*) β *S-list*]
have *solE*: $\beta \models_{dio} (E, \{\})$ **by** *auto*
show $f\ (\lambda\ i.\ \textit{eval-l}\ \beta\ (P\ i)) \wedge \beta \models_{dio} (E, \{\})$
proof (*intro conjI*[*OF - solE*])
{
fix i
have *eval-l* α (*substitute-all-l* σ $(P\ i)$) = *eval-l* β $(P\ i)$
unfolding *eval-substitute-all-l*
proof (*rule eval-l-cong*)
fix x
show *eval-l* α $(\sigma\ x) = \beta\ x$ **unfolding** σ β *solution-subst-def adjust-assign-def*

by (*auto split*: *option.splits*)
qed
}
with *sol* **show** $f\ (\lambda\ i.\ \textit{eval-l}\ \beta\ (P\ i))$ **by** *auto*
qed
next
assume *f*: $f\ (\lambda i.\ \textit{eval-l}\ \alpha\ (P\ i)) \wedge \alpha \models_{dio} (E, \{\})$
and *vV*: $\bigwedge i.\ \textit{vars-l}\ (P\ i) \subseteq V$
from *griggio-steps*[*OF assms*(*1*)]
have *invariant-state* (*Some* ((*S*, {}), *X*)) **.**
note *inv* = *this*[*unfolded invariant-state.simps*]
from *f inv* **obtain** γ
where *sat*: *satisfies-system* γ (*S*, {}) **and** *ab*: $\bigwedge x.\ x \notin X \Longrightarrow \alpha\ x = \gamma\ x$ **by** *blast*
from *inv sat* **have** *E*: *Ball E* (*satisfies-dleq* γ) **by** *auto*
{
fix *i*
have $\textit{eval-l}\ \alpha\ (P\ i) = \textit{eval-l}\ \gamma\ (P\ i)$
proof (*rule eval-l-cong*)
fix *x*
show $x \in \textit{vars-l}\ (P\ i) \Longrightarrow \alpha\ x = \gamma\ x$
by (*rule ab*, *insert vV*[*of i*] *inv*, *auto*)
qed
}
with *f* **have** *f*: $f\ (\lambda i.\ \textit{eval-l}\ \gamma\ (P\ i))$ **by** *auto*
{
fix *i*
have $\textit{eval-l}\ (\lambda x.\ \textit{eval-l}\ \gamma\ (\sigma\ x))\ (P\ i) = \textit{eval-l}\ \gamma\ (P\ i)$
proof (*intro eval-l-cong*)
fix *x*
note *defs* = σ *solution-subst-def*
show $\textit{eval-l}\ \gamma\ (\sigma\ x) = \gamma\ x$
proof (*cases* $x \in \textit{fst}$ ‘ *S*)
case *False*
thus *?thesis* **unfolding** *defs S-list*[*symmetric*]
by (*force split*: *option.splits dest*: *map-of-SomeD*)
next
case *True*
then obtain *e* **where** *xe*: $(x,e) \in S$ **by** *force*
have $\exists!\ e.\ (x,e) \in S$ **using** *inv True* **by** *auto*
with *xe* **have** *map-of S-list x* = *Some e* **unfolding** *S-list*[*symmetric*]
by (*metis map-of-SomeD weak-map-of-SomeI*)
hence *id*: $\sigma\ x = e$ **unfolding** *defs* **by** *auto*
show *?thesis* **unfolding** *id* **using** *xe sat* **by** *auto*
qed
qed
}
thus $\exists \gamma.\ f\ (\lambda i.\ \textit{eval-l}\ \gamma\ (\textit{substitute-all-l}\ \sigma\ (P\ i)))$
unfolding *eval-substitute-all-l*

by (*intro exI*[*of* - γ], *insert f*, *auto*)
qed

corollary *griggio-success-equivalence*:
  fixes P :: $'i \Rightarrow (int,'v)lpoly$ **and** f :: $('i \Rightarrow int) \Rightarrow bool$
  assumes (*Some* (({},*E*),{}), *Some* ((*S*,{}),*X*)) ∈ *griggio-step*^*
    and σ: σ = *solution-subst S-list*
    and *S-list*: *set S-list* = *S*
    and *vV*: ⋀ *i*. *vars-l* (*P i*) ⊆ *V*
  shows
    (∃ α. *f* (λ *i*. *eval-l* α (*substitute-all-l* σ (*P i*))))
     ⟷ (∃ α. *f* (λ *i*. *eval-l* α (*P i*)) ∧ *Ball E* (*satisfies-dleq* α))
proof −
  note *main* = *griggio-success-translations*[*OF assms*(*1*,*2*) *S-list*, *of f* - *P*]
  from *main*(*1*)[*OF* - *refl*] *main*(*2*)[*OF* - *vV*]
  show *?thesis* **by** *blast*
qed

end
end

end

4.2 Executable Algorithm

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

definition *simplify-dleq* :: *'v dleq* ⇒ *'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 −
  from *assms*[*unfolded simplify-dleq-def Let-def gcd-coeffs-l-def*]
  have *gcd*: *Gcd* (*coeff-l p* ‘ *vars-l p*) = *0* **and** *const*: *constant-l p* = *0*
    by (*auto split*: *if-splits*)
  from *gcd* **have** *coeff-l p* ‘ *vars-l p* ⊆ {*0*} **by** *auto*
  hence *vars-l p* = {} **by** *transfer auto*
  with *const* **have** *fun-of-lpoly p* = (λ -. *0*)

  proof (*transfer*, *intro ext*, *goal-cases*)
    case (*1 c x*)
    thus *?case* **by** (*cases x*, *auto*)
  qed
  thus $p = 0$ **by** *transfer auto*
qed

lemma *simplify-dleq-fail*: **assumes** *simplify-dleq p* = *Inr False*
  shows *griggio-unsat p*
proof −
  let *?g* = *Gcd* (*coeff-l p* ‘ *vars-l p*)
  from *assms*[*unfolded simplify-dleq-def gcd-coeffs-l-def Let-def*]
  consider (*const*) *?g* = *0 constant-l p* $\neq$ *0*
    | (*gcd*) ¬ (*?g dvd constant-l p*)
    by (*auto split*: *if-splits*)
  thus *?thesis*
  proof *cases*
    case *const*
    from *const* **have** *coeff-l p* ‘ *vars-l p* $\subseteq$ {*0*} **by** *auto*
    hence *vars-l p* = {} **by** *transfer auto*
    moreover from *const* **have** $p \neq 0$ **by** *transfer auto*
    ultimately show *?thesis* **by** (*rule griggio-constant-unsat*)
  next
    case *gcd*
    thus *?thesis* **by** (*rule griggio-gcd-unsat*)
  qed
qed

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* = *sum* (*abs o coeff-l p*) (*vars-l p*)

lemma *size-dleq-pos*: *size-dleq p* ≥ *0* **unfolding** *size-dleq-def* **by** *simp*

lemma *simplify-dleq-keep*: **assumes** *simplify-dleq p* = *Inl q*
  shows
    ∃ *g* ≥ *1*. *normalize-dleq p* = (*g*, *q*)
    size-dleq p ≥ *size-dleq q*
    dleq-normalized q
proof (*atomize* (*full*), *unfold dleq-normalized-def*, *goal-cases*)
  case *1*
  let *?g* = *Gcd* (*coeff-l p* ‘ *vars-l p*)
  from *assms*[*unfolded simplify-dleq-def gcd-coeffs-l-def Let-def*]
  have *g*: *?g* ≠ *0 ?g dvd constant-l p* **and** *p0*: *p* ≠ *0*
    and *choice*: *?g* = *1* ∧ *q* = *p* ∨ *?g* ≠ *1* ∧ *q* = *sdiv-l p ?g*
    by (*auto split*: *if-splits*)
  from *g* **have** *gG*: *?g* = *Gcd* (*insert* (*constant-l p*) (*coeff-l p* ‘ *vars-l p*)) (**is** - =

?G) **by** *auto*
  from *g*(*1*) **have** *g1*: *?g* $\geq$ *1* **by** (*smt* (*verit*) *Gcd-int-greater-eq-0*)
  obtain *g′ q′* **where** *norm*: *normalize-dleq p* = (*g′*, *q′*) **by** *force*
  note *norm-gcd* = *normalize-dleq-gcd*[*OF norm p0*, *folded gG*]
  from *choice* **show** *?case*
  proof
    assume *?g* = *1* $\wedge$ *q* = *p*
    hence *g*: *?g* = *1* **and** *id*: *q* = *p* **by** *auto*
    with *gG* **have** *?G* = *1* **by** *auto*
    with *norm gG norm-gcd* **have** *normalize-dleq p* = (*1*, *q′*) **by** *metis*
    hence *norm*: *normalize-dleq p* = (*1*,*p*) **by** (*transfer*, *auto*)
    show *?thesis* **unfolding** *id* **apply** (*intro conjI exI*[*of - ?g*])
      subgoal unfolding *g* **by** *auto*
      subgoal unfolding *g id* **using** *norm* **by** *auto*
      subgoal by *simp*
      subgoal by (*rule g*)
      done
  next
    note *g′* = *norm-gcd*(*1*)
    assume *?g* $\neq$ *1* $\wedge$ *q* = *sdiv-l p ?g*
    with *g′ g* **have** *g′01*: *g′* $\neq$ *0 g′* $\neq$ *1* **and** *q*: *q* = *sdiv-l p g′* **by** *auto*
    from *norm* **have** *q′*: *q′* = *q* **unfolding** *q*
      by (*transfer*, *auto*)
    note *norm-gcd* = *norm-gcd*[*unfolded q′*]
    note *norm* = *norm*[*unfolded q′*]
    show *?thesis*
    proof (*intro conjI exI*[*of - g′*])
      show *1* $\leq$ *g′* **by** *fact*
      show *normalize-dleq p* = (*g′*, *q*) **by** *fact*
      from *g′01* **have** *abs g′* $\geq$ *1* **by** *linarith*
      hence *abs* (*y div g′*) $\leq$ *abs y* **for** *y*
         by (*smt* (*verit*) *div-by-1 div-nonpos-pos-le0 int-div-less-self norm-gcd*(*2*)
pos-imp-zdiv-nonneg-iff zdiv-mono2-neg)
      hence *le*: |*coeff-l q x*| $\leq$ |*coeff-l p x*| **for** *x* **unfolding** *q* **by** (*transfer*, *auto*)
      have *pq*: *p* = *smult-l g′ q* **unfolding** *q* **using** *norm*
        by (*transfer*, *auto*)
      have *vars*: *vars-l q* = *vars-l p* **unfolding** *pq* **using** *g′01*
        by (*transfer*, *auto*)
      show *size-dleq q* $\leq$ *size-dleq p* **unfolding** *size-dleq-def vars*
        by (*rule sum-mono*, *auto simp*: *le*)
       from *gG* **have** *?g* = *Gcd* (*range* (*fun-of-lpoly p*)) **unfolding** *g′*[*symmetric*]
using *norm*
        by *transfer auto*
      have *g′* = *?g* **by** (*rule g′*)
      also have *coeff-l p ‘ vars-l p* = (*λ x*. *g′* ∗ *x*) *‘ coeff-l q ‘ vars-l p*
        unfolding *pq* **by** *transfer auto*
      also have *vars-l p* = *vars-l q* **by** (*simp add*: *vars*)
      also have *Gcd* ((∗) *g′ ‘ coeff-l q ‘ vars-l q*) = *g′* ∗ *Gcd* (*coeff-l q ‘ vars-l q*)
      by (*metis Gcd-int-greater-eq-0 Gcd-mult abs-of-nonneg linordered-nonzero-semiring-class.zero-le-one*

norm-gcd(2) normalize-int-def order.trans zero-le-mult-iff)
      finally have *abs g′ = abs g′ ∗ abs (Gcd (coeff-l q ‘ vars-l q))* **by** *simp*
      with *g′01* **show** *Gcd (coeff-l q ‘ vars-l q) = 1* **by** *simp*
    qed
  qed
qed

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′ ⇒ map-option (Cons e′) (simplify-dleqs es))*

context *griggio-input*
begin

lemma *simplify-dleqs*: *simplify-dleqs es = None ⟹ (Some ((S,set es ∪ F),X), None) ∈ griggio-step^∗*
  simplify-dleqs es = Some fs ⟹
    (Some ((S,set es ∪ F),X), Some ((S,set fs ∪ F),X)) ∈ griggio-step^∗
    ∧ Ball (set fs) dleq-normalized ∧ length fs ≤ length es ∧
     (length fs < length es ∨ fs = [] ∨ size-dleq (hd fs) ≤ size-dleq (hd es))
proof (*atomize (full), induct es arbitrary: F fs*)
  case (*Cons e es F fs*)
  let *?ST = Some ((S, set (e # es) ∪ F), X)*
  define *ST* **where** *ST = ?ST*
  consider (*F*) *simplify-dleq e = Inr False*
    | (*T*) *simplify-dleq e = Inr True*
    | (*New*) *e′* **where** *simplify-dleq e = Inl e′*
    by (*cases simplify-dleq e, auto*)
  thus *?case*
  proof *cases*
    case *F*
    from *simplify-dleq-fail[OF F]*
    have *griggio-unsat e* **by** *auto*
    from *griggio-fail-step[OF this] F*
    show *?thesis* **by** *auto*
  next
    case *T*
    with *simplify-dleq-0[OF T]*
    have *e: e = 0* **and** *id: simplify-dleqs (e # es) = simplify-dleqs es* **by** *auto*
    with *griggio-eq-step[OF griggio-trivial]*
    have *(?ST, Some ((S, set es ∪ F), X)) ∈ griggio-step* **by** *auto*
    with *Cons[of F fs]* **show** *?thesis* **unfolding** *ST-def[symmetric] id* **by** *fastforce*
  next
    case (*New e′*)
    with *simplify-dleq-keep[OF New]* **obtain** *g* **where** *g: g ≥ 1*

and *norm*: *normalize-dleq e* = (*g*, *e′*)
and *res*: *simplify-dleqs* (*e* # *es*) = *map-option* (*Cons e′*) (*simplify-dleqs es*)
and *e′*: *dleq-normalized e′*
and *size*: *size-dleq e′* ≤ *size-dleq e*
by *auto*
from *griggio-eq-step*[*OF griggio-normalize*[*OF norm g*]]
have (*?ST*, *Some* ((*S*, *set es* ∪ *insert e′ F*), *X*)) ∈ *griggio-step* **by** *auto*
with *Cons*[*of insert e′ F*] *e′ size* **show** *?thesis* **unfolding** *res ST-def*[*symmetric*]

by *force*
qed
qed *simp*

context
fixes *fresh-var* :: *nat* ⇒ ′*v*
begin

partial-function (*option*) *dleq-solver-main*
:: *nat* ⇒ (′*v* × ′*v dleq*) *list* ⇒ ′*v dleq list* ⇒ (′*v* × (*int*,′*v*)*lpoly*) *list option* **where**
dleq-solver-main n s es = (*case simplify-dleqs es of*
None ⇒ *None*
| *Some* [] ⇒ *Some s*
| *Some* (*p* # *fs*) ⇒
let x = *min-var p*; *c* = *abs* (*coeff-l p x*)
in if c = *1 then*
let e = *reorder-for-var x p*;
σ = *substitute-l x e in*
dleq-solver-main n ((*x*, *e*) # *map* (*map-prod id* σ) *s*) (*map* σ *fs*) *else*
let y = *fresh-var n*;
q = *reorder-nontriv-var x p y*;
e = *reorder-for-var x q*;
σ = *substitute-l x e in*
dleq-solver-main (*Suc n*) ((*x*, *e*) # *map* (*map-prod id* σ) *s*) (σ *p* # *map* σ *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* ∩ *V* = {} *inj fresh-var*
and *inv*: *invariant-state* (*state-of n s es*)
shows *dleq-solver-main n s es* = *None* ⟹ (*state-of n s es*, *None*) ∈ *griggio-step*^*

dleq-solver-main n s es = *Some s′* ⟹ ∃ *X*. (*state-of n s es*, *Some* ((*set s′*, {}), *X*)) ∈ *griggio-step*^*
using *inv*
proof (*atomize*(*full*), *induct es arbitrary*: *n s rule*: *wf-induct*[*OF wf-measures*[*of* [*length*, *nat o size-dleq o hd*]]])
case (*1 es n s*)
note *def*[*simp*] = *dleq-solver-main.simps*[*of n s es*]
show *?case*

```
  proof (cases simplify-dleqs es)
    case None
    with simplify-dleqs(1)[OF this, of set s {}]
    show ?thesis by auto
  next
    case (Some es')
    from simplify-dleqs(2)[OF this, of set s {}]
    have steps: (state-of n s es, state-of n s es') ∈ griggio-step*
      and norm: Ball (set es') dleq-normalized
      and size: length es' ≤ length es length es' < length es ∨ es' = [] ∨ size-dleq (hd es') ≤ size-dleq (hd es)
      by auto
    from steps griggio-step 1(2) have inv: invariant-state (state-of n s es')
      by (induct, auto)
    show ?thesis
    proof (cases es')
      case Nil
      with Some steps show ?thesis unfolding def by auto
    next
      case (Cons p fs)
      note steps = steps[unfolded Cons]
      note Some = Some[unfolded Cons]
      note norm = norm[unfolded Cons]
      note size = size[unfolded Cons]
      note inv = inv[unfolded Cons]
      let ?st = state-of n s (p # fs)
      have np: dleq-normalized p using norm by auto
      hence vp: vars-l p ≠ {} unfolding dleq-normalized-def by auto
      hence p0: p ≠ 0 by auto
      define x where x = min-var p
      define c where c = |coeff-l p x|
      from min-var(1)[of p, folded x-def, OF vp] have c0: c > 0 coeff-l p x ≠ 0 unfolding c-def by auto
      note def = def[unfolded Some option.simps list.simps, unfolded Let-def, folded x-def, folded c-def]
      show ?thesis
      proof (cases c = 1)
        case c1: True
        define e where e = reorder-for-var x p
        define σ where σ = substitute-l x e
        from c1 have (c = 1) = True by auto
        note def = def[unfolded this if-True, folded e-def, folded σ-def]
        let ?s' = (x, e) # map (map-prod id σ) s
        let ?fs = map σ fs
        let ?st' = state-of n ?s' ?fs
        have step: (?st, ?st') ∈ griggio-step unfolding state-of.simps
          using griggio-solve[OF c1[unfolded c-def] e-def, folded σ-def]
          by (intro griggio-eq-step, auto)
        note inv' = griggio-step[OF step inv]
```

from *size* **have** (*?fs*, *es*) ∈ *measures* [*length*, *nat* ∘ *size-dleq* ∘ *hd*] **by** *auto*
from *1(1)*[*rule-format*, *OF this inv′*, *folded def*] *steps step*
show *?thesis* **by** (*meson rtrancl.rtrancl-into-rtrancl rtrancl-trans*)
next
case *False*
with *c0* **have** *c1*: *c* > *1* **by** *auto*
define *y* **where** *y* = *fresh-var n*
define *q* **where** *q* = *reorder-nontriv-var x p y*
define *e* **where** *e* = *reorder-for-var x q*
define *σ* **where** *σ* = *substitute-l x e*
have *y*: *y* ∉ *V* ∪ *fresh-var* ' {..<*n*} **using** *fresh-var* **unfolding** *y-def inj-def* **by** *auto*
from *inv y* **have** *yp*: *y* ∉ *vars-l p* **by** *auto*
from *c1* **have** *coeff-l p x* ≠ *0* **unfolding** *c-def* **by** *auto*
note *cσp* = *reorder-nontriv-var*(*1*,*3*)[*OF refl this yp q-def e-def fun-cong*[*OF σ-def*]]
have *fs*: *fresh-var* ' {..<*Suc n*} = *insert y* (*fresh-var* ' {..< *n*})
unfolding *y-def* **using** *lessThan-Suc* **by** *force*
from *c1* **have** (*c* = *1*) = *False* **by** *auto*
note *def* = *def*[*unfolded this if-False*, *folded y-def*, *folded q-def*, *folded e-def*, *folded σ-def*]
let *?s′* = (*x*, *e*) # *map* (*map-prod id σ*) *s*
let *?fs* = *σ p* # *map σ fs*
let *?st′* = *state-of* (*Suc n*) *?s′ ?fs*
have *step*: (*?st*, *?st′*) ∈ *griggio-step* **unfolding** *state-of.simps*
using *griggio-complex-step*[*OF c0(2) q-def e-def y*, *folded σ-def*,*of set s set fs*]
unfolding *fs* **by** *auto*
note *inv′* = *griggio-step*[*OF step inv*]
have (*?fs*, *es*) ∈ *measures* [*length*, *nat* ∘ *size-dleq* ∘ *hd*]
proof (*cases length* (*p* # *fs*) < *length es*)
case *False*
let *?h* = *hd es*
from *False* **have** *len*: *length es* = *Suc* (*length fs*) **and** *ph*: *size-dleq p* ≤ *size-dleq ?h*
using *size* **by** *auto*
have *main*: *size-dleq* (*σ p*) < *size-dleq p*
proof −
define *p′* **where** *p′* = *σ p*
define *m* **where** *m* = *coeff-l p x*
have *m*: *m* ≠ *0* **using** *c0* **unfolding** *m-def* **by** *auto*
from *c1*[*unfolded c-def*] **have** *x*: *x* ∈ *vars-l p* **by** *transfer auto*
have *vars-l p* ≠ {*x*} **using** *np*[*unfolded dleq-normalized-def*] *c1*[*unfolded c-def*]
by *auto*
with *x* **obtain** *z* **where** *z*: *z* ∈ *vars-l p* − {*x*} **by** *auto*
have *cy*: *coeff-l* (*σ p*) *y* = *coeff-l p x* **by** (*simp add*: *cσp*)
with *c0(2)* **have** *y′*: *y* ∈ *vars-l* (*σ p*) **by** *transfer auto*
{

fix u
assume $u \in$ *vars-l* $(\sigma\ p)$
hence *coeff-l* $(\sigma\ p)\ u \neq 0$ **by** (*transfer*, *auto*)
hence $u \neq x \wedge (u \neq y \longrightarrow$ *coeff-l* $p\ u \neq 0)$ **unfolding** $c\sigma p(2)$ **using** *yp x*
by (*auto split*: *if-splits simp*: *m-def*)
hence $u \neq x \wedge (u \neq y \longrightarrow u \in$ *vars-l* $p)$ **by** *transfer auto*
hence $u \in$ *insert* y (*vars-l* p) $- \{x\}$ **by** *auto*
}
hence *vars*: *vars-l* $(\sigma\ p) \subseteq$ *insert* y (*vars-l* p) $- \{x\}$ **by** *auto*
have *yz*: $y \neq z$ **using** *yp z* **by** *auto*
have *size-dleq* $p = c +$ *sum* (*abs* $\circ$ *coeff-l* p) (*vars-l* $p - \{x\}$)
unfolding *size-dleq-def c-def* **by** (*subst sum.remove*[*OF - x*], *auto*)
also have $\ldots = c +$ *abs* (*coeff-l* $p\ z$) $+$ *sum* (*abs* $\circ$ *coeff-l* p) (*vars-l* $p - \{x,z\}$)
by (*subst sum.remove*[*OF - z*], *force*, *subst sum.cong*, *auto*)
finally have *size-one*: *size-dleq* $p = c + |$*coeff-l* $p\ z| +$ *sum* (*abs* $\circ$ *coeff-l* p) (*vars-l* $p - \{x, z\}$) **.**

have *size-dleq* $(\sigma\ p) = c +$ *sum* (*abs* $\circ$ *coeff-l* $(\sigma\ p)$) (*vars-l* $(\sigma\ p) - \{y\}$)
unfolding *size-dleq-def*
by (*subst sum.remove*[*OF - y'*], *auto simp*: *cy c-def*)
also have $\ldots = c + |$*coeff-l* $(\sigma\ p)\ z| +$ *sum* (*abs* $\circ$ *coeff-l* $(\sigma\ p)$) (*vars-l* $(\sigma\ p) - \{y, z\}$)
proof (*cases* $z \in$ *vars-l* $(\sigma\ p) - \{y\}$)
case *True*
show *?thesis* **by** (*subst sum.remove*[*OF - True*], *force*, *subst sum.cong*, *auto*)
next
case *False*
hence $z \notin$ *vars-l* $(\sigma\ p)$ **using** *yz* **by** *auto*
hence *coeff-l* $(\sigma\ p)\ z = 0$ **by** *transfer auto*
with *False* **show** *?thesis* **by** (*subst sum.cong*, *auto*)
qed
also have $\ldots <$ *size-dleq* p
proof $-$
have *id*: *coeff-l* $(\sigma\ p)\ z =$ *coeff-l* $p\ z$ *mod coeff-l* $p\ x$ **unfolding** $c\sigma p$
using *yz z* **by** *auto*
have $|$*coeff-l* $(\sigma\ p)\ z| < c$ **unfolding** *id c-def* **unfolding** *m-def*[*symmetric*] **using** *m*
by (*rule abs-mod-less*)
also have $\ldots \leq |$*coeff-l* $p\ z|$
using *min-var*(*2*)[*of z p*, *folded x-def*, *folded c-def*] **using** *z* **by** *auto*
finally have *less*: $|$*coeff-l* $(\sigma\ p)\ z| < |$*coeff-l* $p\ z|$ **.**

from *yp x* **have** *xy*: $x \neq y$ **by** *auto*
have x': $x \notin$ *vars-l* $(\sigma\ p)$ **using** *fun-cong*[*OF* $c\sigma p(2)$] *xy* **by** *transfer auto*
have *sum* (*abs* $\circ$ *coeff-l* $(\sigma\ p)$) (*vars-l* $(\sigma\ p) - \{y, z\}$)

```
          = sum (abs ∘ coeff-l (σ p)) (vars-l (σ p) − {x, y, z})
        by (rule sum.cong[OF - refl], insert x′, auto)
      also have ... ≤ sum (abs ∘ coeff-l p) (vars-l (σ p) − {x, y, z})
      proof (rule sum-mono, goal-cases)
        case (1 u)
        with vars have uy: u ≠ y and u ∈ vars-l p by auto
        from min-var(2)[OF this(2), folded x-def, folded m-def]
        have |m| ≤ |coeff-l p u| by auto
       thus ?case unfolding o-def fun-cong[OF cσp(2), folded m-def] using
m uy
          by auto (smt (verit, ccfv-threshold) abs-mod-less)
      qed
      also have ... ≤ sum (abs ∘ coeff-l p) (vars-l p − {x, z})
        by (rule sum-mono2, insert vars, auto)
       finally have le: sum (abs ∘ coeff-l (σ p)) (vars-l (σ p) − {y, z}) ≤
sum (abs ∘ coeff-l p) (vars-l p − {x, z}) .

      from le less show ?thesis unfolding size-one by linarith
     qed
     finally show ?thesis .
    qed
    with ph have size-dleq (σ p) < size-dleq ?h by simp
    with len show ?thesis
     using dual-order.strict-trans2 size-dleq-pos by auto
   qed simp
   from 1(1)[rule-format, OF this inv′, folded def] steps step
   show ?thesis
    by (meson rtrancl.rtrancl-into-rtrancl rtrancl-trans)
  qed
 qed
qed
qed

end

end

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

definition fresh-var-gen :: (′v list ⇒ nat ⇒ ′v) ⇒ bool where
 fresh-var-gen fv = (∀ vs. range (fv vs) ∩ set vs = {} ∧ inj (fv vs))


context
 fixes fresh-var :: ′v :: linorder list ⇒ nat ⇒ ′v
begin

definition dleq-solver :: ′v list ⇒ ′v dleq list ⇒ (′v × (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* $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ (*set e*, {})
res = *Some s* $\Longrightarrow$ *adjust-assign s* $\alpha \models_{dio}$ (*set e*, {})
res = *Some s* $\Longrightarrow \sigma$ = *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* $\Longrightarrow \sigma$ = *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*, {}) $\Longrightarrow$
$\exists\ \gamma$. *f* (λ *i. eval-l* γ (*substitute-all-l* σ (*P i*)))
proof −
define *V* **where** *V* = *v* @ *concat* (*map vars-l-list e*)
interpret *griggio-input set V set e* **.**
define *fv* **where** *fv* = *fresh-var V*
from *dleq-solver-def*[*of v e*, *folded V-def*, *folded fv-def*, *unfolded Let-def*,
unfolded assms(*2*)]
have *res*: *res* = *dleq-solver-main fv 0* [] *e* **by** *auto*
from *assms*(*1*)[*unfolded fresh-var-gen-def*, *rule-format*, *of V*, *folded fv-def*]
have *fv*: *range fv* $\cap$ *set V* = {} *inj fv* **by** *auto*
have *eV*: $\bigcup$ (*vars-l* ‘ *set e*) $\subseteq$ *set V* **unfolding** *V-def* **by** *auto*
have *inv*: *invariant-state* (*state-of fv 0* [] *e*)
by (*simp*, *auto simp*: *V-def*)
note *main* = *dleq-solver-main*[*OF fv inv*, *folded res*]
{
assume *res* = *None*
from *main*(*1*)[*OF this*] *griggio-fail*[*OF eV*]
show $\nexists\ \alpha.\ \alpha \models_{dio}$ (*set e*, {}) **by** *auto*
}
{
assume *res*: *res* = *Some s*
from *main*(*2*)[*OF res*] **obtain** *X*
where *steps*: (*Some* (({}, *set e*), {}), *Some* ((*set s*, {}), *X*)) $\in$ *griggio-step**
by *auto*
from *griggio-success*[*OF eV steps refl refl*]
show *adjust-assign s* $\alpha \models_{dio}$ (*set e*, {}) **.**
{
assume *sig*: σ = *solution-subst s*
and *f*: *f* (λ *i. eval-l* α (*substitute-all-l* σ (*P i*)))
and β: β = *adjust-assign s* α
from *griggio-success-translations*(*1*)[*OF eV steps sig refl*, *of f* α *P*, *OF f* β]
show *f* (λ *i. eval-l* β (*P i*)) $\wedge\ \beta \models_{dio}$ (*set e*,{}) **.**
}
{
assume *vars*: $\bigwedge$ *i. vars-l* (*P i*) $\subseteq$ *set v* **and** *sig*: σ = *solution-subst s*

and f: f (λ i. *eval-l* α (P i)) $\wedge$ α $\models_{dio}$ (*set* e,{})
from *vars* **have** $\bigwedge$ i. *vars-l* (P i) $\subseteq$ *set* V **unfolding** *V-def* **by** *auto*
from *griggio-success-translations*(*2*)[*OF eV steps sig refl*, *of f* α *P*, *OF f this*]
show $\exists$ γ. f (λ i. *eval-l* γ (*substitute-all-l* σ (P i))) **.**
}
}
qed

definition *equality-elim-for-inequalities* :: $'v$ *dleq list* $\Rightarrow$ $'v$ *dlineq list* $\Rightarrow$
($'v$ *dleq 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. α $\models_{dio}$ (*set eqs*, {})
res = *Some* (*ineqs'*, *adj*) $\Longrightarrow$ α $\models_{dio}$ ({}, *set ineqs'*) $\Longrightarrow$ (*adj* α) $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) $\Longrightarrow$ $\nexists$ α. α $\models_{dio}$ ({}, *set ineqs'*) $\Longrightarrow$ $\nexists$ α. α $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) $\Longrightarrow$ *length ineqs'* = *length ineqs*
proof –
define v **where** v = *concat* (*map vars-l-list ineqs*)
note *res* = *equality-elim-for-inequalities-def*[*of eqs ineqs*, *unfolded assms*(*2*) *Let-def*, *folded v-def*]
note *solver* = *dleq-solver*[*OF assms*(*1*) *refl*, *of v eqs*]
show *res* = *None* $\Longrightarrow$ $\nexists$ α. α $\models_{dio}$ (*set eqs*, {})
using *solver*(*1*) **unfolding** *res* **by** (*auto split*: *option.splits*)
assume *res* = *Some* (*ineqs'*, *adj*)
note *res* = *res*[*unfolded this*]
from *res* **obtain** s **where** s: *dleq-solver v eqs* = *Some s*
by (*cases dleq-solver v eqs*, *auto*)
define σ **where** σ = *solution-subst s*
note *res* = *res*[*unfolded s option.simps*, *folded* σ*-def*]
from *res* **have** *adj*: *adj* = *adjust-assign s*
and *ineqs'*: *ineqs'* = *map* (*substitute-all-l* σ) *ineqs*
by *auto*
define P **where** P i = (*if* i < *length ineqs then ineqs* ! i *else 0*) **for** i
define f **where** f *xs* = ($\forall$ i < *length ineqs*. *xs* i $\leq$ (*0* :: *int*)) **for** *xs*
note *solver* = *solver*(*3*–*4*)[*OF s* σ*-def*, **where** P = P **and** f = f]
have *vars-l* (P i) $\subseteq$ *set v* **for** i **unfolding** *v-def P-def* **by** (*auto simp*: *set-conv-nth*[*of ineqs*])
note *solver* = *solver*(*1*)[*OF - refl*, *folded adj*] *solver*(*2*)[*OF this*]

```
  have id: f (λi. eval-l α (P i)) = (Ball (set ineqs) (satisfies-dlineq α)) for α
    unfolding f-def P-def set-conv-nth by (auto simp: satisfies-dlineq-def)
  note solver = solver[unfolded id eval-substitute-all-l σ-def]
  from solver(1)[of α]
  show α ⊨dio ({}, set ineqs') ⟹ (adj α) ⊨dio (set eqs, set ineqs)
    unfolding ineqs' σ-def
    by (auto simp: satisfies-dlineq-def eval-substitute-all-l)
  show length ineqs' = length ineqs unfolding ineqs' by simp
  assume no-sol: ∄ α. α ⊨dio ({}, set ineqs')
  show ∄ α. α ⊨dio (set eqs, set ineqs) (is ∄ α. ?Pr α)
  proof
    assume ∃ α. ?Pr α
    then obtain α where ?Pr α by blast
    with solver(2)[of α] obtain γ
      where Ball (set ineqs) (satisfies-dlineq (λx. eval-l γ (solution-subst s x)))
      by blast
    with no-sol show False
      unfolding ineqs' σ-def
      by (auto simp: satisfies-dlineq-def eval-substitute-all-l)
  qed
qed

end


definition fresh-vars-nat :: nat list ⇒ nat ⇒ 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 −
  {
    fix xs x
    assume Suc (Max (insert 0 (set xs)) + x) ∈ insert 0 (set xs)
    from Max-ge[OF - this] have False by auto
  }
  thus ?thesis unfolding fresh-var-gen-def fresh-vars-nat-def Let-def
    by auto
qed

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 (∃ c ∈ set cs. lec-rel c = Lt-Rel) then Lt-Rel else Leq-Rel)
proof (induct cs)
  case Nil
  thus ?case by (auto simp: zero-le-constraint-def)
next
  case (Cons c cs)
  thus ?case by (cases sum-list cs; cases c; cases lec-rel c; auto)
qed


lemma equality-detection-rat: fixes cs :: rat le-constraint set
    and p :: 'i ⇒ linear-poly
    and co :: 'i ⇒ rat
    and I :: 'i set
  defines n ≡ λ i. Le-Constraint Leq-Rel (p i) (co i)
    and s ≡ λ i. Le-Constraint Lt-Rel (p i) (co i)
  assumes fin: finite cs finite I
    and C: C ⊆ cs ∪ s ` I
    and unsat: ∄ v. ∀ c ∈ C. v ⊨le c
    and min: ⋀ D. D ⊂ C ⟹ ∃ v. ∀ c ∈ D. v ⊨le c
    and sol: ∀ c ∈ cs ∪ n ` I. v ⊨le c
    and i: i ∈ I s i ∈ C
  shows (p i)⦃v⦄ = co i
proof –
  have finite ((cs ∪ s ` I) ∩ C) using fin by auto
  with C have finC: finite C by (simp add: inf-absorb2)
  from Motzkin's-transposition-theorem[OF this] unsat
  obtain D const rel where valid: ∀(r, c)∈set D. 0 < r ∧ c ∈ C and
       eq: (∑(r, c)←D. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R lec-const
c)) =
            Le-Constraint rel 0 const
      and ineq: rel = Leq-Rel ∧ const < 0 ∨ rel = Lt-Rel ∧ const ≤ 0 by auto
   let ?expr = (∑(r, c)←D. Le-Constraint (lec-rel c) (r *R lec-poly c) (r *R
lec-const c))
  {
    assume s i ∉ snd ` set D
    with valid have valid: ∀(r, c)∈set D. 0 < r ∧ c ∈ C – {s i}
      by force
    from finC have finite (C – {s i}) by auto
    from Motzkin's-transposition-theorem[OF this] valid eq ineq
    have ∄ v. ∀c∈C – {s i}. v ⊨le c by blast
```

```
    with min[of C − {s i}] i(2) have False by auto
  }
  hence mem: s i ∈ snd ‘ set D by auto
  from i(1) sol have v ⊨le  n i by auto
  from this[unfolded n-def] have piv: (p i) ⦃ v ⦄ ≤ co i by simp
  from ineq have const0: const ≤ 0 by auto
  define I′ where I′ = cs ∪ n ‘ I
  define f where f c = (if c ∈ insert (s i) I′ then c else (n (SOME j. j ∈ I ∧ s j = c))) for c
  let ?C = insert (s i) I′
  {
    fix c
    assume c ∈ C
    hence c: c ∈ cs ∪ s ‘ I using C by auto
    hence f c ∈ ?C ∧ lec-poly (f c) = lec-poly c ∧ lec-const (f c) = lec-const c
    proof (cases c ∈ cs ∪ n ‘ I ∪ {s i})
      case True
      thus ?thesis unfolding f-def I′-def by auto
    next
      case False
      define j where j = (SOME x. x ∈ I ∧ s x = c)
      from False have ∃ j. j ∈ I ∧ s j = c using c by auto
      from someI-ex[OF this, folded j-def] have j: j ∈ I and c: c = s j by auto
       from False have fc: f c = n j unfolding f-def j-def[symmetric] I′-def by auto
      show ?thesis using j c fc by (auto simp: n-def s-def I′-def)
    qed
    hence f c ∈ insert (s i) I′ lec-poly (f c) = lec-poly c lec-const (f c) = lec-const c
      by auto
  } note f = this

  show ?thesis
  proof (rule ccontr)
    assume ¬ ?thesis
    with piv have (p i)⦃ v ⦄ < co i by simp
    hence vsi: v ⊨le s i unfolding s-def by auto
     with sol have sol: (∃ v. ∀ c ∈ insert (s i) I′. v ⊨le c) = True unfolding I′-def by auto
    let ?D = map (map-prod id f) D
    have fin: finite (insert (s i) I′) unfolding I′-def using fin by auto
    from valid f(1)
    have valid′: ∀ (r, c)∈set ?D. 0 < r ∧ c ∈ ?C by force
     let ?expr′ = ∑(r, c)←?D. Le-Constraint (lec-rel c) (r ∗R lec-poly c) (r ∗R lec-const c)
    have lec-const ?expr′ = lec-const ?expr
      unfolding sum-list-lec
      apply simp
      apply (rule arg-cong[of - - sum-list])
```

apply (*rule map-cong*[*OF refl*])
using *f valid* **by** *auto*
also have ... = *const* **unfolding** *eq* **by** *simp*
finally have *const*: *lec-const ?expr′* = *const* **by** *auto*
have *lec-poly ?expr′* = *lec-poly ?expr*
unfolding *sum-list-lec*
apply *simp*
apply (*rule arg-cong*[*of - - sum-list*])
apply (*rule map-cong*[*OF refl*])
using *f valid* **by** *auto*
also have ... = *0* **unfolding** *eq* **by** *simp*
finally have *poly*: *lec-poly ?expr′* = *0* **by** *auto*
from *mem* **obtain** *c* **where** (*c*, *s i*) ∈ *set D* **by** *auto*
hence (*c*, *f* (*s i*)) ∈ *set ?D* **by** *force*
hence *mem*: (*c*, *s i*) ∈ *set ?D* **unfolding** *f-def* **by** *auto*
moreover have *lec-rel* (*s i*) = *Lt-Rel* **unfolding** *s-def* **by** *auto*
ultimately
have *rel*: *lec-rel ?expr′* = *Lt-Rel*
unfolding *lec-rel-sum-list* **using** *split-list*[*OF mem*] **by** *fastforce*
have *eq′*: *?expr′* = *Le-Constraint Lt-Rel 0 const*
using *const poly rel* **by** (*simp add*: *sum-list-lec*)

from *valid′ eq′ Motzkin′s-transposition-theorem*[*OF fin*, *unfolded sol*] *const0*
show *False* **by** *blast*
qed
qed

end

5.2 Algorithm to Detect all Implicit Equalities in $\mathbb{Q}$

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*: $(S,v) \models_{ics} cs \Longrightarrow T \subseteq S \Longrightarrow (T,v) \models_{ics} cs$
by *auto*

lemma *assert-all-simplex-plain-unsat*: **assumes** *invariant-simplex cs J s*
and *assert-all-simplex K s* = *Unsat I*
shows ¬ (*set K* ∪ *J*, *v*) $\models_{ics}$ *set cs*
proof −
from *assert-all-simplex-unsat*[*OF assms*]
show *?thesis* **unfolding** *minimal-unsat-core-def* **by** *force*

qed

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 –
from *check-simplex-unsat*[*OF assms*]
show *?thesis* **unfolding** *minimal-unsat-core-def* **by** *force*
qed

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* $\Longrightarrow$ $v \models_c c \longleftrightarrow$ ($v \models_{le}$ *le-of-constraint c*)
by (*cases c*, *auto simp*: *valuate-uminus*)

lemma *le-of-constraints*: *Ball cs is-no-equality* $\Longrightarrow$ $v \models_{cs}$ *cs* $\longleftrightarrow$ ($\forall$ $c \in cs$. $v \models_{le}$ *le-of-constraint c*)
using *le-of-constraint* **by** *auto*

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

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

lemma *is-equality-iff*: *is-equality c* = (¬ *is-strict c* ∧ ¬ *is-nstrict c*)
by (*cases c*, *auto*)

lemma *is-nstrict-iff*: *is-nstrict c* = (¬ *is-strict c* ∧ ¬ *is-equality c*)
by (*cases c*, *auto*)

fun *make-strict* :: *constraint* ⇒ *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* ⇒ *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* ⇒ *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* ⇒ *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*)
using *assms* **by** (*cases c*, *auto*)+

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
subgoal using *assms poly-const-repr(1)* **by** *simp*
subgoal using *assms poly-const-repr(2)* **unfolding** *image-comp o-def* **by** *auto*
done

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* × *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* ⇒ *nat list* ⇒ *nat list* ⇒ *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* ⇒ *nat list* ⇒ *nat list* ⇒ *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* ⟷ $v \models_{cs}$ *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* ∧ *is-equality* (*cs* ! *i*)}
set ineqs = {*i*. *i* < *length cs* ∧ *is-nstrict* (*cs* ! *i*)}
set sineqs = {*i*. *i* < *length cs* ∧ *is-strict* (*cs* ! *i*)}
set ics =
(λ*i*. (*Ineq i*, *make-ineq* (*cs* ! *i*))) ‘ *set eqs* ∪
(λ*i*. (*FIneq i*, *make-flipped-ineq* (*cs* ! *i*))) ‘ *set eqs* ∪
((λ*i*. (*Ineq i*, *cs* ! *i*)) ‘ *set ineqs* ∪
(λ*i*. (*FIneq i*, *make-flipped-ineq* (*cs* ! *i*))) ‘ *set ineqs* ∪
(λ*i*. (*TmpSIneq i*, *make-strict* (*cs* ! *i*))) ‘ *set ineqs*) ∪
(λ*i*. (*SIneq i*, *cs* ! *i*)) ‘ *set sineqs* (**is** - = *?Large*)
distinct (*eqs* @ *ineqs* @ *sineqs*)
set (*eqs* @ *ineqs* @ *sineqs*) = {*0* ..< *length cs*}
proof −
let *?R* = *Simplex.restrict-to* (*Ineq* ‘ *set ineqs* ∪ *SIneq* ‘ *set sineqs* ∪ *Ineq* ‘ *set eqs* ∪ *FIneq* ‘ *set eqs*) (*set ics*)
let *?n* = *length cs*
let *?I* = *Ineq* ‘ *set ineqs* ∪ *SIneq* ‘ *set sineqs* ∪ *Ineq* ‘ *set eqs* ∪ *FIneq* ‘ *set eqs*
define *ics′* **where** *ics′* = *zip* [*0* ..< *?n*] *cs*
from *init*[*unfolded init-constraints-def Let-def*, *folded ics′-def*]
have *ics*: *ics* = *concat* (*map index-constraint ics′*) **and**
eqs: *eqs* = *map fst* (*filter* (*is-equality* ∘ *snd*) *ics′*) **and**
ineqs: *ineqs* = *map fst* (*filter* (*is-nstrict* ∘ *snd*) *ics′*) **and**
sineqs: *sineqs* = *map fst* (*filter* (*is-strict* ∘ *snd*) *ics′*) **by** *auto*
from *eqs* **show** *eqs′*: *set eqs* = {*i*. *i* < *?n* ∧ *is-equality* (*cs* ! *i*)}
by (*force simp*: *set-zip ics′-def*)
from *ineqs* **show** *ineqs′*: *set ineqs* = {*i*. *i* < *?n* ∧ *is-nstrict* (*cs* ! *i*)}
by (*force simp*: *set-zip ics′-def*)
from *sineqs* **show** *sineqs′*: *set sineqs* = {*i*. *i* < *?n* ∧ *is-strict* (*cs* ! *i*)}
by (*force simp*: *set-zip ics′-def*)
show *set* (*eqs* @ *ineqs* @ *sineqs*) = {*0* ..< *?n*}
unfolding *set-append eqs′ ineqs′ sineqs′*
by (*auto simp*: *is-nstrict-iff*)
show *distinct* (*eqs* @ *ineqs* @ *sineqs*) **unfolding** *distinct-append*
unfolding *ineqs eqs sineqs ics′-def*
by (*auto intro*: *distinct-map-filter simp*: *set-zip is-nstrict-iff*)
(*simp add*: *is-equality-iff*)
from *eqs′* **have** *eqs″*: *i* ∈ *set eqs* ⟹ *index-constraint* (*i*, *cs* ! *i*) =
[(*Ineq i*, *make-ineq* (*cs* ! *i*)), (*FIneq i*, *make-flipped-ineq* (*cs* ! *i*))] **for** *i*
by (*cases cs* ! *i*, *auto*)
from *ineqs′* **have** *ineqs″*: *i* ∈ *set ineqs* ⟹ *index-constraint* (*i*, *cs* ! *i*) =

[(*Ineq* i, *cs* ! i), (*FIneq* i, *make-flipped-ineq* (*cs* ! i)), (*TmpSIneq* i, *make-strict* (*cs* ! i))] **for** i
by (*cases cs* ! i, *auto*)
from *sineqs'* **have** *sineqs''*: $i \in$ *set sineqs* $\Longrightarrow$ *index-constraint* (i, *cs* ! i) =
[(*SIneq* i, *cs* ! i)] **for** i
by (*cases cs* ! i, *auto*)
let *?IC* = λ I. $\bigcup$ (*set* ' *index-constraint* ' (λi. (i, *cs* ! i)) ' I)
have *set ics'* = (λ i. (i, *cs* ! i)) ' {i. i < *?n*} **unfolding** *ics'-def*
by (*force simp*: *set-zip*)
also have {i. i < *?n*} = *set eqs* $\cup$ *set ineqs* $\cup$ *set sineqs*
unfolding *ineqs' eqs' sineqs'*
by (*auto simp*: *is-equality-iff*)
finally have *set ics* = *?IC* (*set eqs* $\cup$ *set ineqs* $\cup$ *set sineqs*) **unfolding** *ics set-concat set-map*
by *auto*
also have ... = *?IC* (*set eqs*) $\cup$ *?IC* (*set ineqs*) $\cup$ *?IC* (*set sineqs*) **by** *auto*
also have *?IC* (*set eqs*) = (λ i. (*Ineq* i, *make-ineq* (*cs* ! i))) ' *set eqs*
$\cup$ (λ i. (*FIneq* i, *make-flipped-ineq* (*cs* ! i))) ' *set eqs*
using *eqs''* **by** *auto*
also have *?IC* (*set ineqs*) = (λ i. (*Ineq* i, *cs* ! i)) ' *set ineqs*
$\cup$ (λ i. (*FIneq* i, *make-flipped-ineq* (*cs* ! i))) ' *set ineqs*
$\cup$ (λ i. (*TmpSIneq* i, *make-strict* (*cs* ! i))) ' *set ineqs*
using *ineqs''* **by** *auto*
also have *?IC* (*set sineqs*) = (λ i. (*SIneq* i, *cs* ! i)) ' *set sineqs*
using *sineqs''* **by** *auto*
finally show *icsL*: *set ics* = *?Large* **by** *auto*
show *fst* ' *set ics* = *?l* **unfolding** *icsL set-map set-append image-Un image-comp o-def fst-conv*
by *auto*
have *distinct* (*map fst ics'*) **unfolding** *ics'-def* **by** *auto*
thus *dist*: *distinct-indices ics* **unfolding** *ics*
proof (*induct ics'*)
case (*Cons ic ics*)
obtain i c **where** *ic*: *ic* = (i,c) **by** *force*
{
fix j
assume j: $j \in$ *fst* ' *set* (*index-constraint* (i, c))
$j \in$ *fst* ' ($\bigcup a \in$ *set ics*. *set* (*index-constraint* a))
from $j(1)$ **have** *ji*: *num-of-index* $j = i$ **by** (*cases* c, *auto*)
from $j(2)$ **obtain** i' c' **where** *ic'*: (i',c') $\in$ *set ics* **and** $j \in$ *fst* ' *set* (*index-constraint* (i',c')) **by** *force*
from *this*(2) **have** *ji'*: *num-of-index* $j = i'$ **by** (*cases* c', *auto*)
with *ji* **have** $i = i'$ **by** *auto*
with *ic' ic Cons*(2) **have** *False* **by** *force*
} **note** *tedious* = *this*
show *?case* **unfolding** *ic distinct-indices-def*
apply (*simp del*: *index-constraint.simps*, *intro conjI*)
subgoal by (*cases* c, *auto*)
subgoal using *Cons* **by** (*auto simp*: *distinct-indices-def*)

subgoal using *tedious* **by** *blast*
done
qed (*simp add*: *distinct-indices-def*)

show $v \models_{cs}$ *cs-of ineqs sineqs eqs* $\longleftrightarrow$ $v \models_{cs}$ *set cs*
proof
assume *v*: $v \models_{cs}$ *cs-of ineqs sineqs eqs*
{
fix *c*
assume $c \in$ *set cs*
then obtain *i* **where** *c*: $c = cs\ !\ i$ **and** *i*: $i <$ *?n* **unfolding** *set-conv-nth* **by** *auto*
hence *ic*: $(i,c) \in$ *set ics′* **unfolding** *ics′-def set-zip* **by** *force*
hence *ics*: *set* (*index-constraint* (i,c)) $\subseteq$ *set ics* **unfolding** *ics* **by** *force*
consider (*e*) *is-equality c* | (*s*) *is-strict c* | (*n*) *is-nstrict c* **by** (*cases c*, *auto*)
hence $v \models_c c$
proof *cases*
case *e*
hence *eqs*: $i \in$ *set eqs* **unfolding** *eqs* **using** *ic* **by** *force*
from *e* **have** {(*FIneq i*, *make-flipped-ineq c*), (*Ineq i*, *make-ineq c*)} $\subseteq$ *set* (*index-constraint* (i,c)) **by** (*cases c*, *auto*)
moreover with *ics* **have** {(*FIneq i*, *make-flipped-ineq c*), (*Ineq i*, *make-ineq c*)} $\subseteq$ *set ics* **by** *auto*
ultimately have {*make-flipped-ineq c*, *make-ineq c*} $\subseteq$ *cs-of ineqs sineqs eqs* **unfolding** *cs-of-def* **using** *eqs*
unfolding *index-of-def* **using** *e* **by** (*cases c*, *force+*)
with *v* **have** $v \models_c$ *make-flipped-ineq c* $v \models_c$ *make-ineq c* **by** *auto*
with *e* **show** *?thesis* **by** (*cases c*, *auto*)
next
case *s*
hence *sineqs*: $i \in$ *set sineqs* **unfolding** *sineqs* **using** *ic* **by** *force*
from *s* **have** (*SIneq i*, *c*) $\in$ *set* (*index-constraint* (i,c)) **by** (*cases c*, *auto*)
moreover with *ics* **have** (*SIneq i*, *c*) $\in$ *set ics* **by** *auto*
ultimately have $c \in$ *cs-of ineqs sineqs eqs* **unfolding** *cs-of-def* **using** *sineqs*
unfolding *index-of-def* **using** *s* **by** (*cases c*, *force+*)
with *v* **show** $v \models_c c$ **by** *auto*
next
case *n*
hence *ineq*: $i \in$ *set ineqs* **unfolding** *ineqs* **using** *ic* **by** *force*
from *n* **have** (*Ineq i*, *c*) $\in$ *set* (*index-constraint* (i,c)) **by** (*cases c*, *auto*)
moreover with *ics* **have** (*Ineq i*, *c*) $\in$ *set ics* **by** *auto*
ultimately have $c \in$ *cs-of ineqs sineqs eqs* **unfolding** *cs-of-def* **using** *ineq*
unfolding *index-of-def* **using** *n* **by** (*cases c*, *force+*)
with *v* **show** $v \models_c c$ **by** *auto*
qed
}
thus $v \models_{cs}$ *set cs* **by** *auto*
next
assume *v*: $v \models_{cs}$ *set cs*

{
fix *c*
assume $c \in$ *cs-of ineqs sineqs eqs*
hence $c \in$ *?R* **unfolding** *cs-of-def index-of-def* **by** *auto*
then obtain *i* **where** *i*: $i \in$ *?I* **and** *ic*: $(i,c) \in$ *set ics* **by** *force*
from *ic*[*unfolded ics*] **obtain** *kd* **where** *ic*: $(i,c) \in$ *set* (*index-constraint kd*) **and** *mem*: $kd \in$ *set ics'* **by** *auto*
from *mem*[*unfolded ics'-def*] **obtain** *k d* **where** *kd*: $kd = (k,d)$ **and** *d*: $d \in$ *set cs* **and** *k*: $k <$ *?n* $d =$ *cs* ! *k*
unfolding *set-conv-nth* **by** *force*
from *v d* **have** *vd*: $v \models_c d$ **by** *auto*
consider (*s*) *j* **where** $i =$ *SIneq j j* $\in$ *set sineqs* | (*e*) *j* **where** $i =$ *Ineq j* $\vee$ $i =$ *FIneq j j* $\in$ *set eqs* | (*n*) *j* **where** $i =$ *Ineq j j* $\in$ *set ineqs*
using *i* **by** *auto*
then have $v \models_c c$
proof *cases*
case *n*
from *ic*[*unfolded n kd*] **have** *j*: $j = k$ **by** (*cases d*, *auto*)
from *n*(*2*)[*unfolded ineqs j*] **obtain** *eq* **where** *keq*: $(k,eq) \in$ *set ics'* **and** *nstr*: *is-nstrict eq* **by** *force*
from *keq*[*unfolded ics'-def*] *k* **have** $eq = d$ **unfolding** *set-conv-nth* **by** *force*
with *nstr* **have** *is-nstrict d* **by** *auto*
with *ic*[*unfolded n kd*] **have** $c = d$ **by** (*cases d*, *auto*)
then show *?thesis* **using** *vd* **by** *auto*
next
case *e*
from *ic e kd* **have** *j*: $j = k$ **by** (*cases d*, *auto*)
from *e*(*2*)[*unfolded eqs j*] **obtain** *eq* **where** *keq*: $(k,eq) \in$ *set ics'* **and** *iseq*: *is-equality eq* **by** *force*
from *keq*[*unfolded ics'-def*] *k* **have** $eq = d$ **unfolding** *set-conv-nth* **by** *force*
with *iseq* **have** *eq*: *is-equality d* **by** *auto*
with *ic e kd* **have** $c =$ *make-ineq d* $\vee$ $c =$ *make-flipped-ineq d* **by** (*cases d*, *auto*)
then show *?thesis* **using** *vd eq* **by** (*cases d*, *auto*)
next
case *s*
from *ic*[*unfolded s kd*] **have** *j*: $j = k$ **by** (*cases d*, *auto*)
from *s*(*2*)[*unfolded sineqs j*] **obtain** *eq* **where** *keq*: $(k,eq) \in$ *set ics'* **and** *str*: *is-strict eq* **by** *force*
from *keq*[*unfolded ics'-def*] *k* **have** $eq = d$ **unfolding** *set-conv-nth* **by** *force*
with *str* **have** *is-strict d* **by** *auto*
with *ic*[*unfolded s kd*] **have** $c = d$ **by** (*cases d*, *auto*)
then show *?thesis* **using** *vd* **by** *auto*
qed
}
thus $v \models_{cs}$ *cs-of ineqs sineqs eqs* **by** *auto*
qed
qed

definition *init-eq-finder-rat* :: (*eqd-index simplex-state* × *nat list* × *nat list* × *nat list*) *option* **where**
init-eq-finder-rat = (*case init-constraints cs of* (*ics*, *ineqs*, *sineqs*, *eqs*)
⇒ *let s0* = *init-simplex ics*
in (*case assert-all-simplex* (*index-of ineqs sineqs eqs*) *s0*
of Unsat - ⇒ *None*
| *Inr s1* ⇒ (*case check-simplex s1*
of (-, *Some* -) ⇒ *None*
| (*s2*, *None*) ⇒ *Some* (*s2*, *ineqs*, *sineqs*, *eqs*))))

partial-function (*tailrec*) *eq-finder-main-rat* :: *eqd-index simplex-state* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* × *nat list* × (*var* ⇒ *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 ⇒ *Inl* (*s*, *C*)
| *Inr s1* ⇒ (*case check-simplex s1 of*
(*s2*, *None*) ⇒ *Inr* (*solution-simplex s2*)
| (*s2*, *Some C*) ⇒ *Inl* (*backtrack-simplex cp s2*, *C*)))
in case res-strict of
Inr sol ⇒ (*ineq*, *eq*, *sol*) — if indeed all equalities are strictly sat, then no further equality is implied
| *Inl* (*s2*, *C*) ⇒ *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 interface 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* ∉ *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* × (*var* ⇒ *rat*)) *option* **where**
eq-finder-rat = (*case init-eq-finder-rat of None* ⇒ *None*
| *Some* (*s*, *ineqs*, *sineqs*, *eqs*) ⇒ *Some* (
case eq-finder-main-rat s ineqs eqs of (*ineq*, *eq*, *sol*)
⇒ (*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* $\Longrightarrow$ $\nexists$ *v*. *v* $\models_{cs}$ *set cs*
init-eq-finder-rat = *Some* (*s*, *ineq*, *sineq*, *eq*) $\Longrightarrow$
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 (*atomize*(*full*), *goal-cases*)
case *1*
define *s0* **where** *s0* = *init-simplex ics*
define *I* **where** *I* = *index-of ineqs sineqs eqs*
note *init* = *init-eq-finder-rat-def*[*unfolded init-cs split Let-def*, *folded s0-def I-def*]
note *init-cs* = *init-constraints*[*OF init-cs*, *unfolded cs-of-def*, *folded I-def*]
from *init-simplex*[*of ics*, *folded s0-def*]
have *s0*: *invariant-simplex ics* {} *s0* **by** (*rule checked-invariant-simplex*)
show *?case*
proof (*cases assert-all-simplex I s0*)
case *Inl*
from *assert-all-simplex-plain-unsat*[*OF s0 Inl*]
have $\nexists$ *v*. (*set I*,*v*) $\models_{ics}$ *set ics* **by** *auto*
hence $\nexists$ *v*. *v* $\models_{cs}$ *set cs* **using** *init-cs*(*1*) **by** *auto*
with *Inl init* **show** *?thesis* **by** *auto*
next
case (*Inr s1*)
obtain *s2 res* **where** *ch*: *check-simplex s1* = (*s2*, *res*) **by** *force*
note *init* = *init*[*unfolded Inr ch split sum.simps*]
from *assert-all-simplex-ok*[*OF s0 Inr*]
have *s1*: *invariant-simplex ics* (*set I*) *s1* **by** *auto*
show *?thesis*
proof (*cases res*)
case *Some*
note *ch* = *ch*[*unfolded Some*]
from *check-simplex-plain-unsat*[*OF s1 ch*] *init-cs*(*1*)
Some ch init
show *?thesis* **by** *auto*
next
case *None*
note *ch* = *ch*[*unfolded None*]
note *init* = *init*[*unfolded None option.simps*]
from *check-simplex-ok*[*OF s1 ch*]
have *s2*: *checked-simplex ics* (*set I*) *s2* **.**
from *init s2 init-cs*(*1*,*8*,*9*) **show** *?thesis* **unfolding** *I-def equiv-to-cs-def* **by**

fastforce
    qed
  qed
qed

lemma *eq-finder-main-rat*: **fixes** *Ineq Eq*
  assumes *checked-simplex ics* (*set* (*index-of ineqs sineqs eq*)) *s*
  and *set ineq* ⊆ *set ineqs*
  and *set eqs* ⊆ *set eq* ∧ *set eq* ∪ *set ineq* = *set eqs* ∪ *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* ⊆ *set ineqs set eqs* ⊆ *set Eq set Ineq* ∪ *set Eq* = *set eqs* ∪ *set ineqs*

  and *equiv-to-cs Eq*
  and *strict-ineq-sat Ineq Eq v-sol*
  and *distinct* (*Ineq* @ *Eq*)
proof (*atomize*(*full*), *goal-cases*)
  case *1*
  show *?case* **using** *assms*
  proof (*induction ineq arbitrary*: *s eq rule*: *length-induct*)
    case (*1 ineq s eq*)
    define *I* **where** *I* = *set* (*index-of ineqs sineqs eq*)
    note *s* = *1.prems*(*1*)[*folded I-def*]
    note *ineq* = *1.prems*(*2*)
    note *eq* = *1.prems*(*3*)
    note *res* = *1.prems*(*4*)[*unfolded eq-finder-main-rat.simps*[*of - ineq*]]
    note *equiv* = *1.prems*(*5*)
    note *dist* = *1.prems*(*6*)
    note *IH* = *1.IH*[*rule-format*]
    from *s* **have** *inv*: *invariant-simplex ics I s* **by** (*rule checked-invariant-simplex*)
    note *sol* = *solution-simplex*[*OF s refl*]
    show *?case*
    proof (*cases ineq* = [])
      case *True*
      with *res* **have** *Ineq* = [] *Eq* = *eq v-sol* = *solution-simplex s* **by** *auto*
      with *True* **have** *strict-ineq-sat Ineq Eq v-sol* = ((*I*, *solution-simplex s*) $\models_{ics}$ *set ics*)
        unfolding *strict-ineq-sat-def* **by** (*auto simp*: *I-def*)
      with *sol* **have** *strict-ineq-sat Ineq Eq v-sol* **by** *auto*
     with *True res eq ineq equiv sol dist* **show** *?thesis* **by** (*auto simp*: *equiv-to-cs-def strict-ineq-sat-def*)
    next
      case *False*
      hence *False*: (*ineq* = []) = *False* **by** *auto*
      define *cp* **where** *cp* = *checkpoint-simplex s*
      let *?J* = *I* ∪ *TmpSIneq* ‘ *set ineq*
      let *?ass* = *assert-all-simplex* (*map TmpSIneq ineq*) *s*
     define *inner* **where** *inner* = (*case assert-all-simplex* (*map TmpSIneq ineq*) *s*

of Inl I ⇒ *Inl* (*s*, *I*)
| *Inr s1* ⇒ (*case check-simplex s1 of* (*s2*, *None*) ⇒ *Inr* (*solution-simplex s2*) | (*s2*, *Some I*) ⇒ *Inl* (*backtrack-simplex cp s2*, *I*)))
note *res* = *res*[*unfolded False if-False*, *folded cp-def*, *unfolded Let-def*, *folded inner-def*]
{
fix *s2 C*
assume *inner* = *Inl* (*s2*, *C*)
note *inner* = *this*[*unfolded inner-def sum.simps*]
have *set C* ⊆ *?J* ∧ *minimal-unsat-core* (*set C*) *ics* ∧ *invariant-simplex ics I s2*
proof (*cases ?ass*)
case *unsat*: (*Inl D*)
with *inner* **have** *D* = *C s2* = *s* **by** *auto*
with *assert-all-simplex-unsat*[*OF inv unsat*] *inv* **show** *?thesis* **by** *auto*
next
case *ass*: (*Inr s1*)
note *inner* = *inner*[*unfolded ass sum.simps*]
from *inner* **obtain** *s3* **where** *check*: *check-simplex s1* = (*s3*, *Some C*)
and *s2*: *s2* = *backtrack-simplex cp s3*
by (*cases check-simplex s1*, *auto split*: *option.splits*)
note *s1* = *assert-all-simplex-ok*[*OF inv ass*]
from *check-simplex-unsat*[*OF s1 check*]
have *s3*: *weak-invariant-simplex ics ?J s3* **and** *C*: *set C* ⊆ *?J minimal-unsat-core* (*set C*) *ics* **by** *auto*
from *backtrack-simplex*[*OF s cp-def*[*symmetric*] *s3 s2*[*symmetric*]]
have *s2*: *invariant-simplex ics I s2* **by** *auto*
from *s2 C* **show** *?thesis* **by** *auto*
qed
} **note** *inner-Some* = *this*

show *?thesis*
proof (*cases inner*)
case (*Inr sol*)
note *inner* = *this*[*unfolded inner-def*]
from *inner* **obtain** *s1* **where** *ass*: *?ass* = *Inr s1* **by** (*cases ?ass*, *auto*)
note *inner* = *inner*[*unfolded ass sum.simps*]
from *inner* **obtain** *s2* **where** *check*: *check-simplex s1* = (*s2*, *None*) **by** (*cases check-simplex s1*, *auto split*: *option.splits*)
from *solution-simplex*[*OF check-simplex-ok*[*OF assert-all-simplex-ok*[*OF inv ass*] *check*]]
have (*?J*, *sol*) $\models_{ics}$ *set ics* **using** *inner*[*unfolded check split option.simps*] **by** *auto*
hence *str*: *strict-ineq-sat ineq eq sol* **unfolding** *I-def strict-ineq-sat-def* **by** *auto*
from *res*[*unfolded Inr*] **have** *id*: *Ineq* = *ineq Eq* = *eq v-sol* = *sol* **by** *auto*
show *?thesis* **unfolding** *id* **using** *dist eq ineq equiv str* **by** *auto*
next
case (*Inl pair*)

then obtain *s2 C* **where** *inner*: *inner* = *Inl* (*s2*, *C*) **by** (*cases pair*, *auto*)
from *inner-Some*[*OF this*]
have *C*: *set C* $\subseteq$ *I* $\cup$ *TmpSIneq* ' *set ineq*
and *unsat*: *minimal-unsat-core* (*set C*) *ics*
and *s2*: *invariant-simplex ics I s2*
by *auto*
define *eq'* **where** *eq'* = *remdups* [*i*. *TmpSIneq i* <− *C*]
have *ran*: *range TmpSIneq* $\cap$ *I* = {} **unfolding** *I-def index-of-def* **by** *auto*
{
assume *eq'* = []
hence *CI*: *set C* $\subseteq$ *I* **using** *C ran eq'-def* **by** *force*
from *unsat* **have** $\nexists$ *v*. (*set C*, *v*) $\models_{ics}$ *set ics* **unfolding** *minimal-unsat-core-def* **by** *auto*
with *indexed-sat-mono*[*OF sol CI*] **have** *False* **by** *auto*
}
hence *eq'*: *eq'* $\neq$ [] **by** *auto*
let *?eq* = *eq'* @ *eq*
define *s3* **where** *s3* = *sum.projr* (*assert-all-simplex* (*map FIneq eq'*) *s2*)
define *s4* **where** *s4* = *fst* (*check-simplex s3*)
define *ineq'* **where** *ineq'* = *filter* (λ*i*. *i* $\notin$ *set eq'*) *ineq*
have *eq'-ineq*: *set eq'* $\subseteq$ *set ineq* **using** *C ran* **unfolding** *eq'-def* **by** *auto*
have *eq-new*: *set eqs* $\subseteq$ *set ?eq* $\wedge$ *set ?eq* $\cup$ *set ineq'* = *set eqs* $\cup$ *set ineqs* **using** *eq'-ineq ineq eq*
by (*auto simp*: *ineq'-def*)
have *dist*: *distinct* (*ineq'* @ *eq'* @ *eq*) **using** *dist* **unfolding** *ineq'-def* **using** *eq'-ineq*
unfolding *eq'-def* **by** *auto*
have *ineq-new*: *set ineq'* $\subseteq$ *set ineqs* **using** *ineq* **unfolding** *ineq'-def* **by** *auto*
from *eq' eq'-ineq* **have** *len*: *length ineq'* < *length ineq* **unfolding** *ineq'-def*
by (*metis empty-filter-conv filter-True length-filter-less subsetD*)
note *res* = *res*[*unfolded inner sum.simps split*, *folded eq'-def*, *folded s3-def*, *folded ineq'-def s4-def*]
show *?thesis*
proof (*rule IH*[*OF len - ineq-new eq-new res - dist*])
define *I'* **where** *I'* = *index-of ineqs sineqs ?eq*
have *II'*: *set I'* = *set* (*map FIneq eq'*) $\cup$ *I* **unfolding** *I'-def I-def index-of-def* **using** *ineq eq'-ineq* **by** *auto*
show *equiv-new*: *equiv-to-cs ?eq*
proof −
define *c-of* **where** *c-of I* = *Simplex.restrict-to I* (*set ics*) **for** *I*
have *?thesis* $\longleftrightarrow$ ($\forall$ *v*. (*I*, *v*) $\models_{ics}$ *set ics* $\longleftrightarrow$ (*FIneq* ' *set eq'* $\cup$ *I*, *v*) $\models_{ics}$ *set ics*)
unfolding *equiv-to-cs-def* **using** *equiv*[*unfolded equiv-to-cs-def*]
unfolding *I'-def*[*symmetric*] *I-def*[*symmetric*] *II'* **by** *auto*
also have ... $\longleftrightarrow$ ($\forall$ *v*. *v* $\models_{cs}$ *c-of I* $\longrightarrow$ *v* $\models_{cs}$ *c-of* (*FIneq* ' *set eq'*))
unfolding *c-of-def* **by** *auto*
also have ...
proof (*intro allI impI*)

fix *v*
assume *v*: $v \models_{cs}$ *c-of I*
let *?Ineq* = *Equality-Detection-Impl.Ineq* ‘ *set ineq*
let *?SIneq* = *Equality-Detection-Impl.TmpSIneq* ‘ *set ineq*
from *init-constraints*[*OF init-cs*]
have *dist*: *distinct* (*map fst ics*) **unfolding** *distinct-indices-def* **by** *auto*
{
fix *c i*
assume *c*: $c \in$ *c-of* {*i*}
have *c-of* {*i*} = {*c*}
proof −
{
fix *d*
assume $d \in$ *c-of* {*i*}
from *this*[*unfolded c-of-def*]
have *d*: (*i*, *d*) $\in$ *set ics* **by** *force*
from *c*[*unfolded c-of-def*]
have *c*: (*i*, *c*) $\in$ *set ics* **by** *force*
from *c d dist* **have** *c* = *d* **by** (*metis eq-key-imp-eq-value*)
}
with *c* **show** *?thesis* **by** *blast*
qed
} **note** *c-of-inj* = *this*

let *?n* = *length cs*
{
note *init-cs'* = *init-cs*[*unfolded init-constraints-def Let-def*]
fix *i*
assume $i \in$ *set ineq*
with *ineq* **have** $i \in$ *set ineqs* **by** *auto*
with *init-cs'*
have $i \in$ *set* (*map fst* (*filter* (*is-nstrict* $\circ$ *snd*) (*zip* [*0..<length cs*] *cs*))) **by** *auto*
hence *i-n*: *i* < *?n* **and** *nstr*: *is-nstrict* (*cs* ! *i*) **by** (*auto simp*: *set-zip*)
hence (*i*, *cs* ! *i*) $\in$ *set* (*zip* [*0..<?n*] *cs*) **by** (*force simp*: *set-zip*)
with *init-cs'* **have** *set* (*index-constraint* (*i*, *cs* ! *i*)) $\subseteq$ *set ics* **by** *force*
hence
cs ! *i* $\in$ *c-of* {*Equality-Detection-Impl.Ineq i*}
make-strict (*cs* ! *i*) $\in$ *c-of* {*TmpSIneq i*}
make-flipped-ineq (*cs* ! *i*) $\in$ *c-of* {*FIneq i*}
using *nstr* **unfolding** *c-of-def* **by** (*cases cs* ! *i*; *force*)+
with *c-of-inj*
have *c-of* {*Equality-Detection-Impl.Ineq i*} = {*cs* ! *i*}
c-of {*TmpSIneq i*} = {*make-strict* (*cs* ! *i*)}
c-of {*FIneq i*} = {*make-flipped-ineq* (*cs* ! *i*)}
by *auto*
note *nstr this i-n*
} **note** *c-of-ineq* = *this*

have *cIneq*: *c-of ?Ineq* = ((!) *cs*) ‘ *set ineq* **using** *c-of-ineq*(*2*) **unfolding** *c-of-def* **by** *blast*
have *cSIneq*: *c-of ?SIneq* = (*make-strict o* (!) *cs*) ‘ *set ineq*
using *c-of-ineq*(*3*) **unfolding** *c-of-def o-def* **by** *blast*

have *I* ∪ *?Ineq* = *I* **using** *ineq* **unfolding** *I-def index-of-def* **by** *auto*
with *v* **have** $v \models_{cs}$ (*c-of I* ∪ *c-of ?Ineq*) **unfolding** *c-of-def* **by** *auto*
hence *v*: $v \models_{cs}$ (*c-of I* ∪ ((!) *cs*) ‘ *set ineq*) **unfolding** *cIneq* **by** *auto*
have *Ball* (*snd* ‘ *set ics*) *is-no-equality*
using *init-cs*[*unfolded init-constraints-def Let-def*]
apply *clarsimp*
subgoal for *i c j d* **by** (*cases d*, *auto*)
done
hence *no-eq-c*: *Ball* (*c-of I*) *is-no-equality* **for** *I* **unfolding** *c-of-def* **by** *auto*
have *no-eq-ineq*: *i* ∈ *set ineq* ⟹ *is-no-equality* (*cs* ! *i*) **for** *i* **using** *c-of-ineq*(*1*)[*of i*] **by** (*cases cs* ! *i*, *auto*)
define *CI* **where** *CI* = *le-of-constraint* ‘ (*c-of I*)
from *v* **have** *v*: ∀ *c* ∈ *CI* ∪ *le-of-constraint* ‘ ((!) *cs* ‘ *set ineq*). ($v \models_{le}$ *c*)
unfolding *CI-def*
by (*subst* (*asm*) *le-of-constraints*, *insert no-eq-ineq no-eq-c*, *auto*)
define *p* **where** *p* = (λ *i*. *poly-of-constraint* (*cs* ! *i*))
define *co* **where** *co* = (λ *i*. *const-of-constraint* (*cs* ! *i*))
have *nstri*: *Ball* ((!) *cs* ‘ *set ineq*) *is-nstrict* **using** *c-of-ineq*(*1*) **by** *auto*
have *lecs-ineq*: *set ine* ⊆ *set ineq* ⟹ *le-of-constraint*‘ ((!) *cs* ‘ *set ine*) = (λ*i*. *Le-Constraint Leq-Rel* (*p i*) (*co i*)) ‘ *set ine* **for** *ine*
by (*subst poly-const-repr-set*, *insert nstri*, *auto simp*: *p-def co-def*)
from *v lecs-ineq*[*OF subset-refl*]
have *v*: ∀ *c* ∈ *CI* ∪ (λ*i*. *Le-Constraint Leq-Rel* (*p i*) (*co i*)) ‘ *set ineq*. ($v \models_{le}$ *c*) **by** *auto*
have *finCI*: *finite CI* **unfolding** *CI-def c-of-def* **by** *auto*
note *main-step* = *equality-detection-rat*[*OF finCI finite-set - - - v*]

let *?C* = *le-of-constraint* ‘ (*c-of* (*set C*))
from *C* **have** *c-of* (*set C*) ⊆ *c-of I* ∪ *c-of ?SIneq* **unfolding** *c-of-def* **by** *auto*
hence *c-of* (*set C*) ⊆ *c-of I* ∪ (*make-strict o* (!) *cs*) ‘ *set ineq* **unfolding** *cSIneq* **.**
hence *?C* ⊆ *CI* ∪ *le-of-constraint* ‘ ((*make-strict o* (!) *cs*) ‘ *set ineq*)
unfolding *CI-def* **by** *auto*
also have *le-of-constraint*‘ ((*make-strict o* (!) *cs*) ‘ *set ineq*) = (λ*i*. *Le-Constraint Lt-Rel* (*p i*) (*co i*)) ‘ *set ineq*
unfolding *o-def* **unfolding** *p-def co-def*
using *poly-const-repr-set*(*2*)[*OF nstri*, *unfolded image-comp o-def*] **by** *auto*
finally have *?C* ⊆ *CI* ∪ (λ*i*. *Le-Constraint Lt-Rel* (*p i*) (*co i*)) ‘ *set ineq* **by** *auto*

note *main-step* = *main-step*[*OF this*]

from *unsat*[*unfolded minimal-unsat-core-def*]
have $\nexists$ *v*. (*set C*, *v*) $\models_{ics}$ *set ics* **by** *auto*
hence $\nexists$ *v*. *v* $\models_{cs}$ *c-of* (*set C*) **unfolding** *c-of-def* **by** *auto*
hence $\nexists$ *v*. $\forall$ *c*$\in$*le-of-constraint* ' (*c-of* (*set C*)). *v* $\models_{le}$ *c*
by (*subst* (*asm*) *le-of-constraints*[*OF no-eq-c*], *auto*)

note *main-step* = *main-step*[*OF this*]

{
fix *D*
assume *D* $\subset$ *le-of-constraint* ' (*c-of* (*set C*))
hence $\exists$ *CS*. *le-of-constraint* ' *CS* = *D* $\wedge$ *CS* $\subset$ *c-of* (*set C*)
by (*metis subset-image-iff subset-not-subset-eq*)
then obtain *CS* **where** *D*: *D* = *le-of-constraint* ' *CS* **and** *sub*: *CS* $\subset$ *c-of* (*set C*) **by** *auto*
define *c-fun* **where** *c-fun i* = (*THE x*. *x* $\in$ *c-of* {*i*}) **for** *i*
{
fix *C'*
assume *C'*: *C'* $\subseteq$ *set C*
{
fix *i*
assume *i* $\in$ *C'*
with *C' C* **have** *i* $\in$ *I* $\cup$ *TmpSIneq* ' *set ineq* **by** *auto*
from *this*[*unfolded I-def index-of-def*] *ineq eq*
have *i* $\in$ *set* (*map SIneq sineqs* @ *map Equality-Detection-Impl.Ineq eqs* @
map FIneq eqs @ *map Equality-Detection-Impl.Ineq ineqs* @ *map FIneq ineqs* @ *map TmpSIneq ineqs*) (**is** - $\in$ *?S*)
by *auto*
also have *?S* $\subseteq$ *fst* ' *set ics* **using** *init-constraints*(*3*)[*OF init-cs*] **by** *auto*
finally have *i* $\in$ *fst* ' *set ics* **by** *auto*
then obtain *c* **where** (*i*,*c*) $\in$ *set ics* **by** *force*
hence *c* $\in$ *c-of* {*i*} **unfolding** *c-of-def* **by** *force*
from *c-of-inj*[*OF this*] **have** *c*: *c-of* {*i*} = {*c*} **by** *auto*
hence *c-fun i* = *c* **unfolding** *c-fun-def* **by** *auto*
with *c* **have** *c-of* {*i*} = {*c-fun i*} **by** *auto*
}
hence *c-of C'* = *c-fun* ' *C'* **unfolding** *c-of-def* **by** *blast*
} **note** *to-c-fun* = *this*
from *sub*[*unfolded to-c-fun*[*OF subset-refl*]]
have *CS* $\subset$ *c-fun* ' *set C* **by** *auto*
hence $\exists$ *C'*. *C'* $\subset$ *set C* $\wedge$ *CS* = *c-fun* ' *C'*
by (*metis subset-image-iff subset-not-subset-eq*)
then obtain *C'* **where** *sub*: *C'* $\subset$ *set C* **and** *CS*: *CS* = *c-fun* ' *C'* **by** *auto*
from *CS to-c-fun*[*of C'*] *sub* **have** *CS*: *CS* = *c-of C'* **by** *auto*

from *unsat*[*unfolded minimal-unsat-core-def*] *dist sub*
have $\exists\, v.$ (C', v) $\models_{ics}$ *set ics*
unfolding *distinct-indices-def* **by** *auto*
hence $\exists\, v.\ v \models_{cs}$ *CS* **unfolding** *CS c-of-def* **by** *auto*
hence $\exists\, v.\ \forall\, c{\in}D.\ v \models_{le}$ *c* **unfolding** *D*
by (*subst* (*asm*) *le-of-constraints*, *unfold CS*, *insert no-eq-c*, *auto*)
}

note *main-step* = *main-step*[*OF this*]

{
fix *i e*
assume *ieq'*: $i \in$ *set eq'* **and** *mem*: (*FIneq i*, *e*) $\in$ *set ics*
from *ieq' eq'-def* **have** *tmp*: *TmpSIneq i* $\in$ *set C* **by** *auto*
have *i*: $i \in$ *set ineq* **using** *ieq' eq'-ineq* **by** *auto*
from *c-of-ineq*(*1*,*3*,*5*)[*OF i*] *tmp*
have $*$: *make-strict* (*cs* ! *i*) $\in$ *c-of* (*set C*) *is-nstrict* (*cs* ! *i*) *i* < *?n*
by (*auto simp*: *c-of-def*)
from $*$(*3*) **have** (*i*, *cs* ! *i*) $\in$ *set* (*zip* [*0*..< *?n*] *cs*) **by** (*force simp*: *set-zip set-conv-nth*)
hence *set* (*index-constraint* (*i*, *cs* ! *i*)) $\subseteq$ *set ics* **using** *init-cs*[*unfolded init-constraints-def Let-def*]
by *force*
hence (*FIneq i*, *make-flipped-ineq* (*cs* ! *i*)) $\in$ *set ics* **using** $*$(*2*) **by** (*cases cs* ! *i*, *auto*)
with *mem dist* **have** *e*: *e* = *make-flipped-ineq* (*cs* ! *i*) **by** (*metis eq-key-imp-eq-value*)
have *le-of-constraint* (*make-strict* (*cs* ! *i*)) = *Le-Constraint Lt-Rel* (*p i*) (*co i*)
by (*subst poly-const-repr*(*2*), *insert* $*$, *auto simp*: *p-def co-def*)
from *this* $*$ **have** *Le-Constraint Lt-Rel* (*p i*) (*co i*) $\in$ *le-of-constraint* ‘ (*c-of* (*set C*))
by *force*
from *main-step*[*OF - i this*]
have *eq*: (*p i*) $\{\!|\ v\ |\!\}$ = *co i* **by** *auto*
have *id*: *le-of-constraint* (*make-flipped-ineq* (*cs* ! *i*)) = *Le-Constraint Leq-Rel* (− *p i*) (− *co i*)
by (*subst poly-const-repr*(*3*), *insert* $*$, *auto simp*: *p-def co-def*)
from $*$ **have** *is-no-equality* (*make-flipped-ineq* (*cs* ! *i*)) **by** (*cases cs* ! *i*, *auto*)
from *le-of-constraint*[*OF this*, *of v*]
have $v \models_c e$ **using** *e id eq* **by** (*simp add*: *valuate-uminus*)
}
thus $v \models_{cs}$ *c-of* (*FIneq* ‘ *set eq'*) **unfolding** *c-of-def* **by** *auto*
qed
finally show *?thesis* **by** *simp*
qed
from *equiv equiv-new sol*
have *sol*: (*set I'*, *solution-simplex s*) $\models_{ics}$ *set ics* **unfolding** *equiv-to-cs-def*

index-of-def I-def I′-def **by** *auto*
have *II′*: *set I′* = *set* (*map FIneq eq′*) ∪ *I* **unfolding** *I′-def I-def index-of-def* **using** *eq′-ineq ineq* **by** *auto*
let *?ass* = *assert-all-simplex* (*map FIneq eq′*) *s2*
{
fix *K*
assume *?ass* = *Unsat K*
from *assert-all-simplex-plain-unsat*[*OF s2 this*, *folded II′*] *sol* **have** *False* **by** *auto*
}
hence *ass*: *?ass* = *Inr s3* **unfolding** *s3-def* **by** (*cases ?ass*, *auto*)
from *assert-all-simplex-ok*[*OF s2 ass*]
have *s3*: *invariant-simplex ics* (*set I′*) *s3* **unfolding** *II′* **by** (*simp add*: *ac-simps*)
from *s4-def*[*unfolded ass*, *simplified*] **obtain** *c* **where**
check-simplex s3 = (*s4*, *c*) **by** (*cases check-simplex s3*, *auto*)
with *check-simplex-plain-unsat*[*OF s3*] *sol*
have *check-simplex s3* = (*s4*, *None*) **by** (*cases c*, *auto*)
from *check-simplex-ok*[*OF s3 this*]
show *checked-simplex ics* (*set* (*index-of ineqs sineqs* (*eq′* @ *eq*))) *s4* **unfolding** *I′-def* **.**
qed
qed
qed
qed
qed

lemma *eq-finder-rat-in-ctxt*: *eq-finder-rat* = *None* ⟹ ∄ *v*. $v \models_{cs}$ *set cs*
eq-finder-rat = *Some* (*eq-idx*, *v-sol*) ⟹ {*i* . *i* < *length cs* ∧ *is-equality* (*cs* ! *i*)} ⊆ *set eq-idx* ∧
set eq-idx ⊆ {*0* ..< *length cs*} ∧
distinct eq-idx (**is** - ⟹ *?main1*)
eq-finder-rat = *Some* (*eq-idx*, *v-sol*) ⟹
set feq = *make-equality* ‘ (!) *cs* ‘ *set eq-idx* ⟹
set fineq = (!) *cs* ‘ ({*0* ..< *length cs*} − *set eq-idx*) ⟹
(∀ *v*. $v \models_{cs}$ *set cs* ⟷ $v \models_{cs}$ (*set feq* ∪ *set fineq*)) ∧
Ball (*set feq*) *is-equality* ∧ *Ball* (*set fineq*) *is-no-equality* ∧
(*v-sol* $\models_{cs}$ (*set feq* ∪ *make-strict* ‘ *set fineq*)) (**is** - ⟹ - ⟹ - ⟹ *?main2*)
proof −
assume *eq-finder-rat* = *None*
from *this*[*unfolded eq-finder-rat-def*] **have** *init-eq-finder-rat* = *None* **by** (*cases init-eq-finder-rat*, *auto*)
from *init-eq-finder-rat*(*1*)[*OF this*] **show** ∄ *v*. $v \models_{cs}$ *set cs* **.**
next
assume *eq-finder-rat* = *Some* (*eq-idx*, *v-sol*)
note *res* = *this*[*unfolded eq-finder-rat-def*]
then obtain *s ineq sineq eq*
where *init*: *init-eq-finder-rat* = *Some* (*s*, *ineq*, *sineq*, *eq*)
by (*cases init-eq-finder-rat*, *auto*)

from *init-eq-finder-rat*(*2*)[*OF init*] **have** *sineq*: *sineq* = *sineqs*
and *dist*: *distinct* (*ineq* @ *sineq* @ *eq*) **and** *set*: *set* (*ineq* @ *sineq* @ *eq*) = {*0*..<*length cs*} **by** *auto*
note *res* = *res*[*unfolded init option.simps split sineq*]
from *res*
obtain *fi fe* **where** *main*: *eq-finder-main-rat s ineq eq* = (*fi*,*fe*, *v-sol*)
by (*cases eq-finder-main-rat s ineq eq*, *auto*)
note *res* = *res*[*unfolded main split*]
from *res* **have** *eq-idx*: *eq-idx* = *fe*
by *auto*
from *dist* **have** *dist′*: *distinct* (*ineq* @ *eq*) **by** *auto*
from *init-eq-finder-rat*(*2*)[*OF init*]
have *checked-simplex ics* (*set* (*index-of ineqs sineqs eq*)) *s* **and**
∗∗: *set ineq* ⊆ *set ineqs set eqs* ⊆ *set eq* ∧ *set eq* ∪ *set ineq* = *set eqs* ∪ *set ineqs*

equiv-to-cs eq
and ∗∗∗: {*0*..<*length cs*} = *set* (*ineq* @ *sineq* @ *eq*) *distinct* (*ineq* @ *sineq* @ *eq*)
by *auto*
from *eq-finder-main-rat*[*OF this*(*1*,*2*,*3*) *main this*(*4*) *dist′*]
have ∗: *set fi* ⊆ *set ineqs set eqs* ⊆ *set fe set fe* ∪ *set fi* = *set eqs* ∪ *set ineqs*
and *equiv*: *equiv-to-cs fe*
and *sat*: *strict-ineq-sat fi fe v-sol*
and *dist′′*: *distinct* (*fi* @ *fe*) **by** *auto*

note *init* = *init-cs*[*unfolded init-constraints-def Let-def*]
note *init′* = *init-constraints*[*OF init-cs*]
note *eqs* = *init′*(*4*)

show *?main1*
proof (*intro conjI*)
show *distinct eq-idx* **unfolding** *eq-idx* **using** *dist′′* **by** *auto*
show {*i* . *i* < *length cs* ∧ *is-equality* (*cs* ! *i*)} ⊆ *set eq-idx*
unfolding *eq-idx* **using** *set* ∗ ∗∗ *eqs* **by** *auto*
show *set eq-idx* ⊆ {*0*..<*length cs*} **unfolding** *eq-idx* **using** *set* ∗ ∗∗ **by** *auto*
qed

assume *feq*: *set feq* = *make-equality* ‘ (!) *cs* ‘ *set eq-idx*
assume *fineq*: *set fineq* = (!) *cs* ‘ ({*0* ..< *length cs*} − *set eq-idx*)
from *feq eq-idx*
have *feq*: *set feq* = *set* (*map* (λ*i*. *make-equality* (*cs* ! *i*)) *fe*) **by** *auto*
have *fineq*: *set fineq* = *set* (*map* ((!) *cs*) (*sineqs* @ *fi*))
unfolding *set-map* ∗∗∗ **using** ∗∗∗(*2*) **unfolding** *sineq eq-idx fineq*
apply (*intro image-cong*[*OF* - *refl*])
unfolding ∗∗∗ *sineq* **using** ∗ ∗∗(*1*−*2*) *dist′′* **by** *auto*
note *ineqs* = *init′*(*5*)
note *sineqs* = *init′*(*6*)
note *ics* = *init′*(*7*)
from ∗(*3*) **have** *fe*: *i* ∈ *set fe* ⟹ *is-equality* (*cs* ! *i*) ∨ *is-nstrict* (*cs* ! *i*) **for** *i*

unfolding *eqs ineqs* **by** *auto*
let *?n* = *length cs*
show *?main2*
proof (*intro conjI ballI allI*)
define *c-of* **where** *c-of I* = *Simplex.restrict-to I* (*set ics*) **for** *I*
have [*simp*]: *c-of* ($I \cup J$) = *c-of I* $\cup$ *c-of J* **for** *I J* **unfolding** *c-of-def* **by** *auto*
{
fix *v*
have *cs*: $v \models_{cs}$ *set cs* = $v \models_{cs}$ *c-of* (*set* (*index-of ineqs sineqs fe*)) (**is** - = *?cond*)
using *equiv*[*unfolded equiv-to-cs-def*] **unfolding** *c-of-def* **by** *auto*
have *?cond* $\longleftrightarrow$ $v \models_{cs}$ *c-of* (*SIneq* ‘ *set sineqs*)
$\wedge$ ($v \models_{cs}$ *c-of* (*Ineq* ‘ *set fe*)
$\wedge$ $v \models_{cs}$ *c-of* (*FIneq* ‘ *set fe*))
$\wedge$ $v \models_{cs}$ *c-of* (*Ineq* ‘ *set ineqs*) **unfolding** *index-of-def*
by *auto*
also have *c-of* (*SIneq* ‘ *set sineqs*) = ((!) *cs*) ‘ *set sineqs*
unfolding *c-of-def ics*
unfolding *sineqs* **by** *force*
also have *c-of* (*Ineq* ‘ *set ineqs*) = ((!) *cs*) ‘ *set ineqs*
unfolding *c-of-def ics*
unfolding *ineqs eqs*
by (*auto simp*: *is-nstrict-iff*) *force*
also have *c-of* (*FIneq* ‘ *set fe*) = (λ *i*. *make-flipped-ineq* (*cs* ! *i*)) ‘ *set fe* (**is** *?l* = *?r*)
proof
show *?l* $\subseteq$ *?r*
unfolding *c-of-def ics* **using** *fe* $*$(*3*)
unfolding *ineqs eqs* **by** *auto*
show *?r* $\subseteq$ *?l*
proof
fix *c*
assume *c* $\in$ *?r*
then obtain *i* **where** *i*: *i* $\in$ *set fe* **and** *c*: *c* = *make-flipped-ineq* (*cs* ! *i*)
by *auto*
from $*$ *i* **have** *i'*: *i* $\in$ *set eqs* $\cup$ *set ineqs* **by** *auto*
have (*FIneq i*, *c*) $\in$ *set ics* $\cap$ {*FIneq i*} $\times$ *UNIV*
unfolding *c ics* **using** *i'* **by** *auto*
hence *c* $\in$ *c-of* {*FIneq i*} **unfolding** *c-of-def* **by** *force*
with *i* **show** *c* $\in$ *?l* **unfolding** *c-of-def* **by** *auto*
qed
qed
also have *c-of* (*Ineq* ‘ *set fe*) = (λ *i*. *make-ineq* (*cs* ! *i*)) ‘ *set fe* (**is** *?l* = *?r*)
proof
{
fix *i*
have *i* $\in$ *set fe* $\Longrightarrow$ *is-nstrict* (*cs* ! *i*) $\Longrightarrow$ *cs* ! *i* $\in$ (λ*i*. *make-ineq* (*cs* ! *i*)) ‘ *set fe*

by (*cases cs* ! *i*; *force*)
}
thus *?l* ⊆ *?r*
unfolding *c-of-def ics* **using** *fe* ∗(*3*)
unfolding *ineqs eqs* **by** *auto*
show *?r* ⊆ *?l*
proof
fix *c*
assume *c* ∈ *?r*
then obtain *i* **where** *i*: *i* ∈ *set fe* **and** *c*: *c* = *make-ineq* (*cs* ! *i*)
by *auto*
from ∗ *i* **have** *i'*: *i* ∈ *set eqs* ∪ *set ineqs* **by** *auto*
from *fe*[*OF i*]
have (*Ineq i*, *c*) ∈ *set ics* ∩ {*Ineq i*} × *UNIV*
proof
assume *is-equality* (*cs* ! *i*)
with *i'* **have** *i* ∈ *set eqs* **unfolding** *ineqs* **by** (*cases cs* ! *i*, *auto*)
thus *?thesis*
unfolding *c ics* **using** *i'* **by** (*cases cs* ! *i*; *force*)
next
assume *stri*: *is-nstrict* (*cs* ! *i*)
with *i'* **have** *i'*: *i* ∈ *set ineqs* **unfolding** *eqs* **by** (*cases cs* ! *i*, *auto*)
from *stri* **have** *c*: *c* = *cs* ! *i* **unfolding** *c* **by** (*cases cs* ! *i*, *auto*)
thus *?thesis*
unfolding *c ics* **using** *i'* **by** (*cases cs* ! *i*; *force*)
qed
hence *c* ∈ *c-of* {*Ineq i*} **unfolding** *c-of-def* **by** *force*
with *i* **show** *c* ∈ *?l* **unfolding** *c-of-def* **by** *auto*
qed
qed
also have $v \models_{cs}$ ((λ*i*. *make-ineq* (*cs* ! *i*)) ' *set fe*) ∧
$v \models_{cs}$ ((λ*i*. *make-flipped-ineq* (*cs* ! *i*)) ' *set fe*)
⟷ $v \models_{cs}$ ((λ *i*. *make-equality* (*cs* ! *i*)) ' *set fe*) (**is** *?l* = *?r*)
proof −
have *?l* ⟷ (∀ *i* ∈ *set fe*. $v \models_{c}$ *make-ineq* (*cs* ! *i*) ∧ $v \models_{c}$ *make-flipped-ineq* (*cs* ! *i*))
by *auto*
also have ... ⟷ (∀ *i* ∈ *set fe*. $v \models_{c}$ *make-equality* (*cs* ! *i*))
apply (*intro ball-cong*[*OF refl*])
subgoal for *i* **using** *fe*[*of i*]
by (*cases cs* ! *i*, *auto*)
done
also have ... ⟷ *?r* **by** *auto*
finally show *?l* = *?r* **.**
qed
finally have *?cond* ⟷
$v \models_{cs}$ ((!) *cs* ' (*set sineqs* ∪ *set ineqs*) ∪ (λ*i*. *make-equality* (*cs* ! *i*)) ' *set fe*)
by *auto*
also have ... ⟷ $v \models_{cs}$ (*set feq* ∪ *set fineq*) (**is** *?l* = *?r*)

```
      proof
        show ?l ⟹ ?r unfolding feq fineq using ∗ by auto
        assume v: ?r
        show ?l
        proof
          fix c
          assume c: c ∈ (!) cs ‘ (set sineqs ∪ set ineqs) ∪
             (λi. make-equality (cs ! i)) ‘ set fe
          show v ⊨_c c
          proof (cases c ∈ (!) cs ‘ (set sineqs ∪ set fi) ∪
             (λi. make-equality (cs ! i)) ‘ set fe)
            case True
            thus ?thesis using v feq fineq ∗ by auto
          next
            case False
            with c obtain i where i ∈ set ineqs − set fi and c: c = cs ! i by auto
            with ∗ have i: i ∈ set fe by auto
            with v have v ⊨_c make-equality (cs ! i)
              using v feq fineq ∗ by auto
            with fe[OF i] show ?thesis unfolding c by (cases cs ! i, auto)
          qed
        qed
      qed
      finally have main: ?cond ⟷ v ⊨_cs (set feq ∪ set fineq) by auto
      with cs show v ⊨_cs set cs = v ⊨_cs (set feq ∪ set fineq) by auto
      note main
    } note main = this
    fix c
    {
      assume c ∈ set feq
      from this[unfolded feq] obtain i where i: i ∈ set fe
        and c: c = make-equality (cs ! i) by auto
      from i ∗ have i ∈ set eqs ∪ set ineqs by auto
      hence is-equality (cs ! i) ∨ is-nstrict (cs ! i)
        unfolding ineqs eqs by auto
      thus is-equality c unfolding c
        by (cases cs ! i, auto)
    }
    {
      assume c ∈ set fineq
      from this[unfolded fineq] ∗ obtain i where i: i ∈ set sineqs ∪ set ineqs
        and c: c = cs ! i by auto
      hence is-nstrict c ∨ is-strict c unfolding c sineqs ineqs by auto
      thus is-no-equality c by (cases c, auto)
    }
    from sat[unfolded strict-ineq-sat-def]
     have old: v-sol ⊨_cs c-of (set (index-of ineqs sineqs fe)) and new: v-sol ⊨_cs
c-of (TmpSIneq ‘ set fi)
      by (auto simp: c-of-def)
```

```
    have tmp: c-of (TmpSIneq ‘ set fi) = (λ i. make-strict (cs ! i)) ‘ set fi
      apply (rule sym)
      unfolding c-of-def ics using ∗(1) unfolding ineqs
      by force

    fix c
    assume c ∈ set feq ∪ make-strict ‘ set fineq
    thus v-sol ⊨c c
    proof
      assume c ∈ set feq
      thus ?thesis using old[unfolded main] by auto
    next
      assume c ∈ make-strict ‘ set fineq
      from this[unfolded fineq]
      obtain i where i: i ∈ set sineqs ∨ i ∈ set fi
        and c: c = make-strict (cs ! i) by force
      from i show ?thesis
      proof
        assume i ∈ set fi
        with new[unfolded tmp] c show ?thesis by auto
      next
        assume i: i ∈ set sineqs
        hence v: v-sol ⊨c (cs ! i) using old[unfolded main]
          unfolding fineq by auto
        from i[unfolded sineqs] have make-strict (cs ! i) = cs ! i
          by (cases cs ! i, auto)
        with v show ?thesis unfolding c by auto
      qed
    qed
  qed
qed


end
end


lemma eq-finder-rat:
  eq-finder-rat cs = None ⟹ ∄ v. v ⊨cs set cs (is ?p1 ⟹ ?g1)
  eq-finder-rat cs = Some (eq-idx, v-sol) ⟹
    {i . i < length cs ∧ is-equality (cs ! i)} ⊆ set eq-idx ∧
    set eq-idx ⊆ {0 ..< length cs} ∧
    distinct eq-idx (is ?p2 ⟹ ?g2)
  eq-finder-rat cs = Some (eq-idx, v-sol) ⟹
   set eq = make-equality ‘ (!) cs ‘ set eq-idx ⟹
   set ineq = (!) cs ‘ ({0 ..< length cs} − set eq-idx) ⟹
    (∀ v. v ⊨cs set cs ⟷ v ⊨cs (set eq ∪ set ineq)) ∧
    Ball (set eq) is-equality ∧ Ball (set ineq) is-no-equality ∧
    (v-sol ⊨cs (set eq ∪ make-strict ‘ set ineq))
```

(**is** *?p2* ⟹ *?p3* ⟹ *?p4* ⟹ *?g3*)
proof −
obtain *ics ineqs sineqs eqs*
where *init-constraints cs* = (*ics*, *ineqs*, *sineqs*, *eqs*)
by (*cases init-constraints cs*)
from *eq-finder-rat-in-ctxt*[*OF this*]
show *?p1* ⟹ *?g1 ?p2* ⟹ *?g2 ?p2* ⟹ *?p3* ⟹ *?p4* ⟹ *?g3* **by** *auto*
qed

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

end

5.3 Algorithm to Detect Implicit Equalities in 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* :: (*int*,*var*)*lpoly* ⇒ *linear-poly* **where**
[*code del*]: *linear-poly-of-lpoly p* = (*let cxs* = *map* (λ *v*. (*v*, *coeff-l p v*)) (*vars-l-list p*)
in sum-list (*map* (λ (*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* (λ (*x*,*c*). *lp-monom* (*of-int c*) *x*) *cxs*))
unfolding *linear-poly-of-lpoly-def vars-coeffs-impl*(*5*) **..**

lemma *valuate-sum-list*: *valuate* (*sum-list ps*) α = *sum-list* (*map* (λ *p*. *valuate p* α) *ps*)
by (*induct ps*, *auto simp*: *valuate-zero valuate-add*)

lemma *linear-poly-of-lpoly*: *rat-of-int* (*eval-l* α *p*) = *of-int* (*constant-l p*) + *valuate* (*linear-poly-of-lpoly p*) (λ *x*. *of-int* (α *x*))
unfolding *eval-l-def of-int-add*
unfolding *linear-poly-of-lpoly-def Let-def map-map o-def split valuate-sum-list valuate-lp-monom*
unfolding *of-int-mult*[*symmetric*] *of-int-sum*
unfolding *vars-l-list-def*
by (*subst sum-list-distinct-conv-sum-set*, *auto*)

definition *dleq-to-constraint* :: *var dleq* ⇒ *constraint* **where**

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

lemma *dleq-to-constraint*: *satisfies-dleq* α *e* ⟷ *satisfies-constraint* (λ *x*. *rat-of-int* (α *x*)) (*dleq-to-constraint e*)
proof −
  have *satisfies-dleq* α *e* ⟷ *rat-of-int* (*eval-l* α *e*) = *0*
    unfolding *satisfies-dleq-def* **by** *blast*
  also have . . . ⟷ *satisfies-constraint* (λ *x*. *rat-of-int* (α *x*)) (*dleq-to-constraint e*)
    unfolding *linear-poly-of-lpoly*[*of* α *e*] *dleq-to-constraint-def*
    by *auto*
  finally show *?thesis* **.**
qed

definition *dlineq-to-constraint* :: *var dlineq* ⇒ *constraint* **where**
  dlineq-to-constraint p = *LEQ* (*linear-poly-of-lpoly p*) (*of-int* (− *constant-l p*))

lemma *dlineq-to-constraint*: *satisfies-dlineq* α *e* ⟷
  satisfies-constraint (λ *x*. *rat-of-int* (α *x*)) (*dlineq-to-constraint e*)
proof −
  have *satisfies-dlineq* α *e* ⟷ *rat-of-int* (*eval-l* α *e*) ≤ *0*
    unfolding *satisfies-dlineq-def* **by** *simp*
  also have . . . ⟷ *satisfies-constraint* (λ *x*. *rat-of-int* (α *x*)) (*dlineq-to-constraint e*)
    unfolding *linear-poly-of-lpoly*[*of* α *e*] *dlineq-to-constraint-def*
    by *auto*
  finally show *?thesis* **.**
qed

definition *eq-finder-int* :: *var dlineq list* ⇒
    (*var dleq list* × *var dlineq list*) *option* **where**
  [*code del*]: *eq-finder-int ineqs* = (*case*
      eq-finder-rat (*map dlineq-to-constraint ineqs*) *of*
        None ⇒ *None*
       | *Some* (*idx-eq*, -) ⇒ *let I* = *set idx-eq*;
           ics = *zip* [*0*..< *length ineqs*] *ineqs*
         in case List.partition (λ (*i*,*c*). *i* ∈ *I*) *ics*
           of (*eqs2*, *ineqs2*) ⇒ *Some* (*map snd eqs2*, *map snd ineqs2*))

lemma *classify-dlineq-to-constraint*[*simp*]:
  ¬ *is-strict* (*dlineq-to-constraint c*)
  ¬ *is-equality* (*dlineq-to-constraint c*)
  is-nstrict (*dlineq-to-constraint c*)
  by (*auto simp*: *dlineq-to-constraint-def*)

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, [], []))
  unfolding init-constraints-def length-map Let-def
  apply (clarsimp simp flip: set-empty, intro conjI)
  subgoal apply (subst filter-True)
    subgoal by (auto dest!: set-zip-rightD)
    subgoal by auto
    done
  by (auto dest!: set-zip-rightD)

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 ⟹ ∄ α. α ⊨_dio ({}, set ineqs)
    res = Some (eqs, ineqs') ⟹ α ⊨_dio ({}, set ineqs) ⟷ α ⊨_dio (set eqs, set ineqs')
    res = Some (eqs, ineqs') ⟹ ∃ α. α ⊨_cs (make-strict ` dlineq-to-constraint ` set ineqs')
    res = Some (eqs, ineqs') ⟹ length ineqs = length eqs + length ineqs'
proof (atomize(full), goal-cases)
  case 1
  define cs where cs = map dlineq-to-constraint ineqs
  let ?sat = λ α eqs ineqs. Ball (set eqs) (satisfies-dleq α) ∧ Ball (set ineqs) (satisfies-dlineq α)
  note defs = dlineq-to-constraint dleq-to-constraint
  note defs2 = satisfies-dlineq-def satisfies-dleq-def
  note defs3 = dlineq-to-constraint-def dleq-to-constraint-def
  note res = res[unfolded eq-finder-int-def, folded cs-def]
  show ?case
  proof (cases eq-finder-rat cs)
    case None
    with res have res: res = None by auto
    from eq-finder-rat(1)[OF None, unfolded cs-def]
    have ∄ α. ?sat α [] ineqs unfolding defs by auto
    with res show ?thesis by auto
  next
    case (Some pair)
    then obtain eq-idx sol where eq: eq-finder-rat cs = Some (eq-idx, sol) by (cases pair, auto)
    define ics where ics = zip [0 ..< length ineqs] ineqs
    let ?I = set eq-idx
    let ?part = List.partition (λ(i, c). i ∈ ?I) ics
    obtain ineqs2 eqs2 where part: ?part = (eqs2, ineqs2) by force
    let ?ineqs2 = map snd ineqs2
    let ?eqs2 = map snd eqs2
    have ics: ics = map (λ i. (i, ineqs ! i)) [0 ..< length ineqs]
```

unfolding *ics-def* **by** (*intro nth-equalityI*, *auto*)
from *part* **have** *eqs2*: *?eqs2* = *map* ((!) *ineqs*) (*filter* (λ *i*. *i* ∈ *?I*) [*0* ..< *length ineqs*])
unfolding *ics* **by** (*auto simp*: *filter-map o-def*)
from *part* **have** *ineqs2*: *?ineqs2* = *map* ((!) *ineqs*) (*filter* (λ *i*. *i* ∉ *?I*) [*0* ..< *length ineqs*])
unfolding *ics* **by** (*auto simp*: *filter-map o-def*)
note *res* = *res*[*unfolded eq option.simps split Let-def*, *folded ics-def*,
unfolded part split]
from *eq-finder-rat*(*2*)[*OF eq*]
have *eq-finder2*: {*i*. *i* < *length cs* ∧ *is-equality* (*cs* ! *i*)} ⊆ *?I*
?I ⊆ {*0*..<*length cs*}
distinct eq-idx **by** *auto*
have *len*: *length ineqs* = *length cs* **unfolding** *cs-def* **by** *auto*
from *eq-finder2* **have** *filter*: {*x* ∈ *set* [*0*..<*length ineqs*]. *x* ∈ *?I*} = *?I*
unfolding *len* **by** *force*
from *eq-finder2* **have** *filter'*: *set* (*filter* (λ*i*. *i* ∉ *?I*) [*0*..<*length ineqs*]) = {*0* ..< *length cs*} − *?I*
unfolding *len* **by** *force*
have *eqs2'*: *set* (*map dleq-to-constraint ?eqs2*) = *make-equality* ‘ (!) *cs* ‘ *?I*
unfolding *set-map eqs2 set-filter image-comp filter o-def* **using** *eq-finder2*
by (*intro image-cong*[*OF refl*])
(*auto simp*: *cs-def nth-append defs3*)
have *ineqs2'*: *set* (*map dlineq-to-constraint ?ineqs2*) = (!) *cs* ‘ ({*0*..<*length cs*} − *?I*)
unfolding *set-map ineqs2 filter' image-comp o-def*
apply (*intro image-cong*[*OF refl*])
subgoal for *i* **using** *set-mp*[*OF eq-finder2*(*1*), *of i*]
unfolding *defs2* **by** (*auto simp*: *cs-def nth-append defs3*)
done

from *eq-finder-rat*(*3*)[*OF eq eqs2' ineqs2'*] **have**
equiv: ⋀ *v*. *v* $\models_{cs}$ *set cs* = *v* $\models_{cs}$ (*dleq-to-constraint* ‘ *set ?eqs2* ∪ *dlineq-to-constraint* ‘ *set ?ineqs2*)
and *strict*: *sol* $\models_{cs}$ (*set* (*map dleq-to-constraint ?eqs2*) ∪ *make-strict* ‘ *set* (*map dlineq-to-constraint ?ineqs2*))
unfolding *set-map* **by** *metis+*
from *strict* **have** *strict*: *sol* $\models_{cs}$ (*make-strict* ‘ *dlineq-to-constraint* ‘ *set ?ineqs2*) **by** *auto*
{
let *?α* = λ *x* :: *var*. *rat-of-int* (*α x*)
have *?sat α* [] *ineqs* ⟷ *?α* $\models_{cs}$ *set cs* **unfolding** *cs-def*
by (*auto simp*: *defs*)
also have ... ⟷ *?sat α ?eqs2 ?ineqs2* **unfolding** *equiv*
using *defs*[*of α*] **by** *fastforce*
finally have *?sat α* [] *ineqs* ⟷ *?sat α ?eqs2 ?ineqs2* **.**
} **note** *eq* = *this*

have *length ineqs* = *length ics* **unfolding** *ics-def* **by** *auto*

also have ... = *length eqs2* + *length ineqs2* **using** *part*[*simplified*]
by (*smt* (*verit*) *comp-def filter-cong sum-length-filter-compl*)
finally show *?thesis* **using** *eq res strict* **by** *fastforce*
qed
qed

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* ⇒ *var dlineq list* ⇒
(*var dlineq list* × ((*int*,*var*)*assign* ⇒ (*int*,*var*)*assign*)) *option* **where**
dio-elim-equations-and-tighten eqs ineqs = (*case equality-elim-for-inequalities fresh-vars-nat eqs ineqs*
of None ⇒ *None*
| *Some* (*ineqs2*, *adj*) ⇒ *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* ⟹ $\nexists$ α. α $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) ⟹ α $\models_{dio}$ ({}, *set ineqs'*) ⟹ β = *adj* α ⟹ β $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'* , *adj*) ⟹ $\nexists$ α. α $\models_{dio}$ ({}, *set ineqs'*) ⟹ $\nexists$ α . α $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) ⟹ *length ineqs'* ≤ *length ineqs*
proof (*atomize*(*full*), *goal-cases*)
case *1*
note *res* = *res*[*unfolded dio-elim-equations-and-tighten-def*]
show *?case*
proof (*cases equality-elim-for-inequalities fresh-vars-nat eqs ineqs*)
case *None*
from *equality-elim-for-inequalities-nat*(*1*)[*OF None refl*] *None* **show** *?thesis* **using** *res* **by** *auto*

next
case (*Some pair*)
obtain *ineqs2 adj′* **where** *pair*: *pair* = (*ineqs2*, *adj′*) **by** *force*
note *Some* = *Some*[*unfolded pair*]
note *res* = *res*[*unfolded Some option.simps split*]
note *eq-elim* = *equality-elim-for-inequalities-nat*(*2*−)[*OF Some refl*]
show *?thesis*
proof (*cases tighten-ineqs ineqs2*)
case *None*
with *res eq-elim tighten-ineqs*(*1*)[*OF None*] **show** *?thesis* **by** *auto*
next
case (*Some ineqs3*)
with *res eq-elim tighten-ineqs*(*2*)[*OF Some*] **show** *?thesis* **by** *force*
qed
qed
qed

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″*))))
by *pat-completeness auto*

termination
proof (*standard*, *rule wf-measure*[*of length*], *goal-cases*)
case (*1 ineqs pair eqs ineqs′ e eqs′ pair′ ineqs″ adj*)
from *eq-finder-int*(*4*)[*OF 1*(*1*), *folded 1*(*2*), *OF refl*]
dio-elim-equations-and-tighten(*4*)[*OF 1*(*4*), *folded 1*(*5*), *OF refl*]
1(*3*)
show *?case* **by** *auto*
qed

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

lemma *dio-preprocess-main*: **assumes**
res: *dio-preprocess-main ineqs* = *res*
shows *res* = *None* $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ ({}, *set ineqs*)
res = *Some* (*ineqs′*, *adj*) $\Longrightarrow \alpha \models_{dio}$ ({}, *set ineqs′*) $\Longrightarrow$ (*adj* α) $\models_{dio}$ ({}, *set ineqs*)
res = *Some* (*ineqs′* , *adj*) $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ ({}, *set ineqs′*) $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ ({}, *set ineqs*)
res = *Some* (*ineqs′*, *adj*) $\Longrightarrow \exists\ \alpha.\ \alpha \models_{cs}$ (*make-strict* ‘ *dlineq-to-constraint* ‘ *set*

ineqs′)
proof (*atomize*(*full*), *goal-cases*)
case *1*
show *?case* **using** *res*
proof (*induction ineqs arbitrary*: *res ineqs*′ *adj* α *rule*: *dio-preprocess-main.induct*)
case (*1 ineqs res ineqs*′ *adj* α)
note *res* = *dio-preprocess-main.simps*[*of ineqs*, *unfolded 1.prems*]
show *?case*
proof (*cases eq-finder-int ineqs*)
case *None*
from *res*[*unfolded None option.simps*] *eq-finder-int*(*1*)[*OF None*] **show** *?thesis* **by** *auto*
next
case (*Some pair*)
obtain *eqs1 ineqs1* **where** *pair*: *pair* = (*eqs1*, *ineqs1*) **by** *force*
note *Some* = *Some*[*unfolded pair*]
note *res* = *res*[*unfolded Some option.simps split*]
note *eqf* = *eq-finder-int*(*2*,*3*)[*OF Some refl*]
note *IH* = *1.IH*[*OF Some refl*]
show *?thesis*
proof (*cases eqs1*)
case *Nil*
with *res* **have** *res* = *Some* (*ineqs1*, *id*) **by** *auto*
with *res eqf Nil* **show** *?thesis* **by** *auto*
next
case (*Cons e eqs1*′)
note *res* = *res*[*unfolded Cons list.simps*, *folded Cons*]
note *IH* = *IH*[*OF Cons*]
show *?thesis*
proof (*cases dio-elim-equations-and-tighten eqs1 ineqs1*)
case *None*
note *res* = *res*[*unfolded None option.simps*]
from *dio-elim-equations-and-tighten*(*1*)[*OF None*] *res* **show** *?thesis* **using** *eqf* **by** *auto*
next
case (*Some pair2*)
obtain *ineqs2 adj2* **where** *pair2*: *pair2* = (*ineqs2*, *adj2*) **by** *force*
note *Some* = *Some*[*unfolded this*]
note *res* = *res*[*unfolded Some option.simps split*]
note *IH* = *IH*[*OF Some refl refl*]
note *elim* = *dio-elim-equations-and-tighten*(*2*−*3*)[*OF Some refl*]
note *elim* = *elim*(*1*)[*OF - refl*] *elim*(*2*)
show *?thesis*
proof (*cases dio-preprocess-main ineqs2*)
case *None*
with *IH* **have** $\nexists\alpha$. $\forall$ *a*∈*set ineqs2*. *satisfies-dlineq* α *a* **by** *auto*
with *elim res None eqf* **show** *?thesis* **by** *auto*
next
case (*Some pair3*)

obtain *ineqs3 adj3* **where** *pair3*: *pair3* = (*ineqs3*, *adj3*) **by** *force*
note *Some* = *Some*[*unfolded this*]
from *res*[*unfolded Some*]
have *res*: *res* = *Some* (*ineqs3*, *adj2 o adj3*) **by** *auto*
from *IH*[*of ineqs3 adj3*] *Some res IH elim eqf* **show** *?thesis* **by** *auto*
qed
qed
qed
qed
qed
qed

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* (λ *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* $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) $\Longrightarrow$ ($\exists\ \alpha.\ \alpha \models_{dio}$ ({}, *set ineqs'*)) $\longleftrightarrow$ ($\exists\ \alpha.\ \alpha \models_{dio}$ (*set eqs*, *set ineqs*))
res = *Some* (*ineqs'*, *adj*) $\Longrightarrow \alpha \models_{dio}$ ({}, *set ineqs'*) $\Longrightarrow$ (*adj* α) $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) $\Longrightarrow \exists\ \alpha.\ \alpha \models_{cs}$ (*make-strict* ‘ *dlineq-to-constraint* ‘ *set ineqs'*)
proof (*atomize*(*full*), *goal-cases*)
case *1*
note *res* = *res*[*unfolded dio-preprocess-def*]
show *?case*
proof (*cases dio-elim-equations-and-tighten eqs ineqs*)
case *None*
with *dio-elim-equations-and-tighten*(*1*)[*OF None*] *res* **show** *?thesis* **by** *auto*
next
case (*Some pair*)
obtain *ineqs1 adj1* **where** *pair* = (*ineqs1*, *adj1*) **by** *force*
note *Some* = *Some*[*unfolded this*]

note *res* = *res*[*unfolded Some option.simps split*]
note *elim* = *dio-elim-equations-and-tighten*(*2*−*3*)[*OF Some refl*]
note *elim* = *elim*(*1*)[*OF - refl*] *elim*(*2*)
show *?thesis*
proof (*cases dio-preprocess-main ineqs1*)
  case *None*
  with *dio-preprocess-main*(*1*)[*OF None*] *res elim* **show** *?thesis* **by** *auto*
next
  case (*Some pair2*)
  obtain *ineqs2 adj2* **where** *pair2* = (*ineqs2*, *adj2*) **by** *force*
  note *Some* = *Some*[*unfolded this*]
  from *res*[*unfolded Some*]
  have *res*: *res* = *Some* (*ineqs2*, *adj1* ∘ *adj2*) **by** *auto*
  from *dio-preprocess-main*(*2*−*4*)[*OF Some refl*] *elim res*
  show *?thesis* **by** *fastforce*
qed
qed
qed

end

7 Examples

theory *Dio-Preprocessing-Examples*
imports
Dio-Preprocessor
begin

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

definition *example-3-x-min-y* :: (*int*,*var*)*lpoly list* **where**
example-3-x-min-y = (*let x* = *var-l 1*; *y* = *var-l 2 in*
[*const-l 1* − *smult-l 3 x* + *smult-l 3 y*,
smult-l 3 x − *smult-l 3 y* − *const-l 2*])

Preprocessing can detect unsat

lemma *case dio-preprocess* [] *example-3-x-min-y of None* ⇒ *True* | *Some -* ⇒ *False*

by *eval*

Griggio, example 1, unsat detection by preprocessing

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

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

by *eval*

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 ⇒ *True* | *Some -* ⇒ *False*
by *eval*

Termination proof of binary logarithm program $n := 0$; *while* $(x > 1)$ $\{x := x$ *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 satisability 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.