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* ⇒ *'a*. *finite* {*v*. *c v* ≠ *0*}}
  ⟨*proof*⟩

setup-lifting *type-definition-lpoly*

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

lift-definition *uminus-lpoly* :: (*'a*, *'b*) *lpoly* ⇒ (*'a*, *'b*) *lpoly* **is** λ *c x*. − *c x* ⟨*proof*⟩

lift-definition *minus-lpoly* :: (*'a*, *'b*) *lpoly* ⇒ (*'a*, *'b*) *lpoly* ⇒ (*'a*, *'b*) *lpoly* **is** λ *c1 c2 x*. *c1 x* − *c2 x*
⟨*proof*⟩

lift-definition *plus-lpoly* :: (*'a*, *'b*) *lpoly* ⇒ (*'a*, *'b*) *lpoly* ⇒ (*'a*, *'b*) *lpoly* **is** λ *c1 c2 x*. *c1 x* + *c2 x*
⟨*proof*⟩

lift-definition *zero-lpoly* :: (*'a*, *'b*) *lpoly* **is** λ *c*. *0* ⟨*proof*⟩

instance ⟨*proof*⟩
end

lift-definition *var-l* :: *'v* ⇒ (*'a* :: {*comm-monoid-mult*,*zero-neq-one*}, *'v*) *lpoly* **is** λ *x*. (λ *c*. *0*)(*Some x* := *1*) ⟨*proof*⟩
lift-definition *constant-l* :: (*'a* :: *zero*, *'v*) *lpoly* ⇒ *'a* **is** λ *c*. *c None* ⟨*proof*⟩
lift-definition *coeff-l* :: (*'a* :: *zero*, *'v*) *lpoly* ⇒ *'v* ⇒ *'a* **is** λ *c x*. *c* (*Some x*) ⟨*proof*⟩

lift-definition *vars-l* :: (*'a* :: *zero*, *'v*) *lpoly* ⇒ *'v set* **is** λ *c*. { *x*. *c* (*Some x*) ≠ *0*}
⟨*proof*⟩

lemma *finite-vars-l*[*simp*,*intro*]: *finite* (*vars-l p*)
⟨*proof*⟩

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

lemma *vars-l-var*[*simp*]: *vars-l* (*var-l x*) = {*x*} ⟨*proof*⟩

lemma *vars-l-plus*: *vars-l* (*p1* + *p2*) ⊆ *vars-l p1* ∪ *vars-l p2*
⟨*proof*⟩

lemma *vars-l-minus*: *vars-l* (*p1* − *p2*) ⊆ *vars-l p1* ∪ *vars-l p2*
⟨*proof*⟩

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

lemma *vars-l-zero*[*simp*]: *vars-l 0* = {}
⟨*proof*⟩

definition *eval-l* :: ('*a* :: *comm-ring*, '*v*) *assign* ⇒ ('*a*,'*v*) *lpoly* ⇒ '*a* **where**
eval-l α *p* = *constant-l p* + *sum* (λ *x*. *coeff-l p x* ∗ α *x*) (*vars-l p*)

lemma *eval-l-mono*: **assumes** *finite V vars-l p* ⊆ *V*
shows *eval-l* α *p* = *constant-l p* + *sum* (λ *x*. *coeff-l p x* ∗ α *x*) *V*
⟨*proof*⟩

lemma *eval-l-cong*: **assumes** ⋀ *x*. *x* ∈ *vars-l p* ⟹ α *x* = β *x*
shows *eval-l* α *p* = *eval-l* β *p*
⟨*proof*⟩

lemma *eval-l-0*[*simp*]: *eval-l* α *0* = *0* ⟨*proof*⟩

lemma *eval-l-plus*[*simp*]: *eval-l* α (*p1* + *p2*) = *eval-l* α *p1* + *eval-l* α *p2*
⟨*proof*⟩

lemma *eval-l-minus*[*simp*]: *eval-l* α (*p1* − *p2*) = *eval-l* α *p1* − *eval-l* α *p2*
⟨*proof*⟩

lemma *eval-l-uminus*[*simp*]: *eval-l* α (− *p*) = − *eval-l* α *p*
⟨*proof*⟩

lemma *eval-l-var*[*simp*]: *eval-l* α (*var-l x*) = α *x*
⟨*proof*⟩

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*⟩

lemma *vars-substitute-l*: $vars\text{-}l\ (substitute\text{-}l\ x\ p\ q) \subseteq vars\text{-}l\ p \cup (vars\text{-}l\ q - \{x\})$
⟨*proof*⟩

lemma *substitute-l-id*: $x \notin vars\text{-}l\ q \Longrightarrow substitute\text{-}l\ x\ p\ q = q$
⟨*proof*⟩

lemma *eval-substitute-l*: $eval\text{-}l\ \alpha\ (substitute\text{-}l\ x\ p\ q) = eval\text{-}l\ (\alpha(\ x := eval\text{-}l\ \alpha\ p))\ q$
⟨*proof*⟩

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

lift-definition *smult-l* :: $'a :: comm\text{-}ring \Rightarrow ('a,'v) lpoly \Rightarrow ('a,'v) lpoly$ **is**
$\lambda\ y\ c\ z.\ y * c\ z$
⟨*proof*⟩

lemma *coeff-smult-l*[*simp*]: $coeff\text{-}l\ (smult\text{-}l\ c\ p)\ x = c * coeff\text{-}l\ p\ x$
⟨*proof*⟩

lemma *constant-smult-l*[*simp*]: $constant\text{-}l\ (smult\text{-}l\ c\ p) = c * constant\text{-}l\ p$
⟨*proof*⟩

lemma *eval-smult-l*[*simp*]: $eval\text{-}l\ \alpha\ (smult\text{-}l\ c\ p) = c * eval\text{-}l\ \alpha\ p$
⟨*proof*⟩

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

lemma *eval-l-const-l-constant*: $eval\text{-}l\ \alpha\ (const\text{-}l\ (constant\text{-}l\ p)) = constant\text{-}l\ p$
⟨*proof*⟩

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

lemma *eval-substitute-all-l*: $eval\text{-}l\ \alpha\ (substitute\text{-}all\text{-}l\ \sigma\ p) = eval\text{-}l\ (\lambda\ x.\ eval\text{-}l\ \alpha\ (\sigma\ x))\ p$
⟨*proof*⟩

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

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

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

⟨*proof*⟩

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*⟩

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*⟩

lemma *lpoly-fun-of-eqI*: **assumes** ⋀ *x*. *fun-of-lpoly p x* = *fun-of-lpoly q x*
shows *p* = *q*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *vars-reorder-non-triv*: *vars-l* (*reorder-nontriv-var x p y*) ⊆ *insert x* (*insert y* (*vars-l p*))
⟨*proof*⟩

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* :: ($'v \times 'a$) *list*).
sorted (*map fst vcs*) ∧
distinct (*map fst vcs*) ∧
Ball (*snd* ‘ *set vcs*) ((≠) *0*)}
⟨*proof*⟩

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

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

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

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

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

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

code-datatype *lpoly-of*

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

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

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

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

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

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

lift-definition *var-lpoly-impl* :: $'v$ :: *linorder* ⇒ ($'a$ :: {*comm-monoid-mult*,*zero-neq-one*}, $'v$) *lpoly-impl* **is**

λ x. (0, [(x,1)]) ⟨proof⟩

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

lift-definition *uminus-lpoly-impl* :: ('*a* :: *ab-group-add*, '*v* :: *linorder*) *lpoly-impl* ⇒ ('*a*,'*v*) *lpoly-impl* **is**
λ (*c*, *vcs*). (*uminus c*, *map* (*map-prod id uminus*) *vcs*)
⟨proof⟩

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

fun *merge-coeffs-main* :: ('*a* :: *zero* ⇒ '*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*)
⟨proof⟩

definition *filter-0* **where** *filter-0* = *filter* (λ *p*. *snd p* ≠ *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*)) ((≠) *0*)
⟨proof⟩

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*)) ((≠) *0*) (**is** *?C*)
lookup-0 (*merge-coeffs f vxs vys*) *x* = *f* (*lookup-0 vxs x*) (*lookup-0 vys x*) (**is** *?D*)
⟨*proof*⟩

lift-definition *minus-lpoly-impl* :: (*'a* :: *ab-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*)
⟨*proof*⟩

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

lift-definition *plus-lpoly-impl* :: (*'a* :: *ab-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*)
⟨*proof*⟩

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

lift-definition *map-lpoly-impl* :: (*'a* :: *zero* ⇒ *'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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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

instantiation *lpoly* :: (*type*,*type*)*equal* **begin**
definition *equal-lpoly* :: (*'a*, *'b*) *lpoly* ⇒ (*'a*, *'b*) *lpoly* ⇒ *bool* **where** *equal-lpoly* = (=)
instance
⟨*proof*⟩
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* ⟨*proof*⟩
instance
  ⟨*proof*⟩
end

lift-definition *vars-coeffs-impl* :: ('*a* :: *zero*, '*v* :: *linorder*) *lpoly-impl* ⇒ ('*v* × '*a*) *list* **is** *snd* ⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*))

  ⟨*proof*⟩

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

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
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

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

lift-definition *coeff-lpoly-impl* :: *(′a :: zero, ′v :: linorder)lpoly-impl ⇒ ′v ⇒ ′a* **is**
λ (c,p) x. lookup-0 p x ⟨*proof*⟩

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

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

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

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

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

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*
⟨*proof*⟩

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

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

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

end

2 Linear Diophantine Equations and Inequalities

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

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

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

definition *satisfies-dleq* :: (*int*,*'v*) *assign* ⇒ *'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* $\equiv$ *Ball eqs* (*satisfies-dleq* α) $\wedge$ *Ball ineqs* (*satisfies-dlineq* α)

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

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

lemma *trivial-ineq-Some*: **assumes** *trivial-ineq c* = *Some b*
shows *b* = *satisfies-dlineq* α *c*
⟨*proof*⟩

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*⟩

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

end

3 Tightening

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

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

definition *tighten-ineq* :: '*v dlineq* $\Rightarrow$ '*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* $\neq$ {}
shows *satisfies-dlineq* α (*tighten-ineq p*) = *satisfies-dlineq* α *p*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *normalize-dleq-gcd*: **assumes** *normalize-dleq p* = (*g*,*q*)
and *p* $\neq$ *0*
shows *g* = *Gcd* (*insert* (*constant-l p*) (*coeff-l p* ' *vars-l p*))
and *g* $\geq$ *1*
and *normalize-dleq q* = (*1*,*q*)
⟨*proof*⟩

lemma *vars-l-normalize*: *normalize-dleq p* = (*g*,*q*) $\Longrightarrow$ *vars-l q* = *vars-l p*
⟨*proof*⟩

lemma *eval-normalize-dleq*: *normalize-dleq p* = (*g*,*q*) $\Longrightarrow$ *eval-l* α *p* = *g* $*$ *eval-l* α *q*
⟨*proof*⟩

lemma *gcd-unsat-detection*: **assumes** *g* = *Gcd* (*coeff-l p* ‘ *vars-l p*)
  and ¬ *g dvd constant-l p*
shows ¬ *satisfies-dleq* α *p*
⟨*proof*⟩

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*⟩

type-synonym ′*v dleq-sf* = ′*v* × (*int*,′*v*)*lpoly*

fun *satisfies-dleq-sf*:: (*int*,′*v*) *assign* ⇒ ′*v dleq-sf* ⇒ *bool* **where**
  satisfies-dleq-sf α (*x*,*p*) = (α *x* = *eval-l* α *p*)

type-synonym ′*v dleq-system* = ′*v dleq-sf set* × ′*v dleq set*

fun *satisfies-system* :: (*int*,′*v*) *assign* ⇒ ′*v dleq-system* ⇒ *bool* **where**
  satisfies-system α (*S*,*E*) = (*Ball S* (*satisfies-dleq-sf* α) ∧ *Ball E* (*satisfies-dleq* α))

fun *invariant-system* :: ′*v dleq-system* ⇒ *bool* **where**
  invariant-system (*S*,*E*) = (*Ball* (*fst* ‘ *S*) (λ *x*. *x* ∉ $\bigcup$ (*vars-l* ‘ (*snd* ‘ *S* ∪ *E*)) ∧ (∃! *e*. (*x*,*e*) ∈ *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*⟩

lemma *reorder-nontriv-var-sat*: ∃ *a*. *satisfies-dleq* (α(*y* := *a*)) (*reorder-nontriv-var x p y*)
⟨*proof*⟩

lemma *reorder-nontriv-var*: **assumes** *a*: *a* = *coeff-l p x a* ≠ *0*
  and *y*: *y* ∉ *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*⟩

inductive *griggio-equiv-step* :: ′*v dleq-system* ⇒ ′*v dleq-system* ⇒ *bool* **where**
griggio-solve: *abs* (*coeff-l p x*) = *1* ⟹ *e* = *reorder-for-var x p* ⟹
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*) ⟹ *g* ≥ *1* ⟹
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* ⇒ ′*v set* **where**
vars-system (*S*, *E*) = *fst* ‘ *S* ∪ ⋃ (*vars-l* ‘ (*snd* ‘ *S* ∪ *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*
⟨*proof*⟩

inductive *griggio-unsat* :: ′*v dleq* ⇒ *bool* **where**
griggio-gcd-unsat: ¬ *Gcd* (*coeff-l p* ‘ *vars-l p*) *dvd constant-l p* ⟹ *griggio-unsat p*
| *griggio-constant-unsat*: *vars-l p* = {} ⟹ *p* ≠ *0* ⟹ *griggio-unsat p*

lemma *griggio-unsat*: **assumes** *griggio-unsat p*
shows ¬ *satisfies-system* α (*S*, *insert p E*)
⟨*proof*⟩

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* ∉

$X \longrightarrow \alpha\ x = \beta\ x)))))$
| *invariant-state None* = ($\forall\ \alpha$. $\neg$ *Ball E* (*satisfies-dleq* α))

inductive-set *griggio-step* :: ('*v dleq-system* $\times$ '*v set*) *option rel* **where**
griggio-eq-step: *griggio-equiv-step SF TG* $\Longrightarrow$ (*Some* (*SF*,*X*), *Some* (*TG*, *X*)) $\in$ *griggio-step*
| *griggio-fail-step*: *griggio-unsat p* $\Longrightarrow$ (*Some* ((*S*,*insert p F*),*X*), *None*) $\in$ *griggio-step*
| *griggio-complex-step*: *coeff-l p x* $\neq$ *0*
$\Longrightarrow$ *q* = *reorder-nontriv-var x p y*
$\Longrightarrow$ *e* = *reorder-for-var x q*
$\Longrightarrow$ *y* $\notin$ *V* $\cup$ *X*
$\Longrightarrow$ (*Some* ((*S*,*insert p F*),*X*),
Some ((*insert* (*x*,*e*) (*map-prod id* (*substitute-l x e*) ' *S*), *substitute-l x e* ' *insert p F*), *insert y X*))
$\in$ *griggio-step*

lemma *griggio-step*: **assumes** (*A*,*B*) $\in$ *griggio-step*
and *invariant-state A*
shows *invariant-state B*
⟨*proof*⟩

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

lemma *griggio-steps*: **assumes** (*Some* (({},*E*),{}), *SFO*) $\in$ *griggio-step*^* (**is** (*?I*,-) $\in$ -)
shows *invariant-state SFO*
⟨*proof*⟩

lemma *griggio-fail*: **assumes** (*Some* (({},*E*),{}), *None*) $\in$ *griggio-step*^*
shows $\nexists\ \alpha.\ \alpha \models_{dio} (E, \{\})$
⟨*proof*⟩

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*⟩

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

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

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

corollary *griggio-success-equivalence*:
fixes P :: $'i \Rightarrow (int,'v)lpoly$ **and** f :: $('i \Rightarrow int) \Rightarrow bool$
assumes $(Some\ ((\{\},E),\{\}),\ Some\ ((S,\{\}),X)) \in griggio\text{-}step\widehat{\ }*$
and σ: $\sigma = solution\text{-}subst\ S\text{-}list$
and $S\text{-}list$: $set\ S\text{-}list = S$
and vV: $\bigwedge\ i.\ vars\text{-}l\ (P\ i) \subseteq V$
shows
$(\exists\ \alpha.\ f\ (\lambda\ i.\ eval\text{-}l\ \alpha\ (substitute\text{-}all\text{-}l\ \sigma\ (P\ i))))$
$\longleftrightarrow (\exists\ \alpha.\ f\ (\lambda\ i.\ eval\text{-}l\ \alpha\ (P\ i)) \wedge Ball\ E\ (satisfies\text{-}dleq\ \alpha))$
⟨*proof*⟩

end
end

end

4.2 Executable Algorithm

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

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

else if g dvd c then Inl (*sdiv-l p g*) *else Inr False*)

lemma *simplify-dleq-0*: **assumes** *simplify-dleq p* = *Inr True*
shows *p* = *0*
⟨*proof*⟩

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

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

definition *size-dleq* :: ′*v dleq* ⇒ *int* **where** *size-dleq p* = *sum* (*abs o coeff-l p*) (*vars-l p*)

lemma *size-dleq-pos*: *size-dleq p* ≥ *0* ⟨*proof*⟩

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*⟩

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*⟩

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*^∗
⟨*proof*⟩

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* ⟹ ∄ α. α $\models_{dio}$ (*set e*, {})
res = *Some s* ⟹ *adjust-assign s* α $\models_{dio}$ (*set e*, {})
res = *Some s* ⟹ σ = *solution-subst s* ⟹
f (λ *i*. *eval-l* α (*substitute-all-l* σ (*P i*))) ⟹
β = *adjust-assign s* α ⟹
f (λ *i*. *eval-l* β (*P i*)) ∧ β $\models_{dio}$ (*set e*, {})
res = *Some s* ⟹ σ = *solution-subst s* ⟹ (⋀ *i*. *vars-l* (*P i*) ⊆ *set v*) ⟹
f (λ *i*. *eval-l* α (*P i*)) ∧ α $\models_{dio}$ (*set e*, {}) ⟹
∃ γ. *f* (λ *i*. *eval-l* γ (*substitute-all-l* σ (*P i*)))
⟨*proof*⟩

definition *equality-elim-for-inequalities* :: *'v dleq list* ⇒ *'v dlineq list* ⇒
(*'v dleq list* × ((*int*,*'v*)*assign* ⇒ (*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 ⇒ *None*
| *Some s* ⇒ *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* ⟹ ∄ α. α $\models_{dio}$ (*set eqs*, {})
res = *Some* (*ineqs'*, *adj*) ⟹ α $\models_{dio}$ ({}, *set ineqs'*) ⟹ (*adj* α) $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) ⟹ ∄ α. α $\models_{dio}$ ({}, *set ineqs'*) ⟹ ∄ α. α $\models_{dio}$ (*set eqs*, *set ineqs*)
res = *Some* (*ineqs'*, *adj*) ⟹ *length ineqs'* = *length ineqs*
⟨*proof*⟩

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*⟩

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*⟩

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*: $C \subseteq cs \cup s$ ‘ *I*
    and *unsat*: $\nexists\ v.\ \forall\ c \in C.\ v \models_{le}\ c$
    and *min*: $\bigwedge D.\ D \subset C \Longrightarrow \exists\ v.\ \forall\ c \in D.\ v \models_{le}\ c$
    and *sol*: $\forall\ c \in cs \cup n$ ‘ $I.\ v \models_{le}\ c$
    and *i*: $i \in I$ *s* $i \in C$
  shows (*p i*)⦃*v*⦄ = *co i*
⟨*proof*⟩

end

5.2 Algorithm to Detect all Implicit Equalities in ℚ

Use incremental simplex algorithm to recursively detect all implied equalities.

theory *Equality-Detection-Impl*
  imports
    Equality-Detection-Theory
    Simplex.Simplex-Incremental
    Deriving.Compare-Instances
begin

lemma *indexed-sat-mono*: $(S,v) \models_{ics} cs \Longrightarrow T \subseteq S \Longrightarrow (T,v) \models_{ics} cs$
  ⟨*proof*⟩

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

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

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

fun *le-of-constraint* :: *constraint* ⇒ *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* ⇒ *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* ⇒ *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* ⇒ *bool* **where**
is-no-equality (*EQ p c*) = *False*
| *is-no-equality* - = *True*

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

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

lemma *le-of-constraints*: *Ball cs is-no-equality* ⟹ *v* $\models_{cs}$ *cs* ⟷ (∀ *c* ∈ *cs*. *v* $\models_{le}$ *le-of-constraint c*)

⟨proof⟩

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*)
⟨proof⟩

lemma *is-nstrict-iff*: *is-nstrict c* = (¬ *is-strict c* ∧ ¬ *is-equality c*)
⟨proof⟩

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*)
⟨proof⟩

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

(*const-of-constraint c*)) ‘ *cs*
le-of-constraint ‘ (*make-strict* ‘ *cs*) = (λ *c*. *Le-Constraint Lt-Rel* (*poly-of-constraint c*) (*const-of-constraint c*)) ‘ *cs*
⟨*proof*⟩

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

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

derive *compare-order eqd-index*

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

definition *init-constraints* :: *constraint list* ⇒ *eqd-index i-constraint list* × *nat list* × *nat list* × *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*⟩

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 interfact does not ensure distinctness of C
s3 = *sum.projr* (*assert-all-simplex* (*map FIneq eq′*) *s2*); — and permantly add the flipped inequalities
s4 = *fst* (*check-simplex s3*); — this check will succeed, no unsat can be reported here
ineq′ = *filter* (λ *i*. *i* ∉ *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 = (∀ *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*) ∪ *TmpSIneq* ‘ *set ineq*, *v*) $\models_{ics}$ *set ics*)

lemma *init-eq-finder-rat*: *init-eq-finder-rat* = *None* ⟹ ∄ *v*. $v \models_{cs}$ *set cs*
init-eq-finder-rat = *Some* (*s*, *ineq*, *sineq*, *eq*) ⟹
checked-simplex ics (*set* (*index-of ineqs sineqs eq*)) *s*
∧ *eq* = *eqs* ∧ *ineq* = *ineqs* ∧ *sineq* = *sineqs*
∧ *equiv-to-cs eq*
∧ *distinct* (*ineq* @ *sineq* @ *eq*)
∧ *set* (*ineq* @ *sineq* @ *eq*) = {*0* ..< *length cs*}
⟨*proof*⟩

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

lemma *eq-finder-rat-in-ctxt*: *eq-finder-rat* = *None* $\Longrightarrow \nexists$ *v*. *v* $\models_{cs}$ *set cs*
eq-finder-rat = *Some* (*eq-idx*, *v-sol*) $\Longrightarrow$ {*i* . *i* < *length cs* ∧ *is-equality* (*cs* ! *i*)} ⊆ *set eq-idx* ∧
set eq-idx ⊆ {*0* ..< *length cs*} ∧
distinct eq-idx (**is** - $\Longrightarrow$ *?main1*)
eq-finder-rat = *Some* (*eq-idx*, *v-sol*) $\Longrightarrow$
set feq = *make-equality* ‘ (!) *cs* ‘ *set eq-idx* $\Longrightarrow$
set fineq = (!) *cs* ‘ ({*0* ..< *length cs*} − *set eq-idx*) $\Longrightarrow$
(∀ *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** - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ *?main2*)
⟨*proof*⟩

end
end

lemma *eq-finder-rat*:
eq-finder-rat cs = *None* $\Longrightarrow \nexists$ *v*. *v* $\models_{cs}$ *set cs* (**is** *?p1* $\Longrightarrow$ *?g1*)
eq-finder-rat cs = *Some* (*eq-idx*, *v-sol*) $\Longrightarrow$
{*i* . *i* < *length cs* ∧ *is-equality* (*cs* ! *i*)} ⊆ *set eq-idx* ∧
set eq-idx ⊆ {*0* ..< *length cs*} ∧
distinct eq-idx (**is** *?p2* $\Longrightarrow$ *?g2*)
eq-finder-rat cs = *Some* (*eq-idx*, *v-sol*) $\Longrightarrow$
set eq = *make-equality* ‘ (!) *cs* ‘ *set eq-idx* $\Longrightarrow$
set ineq = (!) *cs* ‘ ({*0* ..< *length cs*} − *set eq-idx*) $\Longrightarrow$
(∀ *v*. *v* $\models_{cs}$ *set cs* ⟷ *v* $\models_{cs}$ (*set eq* ∪ *set ineq*)) ∧
Ball (*set eq*) *is-equality* ∧ *Ball* (*set ineq*) *is-no-equality* ∧
(*v-sol* $\models_{cs}$ (*set eq* ∪ *make-strict* ‘ *set ineq*))
(**is** *?p2* $\Longrightarrow$ *?p3* $\Longrightarrow$ *?p4* $\Longrightarrow$ *?g3*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *valuate-sum-list*: *valuate* (*sum-list ps*) α = *sum-list* (*map* (λ *p*. *valuate p* α) *ps*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

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*)

⟨*proof*⟩

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

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

lemma *eq-finder-int*: **assumes**
res: *eq-finder-int ineqs* = *res*
shows *res* = *None* $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ ({}, *set ineqs*)
res = *Some* (*eqs*, *ineqs′*) $\Longrightarrow \alpha \models_{dio}$ ({}, *set ineqs*) $\longleftrightarrow \alpha \models_{dio}$ (*set eqs*, *set ineqs′*)
res = *Some* (*eqs*, *ineqs′*) $\Longrightarrow \exists\ \alpha.\ \alpha \models_{cs}$ (*make-strict* ‘ *dlineq-to-constraint* ‘ *set ineqs′*)
res = *Some* (*eqs*, *ineqs′*) ⟹ *length ineqs* = *length eqs* + *length ineqs′*
⟨*proof*⟩

end

6 A Combined Preprocessor

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

theory *Dio-Preprocessor*
imports
Linear-Polynomial-Impl
Linear-Diophantine-Solver-Impl
Diophantine-Tightening
Linear-Diophantine-Eq-Finder
begin

Combine equality elimination and tightening in one algorithm

definition *dio-elim-equations-and-tighten* :: *var dleq list* ⇒ *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 $res = None \Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio} (set\ eqs,\ set\ ineqs)$
$res = Some\ (ineqs',\ adj) \Longrightarrow \alpha \models_{dio} (\{\},\ set\ ineqs') \Longrightarrow \beta = adj\ \alpha \Longrightarrow \beta \models_{dio} (set\ eqs,\ set\ ineqs)$
$res = Some\ (ineqs'\ ,\ adj) \Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio} (\{\},\ set\ ineqs') \Longrightarrow \nexists\ \alpha\ .\ \alpha \models_{dio} (set\ eqs,\ set\ ineqs)$
$res = Some\ (ineqs',\ adj) \Longrightarrow length\ ineqs' \leq length\ ineqs$
⟨*proof*⟩

Now all three preprocessing steps are combined.

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

function *dio-preprocess-main* :: $(int,\ var)\ lpoly\ list \Rightarrow ((int,\ var)\ lpoly\ list \times ((int,var)assign \Rightarrow (int,var)assign))\ option$ **where**
$dio\text{-}preprocess\text{-}main\ ineqs = (case\ eq\text{-}finder\text{-}int\ ineqs\ of\ None \Rightarrow None$
$\mid Some\ (eqs,\ ineqs') \Rightarrow (case\ eqs\ of\ [] \Rightarrow Some\ (ineqs',\ id)$
$\mid \text{-} \Rightarrow (case\ dio\text{-}elim\text{-}equations\text{-}and\text{-}tighten\ eqs\ ineqs'\ of\ None \Rightarrow None$
$\mid Some\ (ineqs'',\ adj) \Rightarrow map\text{-}option\ (map\text{-}prod\ id\ (\lambda\ adj'.\ adj\ o\ adj'))\ (dio\text{-}preprocess\text{-}main\ ineqs''))))$
⟨*proof*⟩

termination
⟨*proof*⟩

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

lemma *dio-preprocess-main*: **assumes**
res: *dio-preprocess-main ineqs = res*
shows $res = None \Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio} (\{\},\ set\ ineqs)$
$res = Some\ (ineqs',\ adj) \Longrightarrow \alpha \models_{dio} (\{\},\ set\ ineqs') \Longrightarrow (adj\ \alpha) \models_{dio} (\{\},\ set\ ineqs)$
$res = Some\ (ineqs'\ ,\ adj) \Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio} (\{\},\ set\ ineqs') \Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio} (\{\},\ set\ ineqs)$
$res = Some\ (ineqs',\ adj) \Longrightarrow \exists\ \alpha.\ \alpha \models_{cs} (make\text{-}strict\ `\ dlineq\text{-}to\text{-}constraint\ `\ set\ ineqs')$
⟨*proof*⟩

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

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

(*dio-preprocess-main ineqs′*))

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

lemma *dio-preprocess*: **assumes** *res*: *dio-preprocess eqs ineqs* = *res*
  shows *res* = *None* $\Longrightarrow \nexists\ \alpha.\ \alpha \models_{dio}$ (*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*⟩

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*

  ⟨*proof*⟩

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*
⟨*proof*⟩

Griggio, example 2, unsat detection by preprocessing

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

definition *griggio-example-2-ineqs* :: (*int*,*var*) *lpoly list* **where**
griggio-example-2-ineqs = (*let x1* = *var-l 1*; *x2* = *var-l 2*; *x3* = *var-l 3 in*
[− *smult-l 2 x1* − *x2* − *x3* + *const-l 7*,
smult-l 2 x1 + *x2* + *x3* − *const-l 8*])

lemma *case dio-preprocess griggio-example-2-eqs griggio-example-2-ineqs*
of None ⇒ *True* | *Some* - ⇒ *False*
⟨*proof*⟩

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.