Diophantine Equations*

Florian MeSSner Julian Parsert Jonas Schöpf
Christian Sternagel

September 13, 2023

Abstract

In this entry we formalize Huet's [1] bounds for minimal solutions of homogenous linear Diophantine equations (HLDEs). Based on these bounds, we further provide a certified algorithm for computing the set of all minimal solutions of a given HLDE.

Contents

*This work is supported by the Austrian Science Fund (FWF): project P27502.

1 Vectors as Lists of Naturals

theory *List-Vector*
  imports *Main*
begin

lemma *lex-lengthD*: $(x, y) \in lex\ P \Longrightarrow length\ x = length\ y$
  by (*auto simp*: *lexord-lex*)

lemma *lex-take-index*:
  assumes $(xs, ys) \in lex\ r$
  obtains *i* **where** $length\ ys = length\ xs$
    and $i < length\ xs$ **and** $take\ i\ xs = take\ i\ ys$
    and $(xs\ !\ i, ys\ !\ i) \in r$
proof −
  obtain $n\ us\ x\ xs'\ y\ ys'$ **where** $(xs, ys) \in lexn\ r\ n$ **and** $length\ xs = n$ **and** $length\ ys = n$
    and $xs = us\ @\ x\ \#\ xs'$ **and** $ys = us\ @\ y\ \#\ ys'$ **and** $(x, y) \in r$
    using *assms* **by** (*fastforce simp*: *lex-def lexn-conv*)
  then show *?thesis* **by** (*intro that* [*of length us*]) *auto*
qed

lemma *mods-with-nats*:
  assumes $(v::nat) > w$
    and $(v * b)\ mod\ a = (w * b)\ mod\ a$
  shows $((v - w) * b)\ mod\ a = 0$
  using *assms* **by** (*simp add*: *mod-eq-dvd-iff-nat algebra-simps*)

— The 0-vector of length n.
abbreviation $zeroes :: nat \Rightarrow nat\ list$
  where
    $zeroes\ n \equiv replicate\ n\ 0$

lemma *rep-upd-unit*:
  assumes $x = (zeroes\ n)[i := a]$
  shows $\forall j < length\ x.\ (j \neq i \longrightarrow x\ !\ j = 0) \wedge (j = i \longrightarrow x\ !\ j = a)$

using *assms* **by** *simp*

definition *nonzero-iff*: *nonzero xs* ⟷ (∃ *x*∈*set xs*. *x* ≠ *0*)

lemma *nonzero-append* [*simp*]:
nonzero (*xs* @ *ys*) ⟷ *nonzero xs* ∨ *nonzero ys* **by** (*auto simp*: *nonzero-iff*)

1.1 The Inner Product

definition *dotprod* :: *nat list* ⇒ *nat list* ⇒ *nat* (**infixl** · *70*)
where
xs · *ys* = (∑ *i*<*min* (*length xs*) (*length ys*). *xs* ! *i* ∗ *ys* ! *i*)

lemma *dotprod-code* [*code*]:
xs · *ys* = *sum-list* (*map* (λ(*x*, *y*). *x* ∗ *y*) (*zip xs ys*))
by (*auto simp*: *dotprod-def sum-list-sum-nth lessThan-atLeast0*)

lemma *dotprod-commute*:
assumes *length xs* = *length ys*
shows *xs* · *ys* = *ys* · *xs*
using *assms* **by** (*auto simp*: *dotprod-def mult.commute*)

lemma *dotprod-Nil* [*simp*]: [] · [] = *0*
by (*simp add*: *dotprod-def*)

lemma *dotprod-Cons* [*simp*]:
(*x* # *xs*) · (*y* # *ys*) = *x* ∗ *y* + *xs* · *ys*
unfolding *dotprod-def* **and** *length-Cons* **and** *min-Suc-Suc* **and** *sum.lessThan-Suc-shift*
by *auto*

lemma *dotprod-1-right* [*simp*]:
xs · *replicate* (*length xs*) *1* = *sum-list xs*
by (*induct xs*) (*simp-all*)

lemma *dotprod-0-right* [*simp*]:
xs · *zeroes* (*length xs*) = *0*
by (*induct xs*) (*simp-all*)

lemma *dotprod-unit* [*simp*]:
assumes *length a* = *n*
and *k* < *n*
shows *a* · (*zeroes n*)[*k* := *zk*] = *a* ! *k* ∗ *zk*
using *assms* **by** (*induct a arbitrary*: *k n*) (*auto split*: *nat.splits*)

lemma *dotprod-gt0*:
assumes *length x* = *length y* **and** ∃ *i*<*length y*. *x* ! *i* > *0* ∧ *y* ! *i* > *0*
shows *x* · *y* > *0*
using *assms* **by** (*induct x y rule*: *list-induct2*) (*fastforce simp*: *nth-Cons split*: *nat.splits*)+

lemma *dotprod-gt0D*:
  assumes *length x* = *length y*
    and *x* · *y* > *0*
  shows ∃ *i*<*length y*. *x* ! *i* > *0* ∧ *y* ! *i* > *0*
  using *assms* **by** (*induct x y rule*: *list-induct2*) (*auto simp*: *Ex-less-Suc2*)

lemma *dotprod-gt0-iff* [*iff*]:
  assumes *length x* = *length y*
  shows *x* · *y* > *0* ⟷ (∃ *i*<*length y*. *x* ! *i* > *0* ∧ *y* ! *i* > *0*)
  using *assms* **and** *dotprod-gt0D* **and** *dotprod-gt0* **by** *blast*

lemma *dotprod-append*:
  assumes *length a* = *length b*
  shows(*a* @ *x*) · (*b* @ *y*) = *a* · *b* + *x* · *y*
  using *assms* **by** (*induct a b rule*: *list-induct2*) *auto*

lemma *dotprod-le-take*:
  assumes *length a* = *length b*
    and *k* ≤ *length a*
  shows*take k a* · *take k b* ≤ *a* · *b*
  using *assms* **and** *append-take-drop-id* [*of k a*] **and** *append-take-drop-id* [*of k b*]
  by (*metis add-right-cancel leI length-append length-drop not-add-less1 dotprod-append*)

lemma *dotprod-le-drop*:
  assumes *length a* = *length b*
    and *k* ≤ *length a*
  shows *drop k a* · *drop k b* ≤ *a* · *b*
  using *assms* **and** *append-take-drop-id* [*of k a*] **and** *append-take-drop-id* [*of k b*]
  by (*metis dotprod-append length-take order-refl trans-le-add2*)

lemma *dotprod-is-0* [*simp*]:
  assumes *length x* = *length y*
  shows *x* · *y* = *0* ⟷ (∀ *i*<*length y*. *x* ! *i* = *0* ∨ *y* ! *i* = *0*)
  using *assms* **by** (*metis dotprod-gt0-iff neq0-conv*)

lemma *dotprod-eq-0-iff*:
  assumes *length x* = *length a*
    and *0* ∉ *set a*
  shows *x* · *a* = *0* ⟷ (∀ *e* ∈ *set x*. *e* = *0*)
  using *assms* **by** (*fastforce simp*: *in-set-conv-nth*)

lemma *dotprod-eq-nonzero-iff*:
  assumes *a* · *x* = *b* · *y* **and** *length x* = *length a* **and** *length y* = *length b*
    and *0* ∉ *set a* **and** *0* ∉ *set b*
  shows *nonzero x* ⟷ *nonzero y*
  using *assms* **by** (*auto simp*: *nonzero-iff*) (*metis dotprod-commute dotprod-eq-0-iff*
neq0-conv)+

lemma *eq-0-iff*:
  $xs = zeroes\ n \longleftrightarrow length\ xs = n \wedge (\forall x{\in}set\ xs.\ x = 0)$
  using *in-set-replicate* [*of - n 0*] **and** *replicate-eqI* [*of xs n 0*] **by** *auto*

lemma *not-nonzero-iff*: $\neg\ nonzero\ x \longleftrightarrow x = zeroes\ (length\ x)$
  by (*auto simp*: *nonzero-iff replicate-length-same eq-0-iff*)

lemma *neq-0-iff'*:
  $xs \neq zeroes\ n \longleftrightarrow length\ xs \neq n \vee (\exists x{\in}set\ xs.\ x > 0)$
  by (*auto simp*: *eq-0-iff*)

lemma *dotprod-pointwise-le*:
  assumes $length\ as = length\ xs$
    and $i < length\ as$
  shows $as\ !\ i * xs\ !\ i \leq as \cdot xs$
proof −
  have $as \cdot xs = (\sum i{<}min\ (length\ as)\ (length\ xs).\ as\ !\ i * xs\ !\ i)$
    by (*simp add*: *dotprod-def*)
  then show *?thesis*
    using *assms* **by** (*auto intro*: *member-le-sum*)
qed

lemma *replicate-dotprod*:
  assumes $length\ y = n$
  shows $replicate\ n\ x \cdot y = x * sum\text{-}list\ y$
proof −
  have $x * (\sum i{<}length\ y.\ \ y\ !\ i) = (\sum i{<}length\ y.\ x * y\ !\ i)$
    using *sum-distrib-left* **by** *blast*
  then show *?thesis*
    using *assms* **by** (*auto simp*: *dotprod-def sum-list-sum-nth atLeast0LessThan*)
qed

1.2 The Pointwise Order on Vectors

definition *less-eq* :: $nat\ list \Rightarrow nat\ list \Rightarrow bool$ (-/ $\leq_v$ - [*51*, *51*] *50*)
  where
    $xs \leq_v ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall i{<}length\ xs.\ xs\ !\ i \leq ys\ !\ i)$

definition *less* :: $nat\ list \Rightarrow nat\ list \Rightarrow bool$ (-/ $<_v$ - [*51*, *51*] *50*)
  where
    $xs <_v ys \longleftrightarrow xs \leq_v ys \wedge \neg\ ys \leq_v xs$

interpretation *order-vec*: *order less-eq less*
  by (*standard*, *auto simp add*: *less-def less-eq-def dual-order.antisym nth-equalityI*)
(*force*)

lemma *less-eqI* [*intro?*]: $length\ xs = length\ ys \Longrightarrow \forall i{<}length\ xs.\ xs\ !\ i \leq ys\ !\ i \Longrightarrow xs \leq_v ys$
  by (*auto simp*: *less-eq-def*)

lemma *le0* [*simp*, *intro*]: *zeroes* (*length xs*) $\leq_v$ *xs* **by** (*simp add*: *less-eq-def*)

lemma *le-list-update* [*simp*]:
  assumes *xs* $\leq_v$ *ys* **and** $i <$ *length ys* **and** $z \leq$ *ys* ! *i*
  shows *xs*[$i := z$] $\leq_v$ *ys*
  using *assms* **by** (*auto simp*: *less-eq-def nth-list-update*)

lemma *le-Cons*: *x* # *xs* $\leq_v$ *y* # *ys* $\longleftrightarrow$ $x \leq y \wedge$ *xs* $\leq_v$ *ys*
  by (*auto simp add*: *less-eq-def nth-Cons split*: *nat.splits*)

lemma *zero-less*:
  assumes *nonzero x*
  shows *zeroes* (*length x*) $<_v$ *x*
  using *assms* **and** *eq-0-iff order-vec.dual-order.strict-iff-order*
  by (*auto simp*: *nonzero-iff*)

lemma *le-append*:
  assumes *length xs* = *length vs*
  shows *xs* @ *ys* $\leq_v$ *vs* @ *ws* $\longleftrightarrow$ *xs* $\leq_v$ *vs* $\wedge$ *ys* $\leq_v$ *ws*
  using *assms*
  by (*auto simp*: *less-eq-def nth-append*)
    (*metis add.commute add-diff-cancel-left′ nat-add-left-cancel-less not-add-less2*)

lemma *less-Cons*:
  (*x* # *xs*) $<_v$ (*y* # *ys*) $\longleftrightarrow$ *length xs* = *length ys* $\wedge$ ($x \leq y \wedge$ *xs* $<_v$ *ys* $\vee$ $x < y$ $\wedge$ *xs* $\leq_v$ *ys*)
  by (*simp add*: *less-def less-eq-def All-less-Suc2*) (*auto dest*: *leD*)

lemma *le-length* [*dest*]:
  assumes *xs* $\leq_v$ *ys*
  shows *length xs* = *length ys*
  using *assms* **by** (*simp add*: *less-eq-def*)

lemma *less-length* [*dest*]:
  assumes $x <_v y$
  shows *length x* = *length y*
  using *assms* **by** (*auto simp*: *less-def*)

lemma *less-append*:
  assumes *xs* $<_v$ *vs* **and** *ys* $\leq_v$ *ws*
  shows *xs* @ *ys* $<_v$ *vs* @ *ws*
proof –
  have *length xs* = *length vs*
    using *assms* **by** *blast*
  then show *?thesis*
    using *assms* **by** (*induct xs vs rule*: *list-induct2*) (*auto simp*: *less-Cons le-append le-length*)
qed

lemma *less-appendD*:
  assumes xs @ $ys <_v vs$ @ ws
    and *length* xs = *length* vs
  shows $xs <_v vs \vee ys <_v ws$
  by (*auto*) (*metis* (*no-types*, *lifting*) *assms le-append order-vec.order.strict-iff-order*)

lemma *less-append-cases*:
  assumes xs @ $ys <_v vs$ @ ws **and** *length* xs = *length* vs
  obtains $xs <_v vs$ **and** $ys \leq_v ws$ | $xs \leq_v vs$ **and** $ys <_v ws$
  using *assms* **and** *that*
  by (*metis le-append less-appendD order-vec.order.strict-implies-order*)

lemma *less-append-swap*:
  assumes x @ $y <_v u$ @ v
    and *length* x = *length* u
  shows y @ $x <_v v$ @ u
  using *assms*(*2*, *1*)
  by (*induct x u rule*: *list-induct2*)
    (*auto simp*: *order-vec.order.strict-iff-order le-Cons le-append le-length*)

lemma *le-sum-list-less*:
  assumes $xs \leq_v ys$
    and *sum-list* xs < *sum-list* ys
  shows $xs <_v ys$
proof −
  have *length* xs = *length* ys **and** $\forall$ i<*length* ys. xs ! $i \leq ys$! i
    using *assms* **by** (*auto simp*: *less-eq-def*)
  then show *?thesis*
    using ‹*sum-list* xs < *sum-list* ys›
    by (*induct xs ys rule*: *list-induct2*)
      (*auto simp*: *less-Cons All-less-Suc2 less-eq-def*)
qed

lemma *dotprod-le-right*:
  assumes $v \leq_v w$
    and *length* b = *length* w
  shows $b \cdot v \leq b \cdot w$
  using *assms* **by** (*auto simp*: *dotprod-def less-eq-def intro*: *sum-mono*)

lemma *dotprod-pointwise-le-right*:
  assumes *length* z = *length* u
    and *length* u = *length* v
    and $\forall$ i<*length* v. u ! $i \leq v$! i
  shows $z \cdot u \leq z \cdot v$
  using *assms* **by** (*intro dotprod-le-right*) (*auto intro*: *less-eqI*)

lemma *dotprod-le-left*:
  assumes $v \leq_v w$

```
    and length b = length w
  shows v · b ≤ w · b
  using assms by (simp add: dotprod-le-right dotprod-commute le-length)

lemma dotprod-le:
  assumes x ≤v u and y ≤v v
    and length y = length x and length v = length u
  shows x · y ≤ u · v
  using assms by (metis dotprod-le-left dotprod-le-right le-length le-trans)

lemma dotprod-less-left:
  assumes length b = length w
    and 0 ∉ set b
    and v <v w
  shows v · b < w · b
proof –
  have length v = length w using assms
    using less-eq-def order-vec.order.strict-implies-order by blast
  then show ?thesis
    using assms
  proof (induct v w arbitrary: b rule: list-induct2)
    case (Cons x xs y ys)
    then show ?case
     by (cases b) (auto simp: less-Cons add-mono-thms-linordered-field dotprod-le-left)
  qed simp
qed

lemma le-append-swap:
  assumes length y = length v
    and x @ y ≤v w @ v
  shows y @ x ≤v v @ w
proof –
  have length w = length x using assms by auto
  with assms show ?thesis
    by (induct y v arbitrary: x w rule: list-induct2) (auto simp: le-Cons le-append)
qed

lemma le-append-swap-iff:
  assumes length y = length v
  shows y @ x ≤v v @ w ⟷ x @ y ≤v w @ v
  using assms and le-append-swap
  by (auto) (metis (no-types, lifting) add-left-imp-eq le-length length-append)

lemma unit-less:
  assumes i < n
    and x <v (zeroes n)[i := b]
  shows x ! i < b ∧ (∀j<n. j ≠ i ⟶ x ! j = 0)
proof
  show x ! i < b
```

```
    using assms less-def by fastforce
next
  have x ≤_v (zeroes n)[i := b] by (simp add: assms order-vec.less-imp-le)
  then show ∀ j<n. j ≠ i ⟶ x ! j = 0 by (auto simp: less-eq-def)
qed

lemma le-sum-list-mono:
  assumes xs ≤_v ys
  shows sum-list xs ≤ sum-list ys
  using assms and sum-list-mono [of [0..<length ys] (!) xs (!) ys]
  by (auto simp: less-eq-def) (metis map-nth)

lemma sum-list-less-diff-Ex:
  assumes u ≤_v y
    and sum-list u < sum-list y
  shows ∃ i<length y. u ! i < y ! i
proof −
  have length u = length y and ∀ i<length y. u ! i ≤ y ! i
    using ‹u ≤_v y› by (auto simp: less-eq-def)
  then show ?thesis
    using ‹sum-list u < sum-list y›
    by (induct u y rule: list-induct2) (force simp: Ex-less-Suc2 All-less-Suc2)+
qed

lemma less-vec-sum-list-less:
  assumes v <_v w
  shows sum-list v < sum-list w
  using assms
proof −
  have length v = length w
    using assms less-eq-def less-imp-le by blast
  then show ?thesis
    using assms
  proof (induct v w rule: list-induct2)
    case (Cons x xs y ys)
    then show ?case
      using length-replicate less-Cons order-vec.order.strict-iff-order by force
  qed simp
qed

definition maxne0 :: nat list ⇒ nat list ⇒ nat
  where
    maxne0 x a =
      (if length x = length a ∧ (∃ i<length a. x ! i ≠ 0)
      then Max {a ! i | i. i < length a ∧ x ! i ≠ 0}
      else 0)

lemma maxne0-le-Max:
  maxne0 x a ≤ Max (set a)
```

by (*auto simp*: *maxne0-def nonzero-iff in-set-conv-nth*) *simp*

lemma *maxne0-Nil* [*simp*]:
maxne0 [] *as* = *0*
maxne0 xs [] = *0*
by (*auto simp*: *maxne0-def*)

lemma *maxne0-Cons* [*simp*]:
maxne0 (*x* # *xs*) (*a* # *as*) =
(*if length xs* = *length as then*
(*if x* = *0 then maxne0 xs as else max a* (*maxne0 xs as*))
else 0)
proof −
let *?a* = *a* # *as* **and** *?x* = *x* # *xs*
have *eq*: {*?a* ! *i* | *i*. *i* < *length ?a* ∧ *?x* ! *i* ≠ *0*} =
(*if x* > *0 then* {*a*} *else* {}) ∪ {*as* ! *i* | *i*. *i* < *length as* ∧ *xs* ! *i* ≠ *0*}
by (*auto simp*: *nth-Cons split*: *nat.splits*) (*metis Suc-pred*)+
show *?thesis*
unfolding *maxne0-def* **and** *eq*
by (*auto simp*: *less-Suc-eq-0-disj nth-Cons′ intro*: *Max-insert2*)
qed

lemma *maxne0-times-sum-list-gt-dotprod*:
assumes *length b* = *length ys*
shows *maxne0 ys b* ∗ *sum-list ys* ≥ *b* · *ys*
using *assms*
apply (*induct b ys rule*: *list-induct2*)
apply (*auto simp*: *max-def ring-distribs add-mono-thms-linordered-semiring*(*1*))
by (*meson leI le-trans mult-less-cancel2 nat-less-le*)

lemma *max-times-sum-list-gt-dotprod*:
assumes *length b* = *length ys*
shows *Max* (*set b*) ∗ *sum-list ys* ≥ *b* · *ys*
proof −
have ∀ *e* ∈ *set b* . *Max* (*set b*) ≥ *e* **by** *simp*
then have *replicate* (*length ys*) (*Max* (*set b*)) · *ys* ≥ *b* · *ys* (**is** *?rep* ≥ -)
by (*metis assms dotprod-pointwise-le-right dotprod-commute*
length-replicate nth-mem nth-replicate)
moreover have *Max* (*set b*) ∗ *sum-list ys* = *?rep*
using *replicate-dotprod* [*of ys* - *Max* (*set b*)] **by** *auto*
ultimately show *?thesis*
by (*simp add*: *assms*)
qed

lemma *maxne0-mono*:
assumes $y \le_v x$
shows *maxne0 y a* ≤ *maxne0 x a*
proof (*cases length y* = *length a*)
case *True*

have *length y* = *length x* **using** *assms* **by** (*auto*)
then show *?thesis*
using *assms* **and** *True*
proof (*induct y x arbitrary*: *a rule*: *list-induct2*)
case (*Cons x xs y ys*)
then show *?case* **by** (*cases a*) (*force simp*: *less-eq-def All-less-Suc2 le-max-iff-disj*)+
qed *simp*
next
case *False*
then show *?thesis*
using *assms* **by** (*auto simp*: *maxne0-def*)
qed

lemma *all-leq-Max*:
assumes $x \leq_v y$
and $x \neq []$
shows $\forall\, xi \in set\ x.\ xi \leq Max\ (set\ y)$
by (*metis* (*no-types*, *lifting*) *List.finite-set Max-ge-iff*
assms in-set-conv-nth length-0-conv less-eq-def set-empty)

lemma *le-not-less-replicate*:
$\forall\, x{\in}set\ xs.\ x \leq b \Longrightarrow \neg\ xs <_v replicate\ (length\ xs)\ b \Longrightarrow xs = replicate\ (length$
$xs)\ b$
by (*induct xs*) (*auto simp*: *less-Cons*)

lemma *le-replicateI*: $\forall\, x{\in}set\ xs.\ x \leq b \Longrightarrow xs \leq_v replicate\ (length\ xs)\ b$
by (*induct xs*) (*auto simp*: *le-Cons*)

lemma *le-take*:
assumes $x \leq_v y$ **and** $i \leq length\ x$ **shows** $take\ i\ x \leq_v take\ i\ y$
using *assms* **by** (*auto simp*: *less-eq-def*)

lemma *wf-less*:
$wf\ \{(x,\ y).\ x <_v y\}$
proof −
have *wf* (*measure sum-list*) **..**
moreover have $\{(x,\ y).\ x <_v y\} \subseteq measure\ sum\text{-}list$
by (*auto simp*: *less-vec-sum-list-less*)
ultimately show $wf\ \{(x,\ y).\ x <_v y\}$
by (*rule wf-subset*)
qed

1.3 Pointwise Subtraction

definition *vdiff* :: *nat list* ⇒ *nat list* ⇒ *nat list* (**infixl** $-_v$ *65*)
where
$w -_v v = map\ (\lambda i.\ w\ !\ i - v\ !\ i)\ [0\ ..{<}\ length\ w]$

lemma *vdiff-Nil* [*simp*]: $[] -_v [] = []$ **by** (*simp add*: *vdiff-def*)

```
lemma upt-Cons-conv:
  assumes j < n
  shows [j..<n] = j # [j+1..<n]
  by (simp add: assms upt-eq-Cons-conv)

lemma map-upt-Suc: map f [Suc m ..< Suc n] = map (f ∘ Suc) [m ..< n]
  by (fold list.map-comp [of f Suc [m ..< n]]) (simp add: map-Suc-upt)

lemma vdiff-Cons [simp]:
  (x # xs) −_v (y # ys) = (x − y) # (xs −_v ys)
 by (simp add: vdiff-def upt-Cons-conv [OF zero-less-Suc] map-upt-Suc del: upt-Suc)

lemma vdiff-alt-def:
  assumes length w = length v
  shows w −_v v = map (λ(x, y). x − y) (zip w v)
  using assms by (induct rule: list-induct2) simp-all

lemma vdiff-dotprod-distr:
  assumes length b = length w
    and v ≤_v w
  shows (w −_v v) · b = w · b − v · b
proof −
  have length v = length w and ∀ i<length w. v ! i ≤ w ! i
    using assms less-eq-def by auto
  then show ?thesis
    using ‹length b = length w›
  proof (induct v w arbitrary: b rule: list-induct2)
    case (Cons x xs y ys)
    then show ?case
      by (cases b) (auto simp: All-less-Suc2 diff-mult-distrib
          dotprod-commute dotprod-pointwise-le-right)
  qed simp
qed

lemma sum-list-vdiff-distr [simp]:
  assumes v ≤_v u
  shows sum-list (u −_v v) = sum-list u − sum-list v
  by (metis (no-types, lifting) assms diff-zero dotprod-1-right
      length-map length-replicate length-upt
      less-eq-def vdiff-def vdiff-dotprod-distr)

lemma vdiff-le:
  assumes v ≤_v w
    and length v = length x
  shows v −_v x ≤_v w
  using assms by (auto simp add: less-eq-def vdiff-def)

lemma mods-with-vec:
```

assumes $v <_v w$
  and $0 \notin set\ b$
  and $length\ b = length\ w$
  and $(v \cdot b)\ mod\ a = (w \cdot b)\ mod\ a$
shows $((w -_v v) \cdot b)\ mod\ a = 0$
proof −
  have $*$: $v \cdot b < w \cdot b$
    using *dotprod-less-left* **and** *assms* **by** *blast*
  have $v \le_v w$
    using *assms* **by** *auto*
  from *vdiff-dotprod-distr* [*OF assms*(*3*) *this*]
  have $((w -_v v) \cdot b)\ mod\ a = (w \cdot b - v \cdot b)\ mod\ a$
    by *simp*
  also have ... $= 0\ mod\ a$
    using *mods-with-nats* [*of* $v \cdot b\ w \cdot b\ 1\ a$, *OF* $*$] *assms* **by** *auto*
  finally show *?thesis* **by** *simp*
qed

lemma *mods-with-vec-2*:
  assumes $v <_v w$
    and $0 \notin set\ b$
    and $length\ b = length\ w$
    and $(b \cdot v)\ mod\ a = (b \cdot w)\ mod\ a$
  shows $(b \cdot (w -_v v))\ mod\ a = 0$
  by (*metis* (*no-types*, *lifting*) *assms diff-zero dotprod-commute*
    length-map length-upt less-eq-def order-vec.less-imp-le
    mods-with-vec vdiff-def)

1.4 The Lexicographic Order on Vectors

abbreviation *lex-less-than* (-/ $<_{lex}$ - [*51*, *51*] *50*)
  where
    $xs <_{lex} ys \equiv (xs, ys) \in lex\ less\text{-}than$

definition *rlex* (**infix** $<_{rlex}$ *50*)
  where
    $xs <_{rlex} ys \longleftrightarrow rev\ xs <_{lex} rev\ ys$

lemma *rev-le* [*simp*]:
  $rev\ xs \le_v rev\ ys \longleftrightarrow xs \le_v ys$
proof −
  { fix i **assume** i: $i < length\ ys$ **and** [*simp*]: $length\ xs = length\ ys$
    and $\forall i < length\ ys.\ rev\ xs\ !\ i \le rev\ ys\ !\ i$
    then have $rev\ xs\ !\ (length\ ys - i - 1) \le rev\ ys\ !\ (length\ ys - i - 1)$ **by** *auto*
    then have $xs\ !\ i \le ys\ !\ i$ **using** i **by** (*auto simp*: *rev-nth*) **}**
  then show *?thesis* **by** (*auto simp*: *less-eq-def rev-nth*)
qed

lemma *rev-less* [*simp*]:

rev xs $<_v$ *rev ys* $\longleftrightarrow$ *xs* $<_v$ *ys*
by (*simp add*: *less-def*)

lemma *less-imp-lex*:
assumes *xs* $<_v$ *ys* **shows** *xs* $<_{lex}$ *ys*
proof –
have *length ys* = *length xs* **using** *assms* **by** *auto*
then show *?thesis* **using** *assms*
by (*induct rule*: *list-induct2*) (*auto simp*: *less-Cons*)
qed

lemma *less-imp-rlex*:
assumes *xs* $<_v$ *ys* **shows** *xs* $<_{rlex}$ *ys*
using *assms* **and** *less-imp-lex* [*of rev xs rev ys*]
by (*simp add*: *rlex-def*)

lemma *lex-not-sym*:
assumes *xs* $<_{lex}$ *ys*
shows ¬ *ys* $<_{lex}$ *xs*
proof
assume *ys* $<_{lex}$ *xs*
then obtain *i* **where** *i* < *length xs* **and** *take i xs* = *take i ys*
and *ys* ! *i* < *xs* ! *i* **by** (*elim lex-take-index*) *auto*
moreover obtain *j* **where** *j* < *length xs* **and** *length ys* = *length xs* **and** *take j xs* = *take j ys*
and *xs* ! *j* < *ys* ! *j* **using** *assms* **by** (*elim lex-take-index*) *auto*
ultimately show *False* **by** (*metis le-antisym nat-less-le nat-neq-iff nth-take*)
qed

lemma *rlex-not-sym*:
assumes *xs* $<_{rlex}$ *ys*
shows ¬ *ys* $<_{rlex}$ *xs*
proof
assume *ass*: *ys* $<_{rlex}$ *xs*
then obtain *i* **where** *i* < *length xs* **and** *take i xs* = *take i ys*
and *ys* ! *i* > *xs* ! *i* **using** *assms lex-not-sym rlex-def* **by** *blast*
moreover obtain *j* **where** *j* < *length xs* **and** *length ys* = *length xs* **and** *take j xs* = *take j ys*
and *xs* ! *j* > *ys* ! *j* **using** *assms rlex-def ass lex-not-sym* **by** *blast*
ultimately show *False*
by (*metis leD nat-less-le nat-neq-iff nth-take*)
qed

lemma *lex-trans*:
assumes *x* $<_{lex}$ *y* **and** *y* $<_{lex}$ *z*
shows *x* $<_{lex}$ *z*
using *assms* **by** (*auto simp*: *antisym-def intro*: *transD* [*OF lex-transI*])

lemma *rlex-trans*:

assumes $x <_{rlex} y$ **and** $y <_{rlex} z$
shows $x <_{rlex} z$
using *assms lex-trans rlex-def* **by** *blast*

lemma *lex-append-rightD*:
assumes *xs* @ *us* $<_{lex}$ *ys* @ *vs* **and** *length xs* = *length ys*
and ¬ *xs* $<_{lex}$ *ys*
shows *ys* = *xs* ∧ *us* $<_{lex}$ *vs*
using *assms*(*2*,*1*,*3*)
by (*induct xs ys rule*: *list-induct2*) *auto*

lemma *rlex-Cons*:
x # *xs* $<_{rlex}$ *y* # *ys* ⟷ *xs* $<_{rlex}$ *ys* ∨ *ys* = *xs* ∧ *x* < *y* (**is** *?A* = *?B*)
by (*cases length ys* = *length xs*)
(*auto simp*: *rlex-def intro*: *lex-append-rightI lex-append-leftI dest*: *lex-append-rightD lex-lengthD*)

lemma *rlex-irrefl*:
¬ $x <_{rlex} x$
by (*induct x*) (*auto simp*: *rlex-def dest*: *lex-append-rightD*)

1.5 Code Equations

fun *exists2*
where
exists2 d P [] [] ⟷ *False*
| *exists2 d P* (*x*#*xs*) (*y*#*ys*) ⟷ *P x y* ∨ *exists2 d P xs ys*
| *exists2 d P* - - ⟷ *d*

lemma *not-le-code* [*code-unfold*]: ¬ *xs* $\le_v$ *ys* ⟷ *exists2 True* (>) *xs ys*
by (*induct True* (>) :: *nat* ⇒ *nat* ⇒ *bool xs ys rule*: *exists2.induct*) (*auto simp*: *le-Cons*)

end

2 Homogeneous Linear Diophantine Equations

theory *Linear-Diophantine-Equations*
imports *List-Vector*
begin

lemma *lcm-div-le*:
fixes *a* :: *nat*
shows *lcm a b div b* ≤ *a*
by (*metis div-by-0 div-le-dividend div-le-mono div-mult-self-is-m lcm-nat-def neq0-conv*)

lemma *lcm-div-le′*:

fixes *a* :: *nat*
shows *lcm a b div a* $\leq$ *b*
by (*metis lcm.commute lcm-div-le*)

lemma *lcm-div-gt-0*:
fixes *a* :: *nat*
assumes *a* > *0* **and** *b* > *0*
shows *lcm a b div a* > *0*
proof −
have *lcm a b* = (*a* ∗ *b*) *div* (*gcd a b*)
using *lcm-nat-def* **by** *blast*
moreover have . . . > *0*
using *assms*
by (*metis assms calculation lcm-pos-nat*)
ultimately show *?thesis*
using *assms*
by *simp* (*metis div-greater-zero-iff div-le-mono2 div-mult-self-is-m gcd-le2-nat not-gr0*)
qed

lemma *sum-list-list-update-Suc*:
assumes *i* < *length u*
shows *sum-list* (*u*[*i* := *Suc* (*u* ! *i*)]) = *Suc* (*sum-list u*)
using *assms*
proof (*induct u arbitrary*: *i*)
case (*Cons x xs*)
then show *?case* **by** (*simp-all split*: *nat.splits*)
qed (*simp*)

lemma *lessThan-conv*:
assumes *card A* = *n* **and** ∀ *x*∈*A*. *x* < *n*
shows *A* = {..<*n*}
using *assms* **by** (*simp add*: *card-subset-eq subsetI*)

Given a non-empty list *xs* of *n* natural numbers, either there is a value in *xs* that is *0* modulo *n*, or there are two values whose moduli coincide.

lemma *list-mod-cases*:
assumes *length xs* = *n* **and** *n* > *0*
shows (∃ *x*∈*set xs*. *x mod n* = *0*) ∨
(∃ *i*<*length xs*. ∃ *j*<*length xs*. *i* ≠ *j* ∧ (*xs* ! *i*) *mod n* = (*xs* ! *j*) *mod n*)
proof −
let *?f* = λ*x*. *x mod n* **and** *?X* = *set xs*
have ∗: ∀ *x* ∈ *?f* ‘ *?X*. *x* < *n* **using** ‹*n* > *0*› **by** *auto*
consider (*eq*) *card* (*?f* ‘ *?X*) = *card ?X* | (*less*) *card* (*?f* ‘ *?X*) < *card ?X*
using *antisym-conv2* **and** *card-image-le* **by** *blast*
then show *?thesis*

```
  proof (cases)
    case eq
    show ?thesis
    proof (cases distinct xs)
      assume distinct xs
      with eq have card (?f ` ?X) = n
        using ‹distinct xs› by (simp add: assms card-distinct distinct-card)
      from lessThan-conv [OF this *] and ‹n > 0›
      have ∃x∈set xs. x mod n = 0 by (metis imageE lessThan-iff)
      then show ?thesis ..
    next
      assume ¬ distinct xs
      then show ?thesis by (auto) (metis distinct-conv-nth)
    qed
  next
    case less
    from pigeonhole [OF this]
    show ?thesis by (auto simp: inj-on-def iff: in-set-conv-nth)
  qed
qed
```

Homogeneous linear Diophantine equations: $a_1 x_1 + \cdots + a_m x_m = b_1 y_1 + \cdots + b_n y_n$

```
locale hlde-ops =
  fixes a b :: nat list
begin

abbreviation m ≡ length a
abbreviation n ≡ length b
```

— The set of all solutions.

```
definition Solutions :: (nat list × nat list) set
  where
    Solutions = {(x, y). a • x = b • y ∧ length x = m ∧ length y = n}

lemma in-Solutions-iff:
  (x, y) ∈ Solutions ⟷ length x = m ∧ length y = n ∧ a • x = b • y
  by (auto simp: Solutions-def)
```

— The set of pointwise minimal solutions.

```
definition Minimal-Solutions :: (nat list × nat list) set
  where
    Minimal-Solutions = {(x, y) ∈ Solutions. nonzero x ∧
      ¬ (∃(u, v) ∈ Solutions. nonzero u ∧ u @ v <_v x @ y)}

definition dij :: nat ⇒ nat ⇒ nat
  where
    dij i j = lcm (a ! i) (b ! j) div (a ! i)
```

definition *eij* :: *nat* ⇒ *nat* ⇒ *nat*
where
eij i j = *lcm* (*a* ! *i*) (*b* ! *j*) *div* (*b* ! *j*)

definition *sij* :: *nat* ⇒ *nat* ⇒ (*nat list* × *nat list*)
where
sij i j = ((*zeroes m*)[*i* := *dij i j*], (*zeroes n*)[*j* := *eij i j*])

2.1 Further Constraints on Minimal Solutions

definition *Ej* :: *nat* ⇒ *nat list* ⇒ *nat set*
where
Ej j x = { *eij i j* − *1* | *i*. *i* < *length x* ∧ *x* ! *i* ≥ *dij i j* }

definition *Di* :: *nat* ⇒ *nat list* ⇒ *nat set*
where
Di i y = { *dij i j* − *1* | *j*. *j* < *length y* ∧ *y* ! *j* ≥ *eij i j* }

definition *Di′* :: *nat* ⇒ *nat list* ⇒ *nat set*
where
Di′ i y = { *dij i* (*j* + *length b* − *length y*) − *1* | *j*. *j* < *length y* ∧ *y* ! *j* ≥ *eij i* (*j* + *length b* − *length y*) }

lemma *Ej-take-subset*:
Ej j (*take k x*) ⊆ *Ej j x*
by (*auto simp*: *Ej-def*)

lemma *Di-take-subset*:
Di i (*take l y*) ⊆ *Di i y*
by (*auto simp*: *Di-def*)

lemma *Di′-drop-subset*:
Di′ i (*drop l y*) ⊆ *Di′ i y*
by (*auto simp*: *Di′-def*) (*metis add.assoc add.commute less-diff-conv*)

lemma *finite-Ej*:
finite (*Ej j x*)
by (*rule finite-subset* [*of* - (λ*i*. *eij i j* − *1*) ‘ {*0* ..< *length x*}]) (*auto simp*: *Ej-def*)

lemma *finite-Di*:
finite (*Di i y*)
by (*rule finite-subset* [*of* - (λ*j*. *dij i j* − *1*) ‘ {*0* ..< *length y*}]) (*auto simp*: *Di-def*)

lemma *finite-Di′*:
finite (*Di′ i y*)
by (*rule finite-subset* [*of* - (λ*j*. *dij i* (*j* + *length b* − *length y*) − *1*) ‘ {*0* ..< *length y*}])
(*auto simp*: *Di′-def*)

```
definition max-y :: nat list ⇒ nat ⇒ nat
  where
    max-y x j = (if j < n ∧ Ej j x ≠ {} then Min (Ej j x) else Max (set a))

definition max-x :: nat list ⇒ nat ⇒ nat
  where
    max-x y i = (if i < m ∧ Di i y ≠ {} then Min (Di i y) else Max (set b))

definition max-x′ :: nat list ⇒ nat ⇒ nat
  where
    max-x′ y i = (if i < m ∧ Di′ i y ≠ {} then Min (Di′ i y) else Max (set b))

lemma Min-Ej-le:
  assumes j < n
    and e ∈ Ej j x
    and length x ≤ m
  shows Min (Ej j x) ≤ Max (set a) (is ?m ≤ -)
proof −
  have ?m ∈ Ej j x
    using assms and finite-Ej and Min-in by blast
  then obtain i where
    i: ?m = eij i j − 1 i < length x x ! i ≥ dij i j
    by (auto simp: Ej-def)
  have lcm (a ! i) (b ! j) div b ! j ≤ a ! i by (rule lcm-div-le)
  then show ?thesis
    using i and assms
    by (auto simp: eij-def)
      (meson List.finite-set Max-ge diff-le-self le-trans less-le-trans nth-mem)
qed

lemma Min-Di-le:
  assumes i < m
    and e ∈ Di i y
    and length y ≤ n
  shows Min (Di i y) ≤ Max (set b) (is ?m ≤ -)
proof −
  have ?m ∈ Di i y
    using assms and finite-Di and Min-in by blast
  then obtain j where
    j: ?m = dij i j − 1 j < length y y ! j ≥ eij i j
    by (auto simp: Di-def)
  have lcm (a ! i) (b ! j) div a ! i ≤ b ! j by (rule lcm-div-le′)
  then show ?thesis
    using j and assms
    by (auto simp: dij-def)
      (meson List.finite-set Max-ge diff-le-self le-trans less-le-trans nth-mem)
qed

lemma Min-Di′-le:
```

assumes *i* < *m*
and *e* ∈ *Di′ i y*
and *length y* ≤ *n*
shows *Min* (*Di′ i y*) ≤ *Max* (*set b*) (**is** *?m* ≤ -)
proof −
have *?m* ∈ *Di′ i y*
using *assms* **and** *finite-Di′* **and** *Min-in* **by** *blast*
then obtain *j* **where**
j: *?m* = *dij i* (*j* + *length b* − *length y*) − *1 j* < *length y y* ! *j* ≥ *eij i* (*j* + *length b* − *length y*)
by (*auto simp*: *Di′-def*)
then have *j* + *length b* − *length y* < *length b* **using** *assms* **by** *auto*
moreover
have *lcm* (*a* ! *i*) (*b* ! (*j* + *length b* − *length y*)) *div a* ! *i* ≤ *b* ! (*j* + *length b* − *length y*) **by** (*rule lcm-div-le′*)
ultimately show *?thesis*
using *j* **and** *assms*
by (*auto simp*: *dij-def*)
(*meson List.finite-set Max-ge diff-le-self le-trans less-le-trans nth-mem*)
qed

lemma *max-y-le-take*:
assumes *length x* ≤ *m*
shows *max-y x j* ≤ *max-y* (*take k x*) *j*
using *assms* **and** *Min-Ej-le* **and** *Ej-take-subset* **and** *Min.subset-imp* [*OF* - - *finite-Ej*]
by (*auto simp*: *max-y-def*) *blast*

lemma *max-x-le-take*:
assumes *length y* ≤ *n*
shows *max-x y i* ≤ *max-x* (*take l y*) *i*
using *assms* **and** *Min-Di-le* **and** *Di-take-subset* **and** *Min.subset-imp* [*OF* - - *finite-Di*]
by (*auto simp*: *max-x-def*) *blast*

lemma *max-x′-le-drop*:
assumes *length y* ≤ *n*
shows *max-x′ y i* ≤ *max-x′* (*drop l y*) *i*
using *assms* **and** *Min-Di′-le* **and** *Di′-drop-subset* **and** *Min.subset-imp* [*OF* - - *finite-Di′*]
by (*auto simp*: *max-x′-def*) *blast*

end

abbreviation *Solutions* ≡ *hlde-ops.Solutions*
abbreviation *Minimal-Solutions* ≡ *hlde-ops.Minimal-Solutions*

abbreviation *dij* ≡ *hlde-ops.dij*
abbreviation *eij* ≡ *hlde-ops.eij*

abbreviation *sij* ≡ *hlde-ops.sij*

declare *hlde-ops.dij-def* [*code*]
declare *hlde-ops.eij-def* [*code*]
declare *hlde-ops.sij-def* [*code*]

lemma *Solutions-sym*: $(x, y) \in$ *Solutions a b* $\longleftrightarrow (y, x) \in$ *Solutions b a*
  by (*auto simp*: *hlde-ops.in-Solutions-iff*)

lemma *Minimal-Solutions-imp-Solutions*: $(x, y) \in$ *Minimal-Solutions a b* $\Longrightarrow (x, y) \in$ *Solutions a b*
  by (*auto simp*: *hlde-ops.Minimal-Solutions-def*)

lemma *Minimal-SolutionsI*:
  assumes $(x, y) \in$ *Solutions a b*
    and *nonzero x*
    and $\neg$ ($\exists(u, v) \in$ *Solutions a b*. *nonzero u* $\wedge$ u @ $v <_v x$ @ y)
  shows $(x, y) \in$ *Minimal-Solutions a b*
  using *assms* **by** (*auto simp*: *hlde-ops.Minimal-Solutions-def*)

lemma *minimize-nonzero-solution*:
  assumes $(x, y) \in$ *Solutions a b* **and** *nonzero x*
  obtains *u* **and** *v* **where** u @ $v \le_v x$ @ y **and** $(u, v) \in$ *Minimal-Solutions a b*
  using *assms*
proof (*induct x* @ *y arbitrary*: *x y thesis rule*: *wf-induct* [*OF wf-less*])
  case *1*
  then show *?case*
  proof (*cases* $(x, y) \in$ *Minimal-Solutions a b*)
    case *False*
    then obtain *u* **and** *v* **where** *nonzero u* **and** $(u, v) \in$ *Solutions a b* **and** *uv*: u @ $v <_v x$ @ y
      using *1*(*3*,*4*) **by** (*auto simp*: *hlde-ops.Minimal-Solutions-def*)
    with *1*(*1*) [*rule-format*, *of u* @ *v u v*] **obtain** u' **and** v' **where** uv': u' @ $v' \le_v u$ @ v
      and $(u', v') \in$ *Minimal-Solutions a b* **by** *blast*
    moreover have u' @ $v' \le_v x$ @ y **using** *uv* **and** uv' **by** *auto*
    ultimately show *?thesis* **by** (*intro 1*(*2*))
  qed *blast*
qed

lemma *Minimal-SolutionsI'*:
  assumes $(x, y) \in$ *Solutions a b*
    and *nonzero x*
    and $\neg$ ($\exists(u, v) \in$ *Minimal-Solutions a b*. u @ $v <_v x$ @ y)
  shows $(x, y) \in$ *Minimal-Solutions a b*
proof (*rule Minimal-SolutionsI* [*OF assms*(*1*,*2*)])
  show $\neg$ ($\exists(u, v) \in$ *Solutions a b*. *nonzero u* $\wedge$ u @ $v <_v x$ @ y)
  proof
    assume $\exists(u, v) \in$ *Solutions a b*. *nonzero u* $\wedge$ u @ $v <_v x$ @ y

then obtain *u* **and** *v* **where** $(u, v) \in$ *Solutions a b* **and** *nonzero u*
and *uv*: $u @ v <_v x @ y$ **by** *blast*
then obtain u' **and** v' **where** $(u', v') \in$ *Minimal-Solutions a b*
and uv': $u' @ v' \leq_v u @ v$ **by** (*blast elim*: *minimize-nonzero-solution*)
moreover have $u' @ v' <_v x @ y$ **using** *uv* **and** uv' **by** *auto*
ultimately show *False* **using** *assms* **by** *blast*
qed
qed

lemma *Minimal-Solutions-length*:
$(x, y) \in$ *Minimal-Solutions a b* $\Longrightarrow$ *length x* = *length a* $\wedge$ *length y* = *length b*
by (*auto simp*: *hlde-ops.Minimal-Solutions-def hlde-ops.in-Solutions-iff*)

lemma *Minimal-Solutions-gt0*:
$(x, y) \in$ *Minimal-Solutions a b* $\Longrightarrow$ *zeroes* (*length x*) $<_v x$
using *zero-less* **by** (*auto simp*: *hlde-ops.Minimal-Solutions-def*)

lemma *Minimal-Solutions-sym*:
assumes $0 \notin$ *set a* **and** $0 \notin$ *set b*
shows $(xs, ys) \in$ *Minimal-Solutions a b* $\longrightarrow$ $(ys, xs) \in$ *Minimal-Solutions b a*
using *assms*
by (*auto simp*: *hlde-ops.Minimal-Solutions-def hlde-ops.Solutions-def*
dest: *dotprod-eq-nonzero-iff dest*!: *less-append-swap* [*of - - ys xs*])

locale *hlde* = *hlde-ops* +
assumes *no0*: $0 \notin$ *set a* $0 \notin$ *set b*
begin

lemma *nonzero-Solutions-iff*:
assumes $(x, y) \in$ *Solutions*
shows *nonzero x* $\longleftrightarrow$ *nonzero y*
using *assms* **and** *no0* **by** (*auto simp*: *in-Solutions-iff dest*: *dotprod-eq-nonzero-iff*)

lemma *Minimal-Solutions-min*:
assumes $(x, y) \in$ *Minimal-Solutions*
and $u @ v <_v x @ y$
and $a \cdot u = b \cdot v$
and [*simp*]: *length u* = *m*
and *non0*: *nonzero* $(u @ v)$
shows *False*
proof −
have [*simp*]: *length v* = *n* **using** *assms* **by** (*force dest*: *less-appendD Minimal-Solutions-length*)
have $(u, v) \in$ *Solutions* **using** ‹$a \cdot u = b \cdot v$› **by** (*simp add*: *in-Solutions-iff*)
moreover from *nonzero-Solutions-iff* [*OF this*] **have** *nonzero u* **using** *non0* **by** *auto*
ultimately show *False* **using** *assms* **by** (*auto simp*: *hlde-ops.Minimal-Solutions-def*)
qed

lemma *Solutions-snd-not-0*:
  assumes $(x, y) \in$ *Solutions*
    and *nonzero x*
  shows *nonzero y*
  using *assms* **by** (*metis nonzero-Solutions-iff*)

end

2.2 Pointwise Restricting Solutions

Constructing the list of *u* vectors from Huet's proof [1], satisfying

- $\forall\, i<$*length u*. *u* ! $i \leq y$! *i* and
- $0 <$ *sum-list u* $\leq a_k$.

Given *y*, increment a "previous" *u* vector at first position starting from *i* where *u* is strictly smaller than *y*. If this is not possible, return *u* unchanged.

function *inc* :: *nat list* ⇒ *nat* ⇒ *nat list* ⇒ *nat list*
  where
    inc y i u =
      (*if i* < *length y then*
        if u ! *i* < *y* ! *i then u*[*i* := *u* ! *i* + *1*]
        else inc y (*Suc i*) *u*
      else u)
  by (*pat-completeness*) *auto*
termination *inc*
  by (*relation measure* (λ(*y*, *i*, *u*). *max* (*length y*) (*length u*) − *i*)) *auto*

declare *inc.simps* [*simp del*]

Starting from the 0-vector produce *u*s by iteratively incrementing with respect to *y*.

definition *huets-us* :: *nat list* ⇒ *nat* ⇒ *nat list* (**u** *1000*)
  where
    u *y i* = ((*inc y 0*) ^^ *Suc i*) (*zeroes* (*length y*))

lemma *huets-us-simps* [*simp*]:
  u *y 0* = *inc y 0* (*zeroes* (*length y*))
  u *y* (*Suc i*) = *inc y 0* (**u** *y i*)
  by (*auto simp*: *huets-us-def*)

lemma *length-inc* [*simp*]: *length* (*inc y i u*) = *length u*
  by (*induct y i u rule*: *inc.induct*) (*simp add*: *inc.simps*)

lemma *length-us* [*simp*]:
  length (**u** *y i*) = *length y*

by (*induct i*) (*simp-all*)

inc produces vectors that are pointwise smaller than *y*

lemma *inc-le*:
assumes *length u* = *length y* **and** *i* < *length y* **and** $u \le_v y$
shows *inc y i u* $\le_v$ *y*
using *assms* **by** (*induct y i u rule*: *inc.induct*)
(*auto simp*: *inc.simps nth-list-update less-eq-def*)

lemma *us-le*:
assumes *length y* > *0*
shows **u** *y i* $\le_v$ *y*
using *assms* **by** (*induct i*) (*auto simp*: *inc-le le-length*)

lemma *sum-list-inc-le*:
$u \le_v y \Longrightarrow$ *sum-list* (*inc y i u*) ≤ *sum-list y*
by (*induct y i u rule*: *inc.induct*)
(*auto simp*: *inc.simps intro*: *le-sum-list-mono*)

lemma *sum-list-inc-gt0*:
assumes *sum-list u* > *0* **and** *length y* = *length u*
shows *sum-list* (*inc y i u*) > *0*
using *assms*
proof (*induct y i u rule*: *inc.induct*)
case (*1 y i u*)
then show *?case*
by (*auto simp add*: *inc.simps*)
(*meson Suc-neq-Zero gr-zeroI set-update-memI sum-list-eq-0-iff*)
qed

lemma *sum-list-inc-gt0'*:
assumes *length u* = *length y* **and** *i* < *length y* **and** *y* ! *i* > *0* **and** *j* ≤ *i*
shows *sum-list* (*inc y j u*) > *0*
using *assms*
proof (*induct y j u rule*: *inc.induct*)
case (*1 y i u*)
then show *?case*
by (*auto simp*: *inc.simps* [*of y i*] *sum-list-update*)
(*metis elem-le-sum-list le-antisym le-zero-eq neq0-conv not-less-eq-eq sum-list-inc-gt0*)
qed

lemma *sum-list-us-gt0*:
assumes *sum-list y* ≠ *0*
shows *0* < *sum-list* (**u** *y i*)
using *assms* **by** (*induct i*) (*auto simp*: *in-set-conv-nth sum-list-inc-gt0' sum-list-inc-gt0*)

lemma *sum-list-inc-le'*:
assumes *length u* = *length y*
shows *sum-list* (*inc y i u*) ≤ *sum-list u* + *1*

using *assms*
by (*induct y i u rule*: *inc.induct*) (*auto simp*: *inc.simps sum-list-update*)

lemma *sum-list-us-le*:
sum-list (**u** *y i*) $\leq$ *i* + *1*
proof (*induct i*)
case *0*
then show *?case*
by (*auto simp*: *sum-list-update*)
(*metis Suc-eq-plus1 in-set-replicate length-replicate sum-list-eq-0-iff sum-list-inc-le'*)
next
case (*Suc i*)
then show *?case*
by *auto* (*metis Suc-le-mono add.commute le-trans length-us plus-1-eq-Suc sum-list-inc-le'*)
qed

lemma *sum-list-us-bounded*:
assumes *i* < *k*
shows *sum-list* (**u** *y i*) $\leq$ *k*
using *assms* **and** *sum-list-us-le* [*of y i*] **by** *force*

lemma *sum-list-inc-eq-sum-list-Suc*:
assumes *length u* = *length y* **and** *i* < *length y*
and ∃*j*≥*i*. *j* < *length y* ∧ *u* ! *j* < *y* ! *j*
shows *sum-list* (*inc y i u*) = *Suc* (*sum-list u*)
using *assms*
by (*induct y i u rule*: *inc.induct*)
(*metis inc.simps Suc-eq-plus1 Suc-leI antisym-conv2 leD sum-list-list-update-Suc*)

lemma *sum-list-us-eq*:
assumes *i* < *sum-list y*
shows *sum-list* (**u** *y i*) = *i* + *1*
using *assms*
proof (*induct i*)
case (*Suc i*)
then show *?case*
by (*auto*)
(*metis* (*no-types*, *lifting*) *Suc-eq-plus1 gr-implies-not0 length-pos-if-in-set*
length-us less-Suc-eq-le less-imp-le-nat antisym-conv2 not-less-eq-eq
sum-list-eq-0-iff sum-list-inc-eq-sum-list-Suc sum-list-less-diff-Ex us-le)
qed (*metis Suc-eq-plus1 Suc-leI antisym-conv gr-implies-not0 sum-list-us-gt0 sum-list-us-le*)

lemma *inc-ge*: *length u* = *length y* ⟹ *u* $\leq_v$ *inc y i u*
by (*induct y i u rule*: *inc.induct*) (*auto simp*: *inc.simps nth-list-update less-eq-def*)

lemma *us-le-mono*:
assumes *i* < *j*
shows **u** *y i* $\leq_v$ **u** *y j*
using *assms*

```
proof (induct j − i arbitrary: j i)
  case (Suc n)
  then show ?case
   by (simp add: Suc.prems inc-ge order.strict-implies-order order-vec.lift-Suc-mono-le)
qed simp

lemma us-mono:
  assumes i < j and j < sum-list y
  shows u y i <v u y j
proof −
  let ?u = u y i and ?v = u y j
  have ?u ≤v ?v
    using us-le-mono [OF ‹i < j›] by simp
  moreover have sum-list ?u < sum-list ?v
    using assms by (auto simp: sum-list-us-eq)
  ultimately show ?thesis by (intro le-sum-list-less) (auto simp: less-eq-def)
qed

context hlde
begin

lemma max-coeff-bound-right:
  assumes (xs, ys) ∈ Minimal-Solutions
  shows ∀ x ∈ set xs. x ≤ maxne0 ys b (is ∀ x∈set xs. x ≤ ?m)
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain k
    where k-def: k < length xs ∧ ¬ (xs ! k ≤ ?m)
    by (metis in-set-conv-nth)
  have sol: (xs, ys) ∈ Solutions
    using assms Minimal-Solutions-def by auto
  then have len: m = length xs by (simp add: in-Solutions-iff)
  have max-suml: ?m ∗ sum-list ys ≥ b • ys
    using maxne0-times-sum-list-gt-dotprod sol by (auto simp: in-Solutions-iff)
  then have is-sol: b • ys = a • xs
    using sol by (auto simp: in-Solutions-iff)
  then have a-ge-ak: a • xs ≥ a ! k ∗ xs ! k
    using dotprod-pointwise-le k-def len by auto
  then have ak-gt-max: a ! k ∗ xs ! k > a ! k ∗ ?m
    using no0 in-set-conv-nth k-def len by fastforce
  then have sl-ys-g-ak: sum-list ys > a ! k
    by (metis a-ge-ak is-sol less-le-trans max-suml
        mult.commute mult-le-mono1 not-le)
  define Seq where
    Seq-def: Seq = map (u ys) [0 ..< a ! k]
  have ak-n0: a ! k ≠ 0
    using ‹a ! k ∗ ?m < a ! k ∗ xs ! k› by auto
  have zeroes (length ys) <v ys
   by (intro zero-less) (metis gr-implies-not0 nonzero-iff sl-ys-g-ak sum-list-eq-0-iff)
```

then have *length Seq* $>$ *0*
  using *ak-n0 Seq-def* **by** *auto*
have *u-in-nton*: $\forall u \in$ *set Seq. length u* $=$ *length ys*
  by (*simp add*: *Seq-def*)
have *prop-3*: $\forall u \in$ *set Seq. u* $\leq_v$ *ys*
proof −
  have *length ys* $>$ *0*
    using *sl-ys-g-ak* **by** *auto*
  then show *?thesis*
    using *us-le* [*of ys*] *less-eq-def Seq-def* **by** (*simp*)
qed
have *prop-4-1*: $\forall u \in$ *set Seq. sum-list u* $>$ *0*
  by (*metis Seq-def sl-ys-g-ak gr-implies-not-zero imageE*
    set-map sum-list-us-gt0)
have *prop-4-2*: $\forall u \in$ *set Seq. sum-list u* $\leq$ *a* ! *k*
  by (*simp add*: *Seq-def sum-list-us-bounded*)
have *prop-5*: $\exists u$. *length u* $=$ *length ys* $\wedge$ *u* $\leq_v$ *ys* $\wedge$ *sum-list u* $>$ *0* $\wedge$ *sum-list u*
$\leq$ *a* ! *k*
  using ‹*0* $<$ *length Seq*› *nth-mem prop-3 prop-4-1 prop-4-2 u-in-nton* **by** *blast*
define *Us* **where**
  Us $=$ {*u. length u* $=$ *length ys* $\wedge$ *u* $\leq_v$ *ys* $\wedge$ *sum-list u* $>$ *0* $\wedge$ *sum-list u* $\leq$ *a* !
k}
have $\exists u \in$ *Us. b* • *u mod a* ! *k* $=$ *0*
proof (*rule ccontr*)
  assume *neg-th*: ¬ *?thesis*
  define *Seq-p* **where**
    Seq-p $=$ *map* (*dotprod b*) *Seq*
  have *length Seq* $=$ *a* ! *k*
    by (*simp add*: *Seq-def*)
  then consider (*eq-0*) ($\exists$ *x*∈*set Seq-p. x mod* (*a* ! *k*) $=$ *0*) |
    (*not-0*) ($\exists$ *i*<*length Seq-p.* $\exists$ *j*<*length Seq-p. i* $\neq$ *j* $\wedge$
        (*Seq-p* ! *i*) *mod* (*a*!*k*) $=$ (*Seq-p* ! *j*) *mod* (*a*!*k*))
    using *list-mod-cases*[*of Seq-p*] *Seq-p-def ak-n0* **by** *auto force*
  then show *False*
  proof (*cases*)
    case *eq-0*
    have $\exists u \in$ *set Seq. b* • *u mod a* ! *k* $=$ *0*
      using *Seq-p-def eq-0* **by** *auto*
    then show *False*
      by (*metis* (*mono-tags*, *lifting*) *Us-def mem-Collect-eq*
        neg-th prop-3 prop-4-1 prop-4-2 u-in-nton)
  next
    case *not-0*
    obtain *i* **and** *j* **where**
      i-j: *i*<*length Seq-p j*<*length Seq-p* *i* $\neq$ *j*
       Seq-p ! *i mod a* ! *k* $=$ *Seq-p* ! *j mod a* ! *k*
      using *not-0* **by** *blast*
    define *v* **where**
      v-def: *v* $=$ *Seq*!*i*

```
define w where
  w-def: w = Seq!j
have mod-eq: b · v mod a!k = b · w mod a!k
  using Seq-p-def i-j w-def v-def i-j  by auto
have v <_v w ∨ w <_v v
  using ‹i ≠ j› and i-j
proof (cases i < j)
  case True
  then show ?thesis
   using Seq-p-def sl-ys-g-ak i-j(2) local.Seq-def us-mono v-def w-def by auto
next
  case False
  then show ?thesis
    using Seq-p-def sl-ys-g-ak ‹i ≠ j› i-j(1) local.Seq-def us-mono v-def w-def
by auto
qed
then show False
proof
  assume ass: v <_v w
  define u where
    u-def: u = w -_v v
  have w ≤_v ys
    using Seq-p-def w-def i-j(2) prop-3 by force
  then have prop-3: less-eq u ys
    using vdiff-le ass less-eq-def order-vec.less-imp-le u-def by auto
  have prop-4-1: sum-list u > 0
    using le-sum-list-mono [of v w] ass u-def sum-list-vdiff-distr [of v w]
    by (simp add: less-vec-sum-list-less)
  have prop-4-2: sum-list u ≤ a ! k
  proof -
    have u ≤_v w using u-def
      using ass less-eq-def order-vec.less-imp-le vdiff-le by auto
    then show ?thesis
      by (metis Seq-p-def i-j(2) length-map le-sum-list-mono
          less-le-trans not-le nth-mem prop-4-2 w-def)
  qed
  have b · u mod a ! k = 0
   by (metis (mono-tags, lifting) in-Solutions-iff ‹w ≤_v ys› u-def ass no0(2)
        less-eq-def mem-Collect-eq mod-eq mods-with-vec-2 prod.simps(2) sol)
  then show False using neg-th
    by (metis (mono-tags, lifting) Us-def less-eq-def mem-Collect-eq
        prop-3 prop-4-1 prop-4-2)
next
  assume ass: w <_v v
  define u where
    u-def: u = v -_v w
  have v ≤_v ys
    using Seq-p-def v-def i-j(1) prop-3 by force
  then have prop-3: u ≤_v ys
```

using *vdiff-le ass less-eq-def order-vec.less-imp-le u-def* **by** *auto*
have *prop-4-1*: *sum-list u* $>$ *0*
using *le-sum-list-mono* [*of w v*] *sum-list-vdiff-distr* [*of w v*]
‹$u \equiv v -_v w$› *ass less-vec-sum-list-less* **by** *auto*
have *prop-4-2*: *sum-list u* $\leq$ *a*!*k*
proof −
have $u \leq_v v$ **using** *u-def*
using *ass less-eq-def order-vec.less-imp-le vdiff-le* **by** *auto*
then show *?thesis*
by (*metis Seq-p-def i-j*(*1*) *le-neq-implies-less length-map less-imp-le-nat less-le-trans nth-mem prop-4-2 le-sum-list-mono v-def*)
qed
have *b* · *u mod a* ! *k* = *0*
by (*metis* (*mono-tags*, *lifting*) *in-Solutions-iff* ‹$v \leq_v ys$› *u-def ass no0*(*2*) *less-eq-def mem-Collect-eq mod-eq mods-with-vec-2 prod.simps*(*2*) *sol*)
then show *False*
by (*metis* (*mono-tags*, *lifting*) *neg-th Us-def less-eq-def mem-Collect-eq prop-3 prop-4-1 prop-4-2*)
qed
qed
qed
then obtain *u* **where**
u3-4: $u \leq_v ys$ *sum-list u* $>$ *0 sum-list u* $\leq$ *a* ! *k* *b* · *u mod* (*a* ! *k*) = *0*
length u = *length ys*
unfolding *Us-def* **by** *auto*
have *u-b-len*: *length u* = *n*
using *less-eq-def u3-4 in-Solutions-iff sol* **by** *simp*
have *b* · *u* $\leq$ *maxne0 u b* ∗ *sum-list u*
by (*simp add*: *maxne0-times-sum-list-gt-dotprod u-b-len*)
also have ... $\leq$ *?m* ∗ *a* ! *k*
by (*intro mult-le-mono*) (*simp-all add*: *u3-4 maxne0-mono*)
also have ... $<$ *a* ! *k* ∗ *xs* ! *k*
using *ak-gt-max* **by** *auto*
then obtain *zk* **where**
zk: *b* · *u* = *zk* ∗ *a* ! *k*
using *u3-4*(*4*) **by** *auto*
have *length xs* $>$ *k*
by (*simp add*: *k-def*)
have *zk* $\neq$ *0*
proof −
have ∃ *e* ∈ *set u*. *e* $\neq$ *0*
using *u3-4*
by (*metis neq0-conv sum-list-eq-0-iff*)
then have *b* · *u* $>$ *0*
using *assms no0 u3-4*
unfolding *dotprod-gt0-iff*[*OF u-b-len* [*symmetric*]]
by (*fastforce simp add*: *in-set-conv-nth u-b-len*)
then have *a* ! *k* $>$ *0*
using ‹*a* ! *k* $\neq$ *0*› **by** *blast*

```
    then show ?thesis
      using ‹0 < b • u› zk by auto
  qed
  define z where
    z-def: z = (zeroes (length xs))[k := zk]
  then have zk-zk: z ! k = zk
    by (auto simp add: ‹k < length xs›)
  have length z = length xs
    using assms z-def ‹k < length xs› by auto
  then have bu-eq-akzk: b • u = a ! k * z ! k
    by (simp add: ‹b • u = zk * a ! k› zk-zk)
  then have z!k < xs!k
    using ak-gt-max  calculation by auto
  then have z-less-xs: z <v xs
    by (auto simp add: z-def) (metis ‹k < length xs› le0 le-list-update less-def
        less-imp-le order-vec.dual-order.antisym nat-neq-iff z-def zk-zk)
  then have z @ u <v xs @ ys
    by (intro less-append) (auto simp add: u3-4(1) z-less-xs)
  moreover have (z, u) ∈ Solutions
    by (auto simp add: bu-eq-akzk in-Solutions-iff z-def u-b-len ‹k < length xs› len)
  moreover have nonzero z
    using ‹length z = length xs› and ‹zk ≠ 0› and k-def and zk-zk by (auto simp:
nonzero-iff)
  ultimately show False using assms by (auto simp: Minimal-Solutions-def)
qed
```

Proof of Lemma 1 of Huet's paper.

```
lemma max-coeff-bound:
  assumes (xs, ys) ∈ Minimal-Solutions
  shows (∀ x ∈ set xs. x ≤ maxne0 ys b) ∧ (∀ y ∈ set ys. y ≤ maxne0 xs a)
proof −
  interpret ba: hlde b a by (standard) (auto simp: no0)
  show ?thesis
    using assms and Minimal-Solutions-sym [OF no0, of xs ys]
    by (auto simp: max-coeff-bound-right ba.max-coeff-bound-right)
qed

lemma max-coeff-bound':
  assumes (x, y) ∈ Minimal-Solutions
  shows ∀ i<length x. x ! i ≤ Max (set b) and ∀ j<length y. y ! j ≤ Max (set a)
  using max-coeff-bound [OF assms] and maxne0-le-Max
  by auto (metis le-eq-less-or-eq less-le-trans nth-mem)+

lemma Minimal-Solutions-alt-def:
  Minimal-Solutions = {(x, y)∈Solutions.
    (x, y) ≠ (zeroes m, zeroes n) ∧
    x ≤v replicate m (Max (set b)) ∧
    y ≤v replicate n (Max (set a)) ∧
    ¬ (∃ (u, v)∈Solutions. nonzero u ∧ u @ v <v x @ y)}
```

by (*auto simp*: *not-nonzero-iff Minimal-Solutions-imp-Solutions less-eq-def Minimal-Solutions-length max-coeff-bound′*
intro!: *Minimal-SolutionsI′ dest*: *Minimal-Solutions-gt0*)
(*auto simp*: *Minimal-Solutions-def nonzero-Solutions-iff not-nonzero-iff*)

2.3 Special Solutions

definition *Special-Solutions* :: (*nat list* × *nat list*) *set*
where
Special-Solutions = {*sij i j* | *i j*. *i* < *m* ∧ *j* < *n*}

lemma *dij-neq-0*:
assumes *i* < *m*
and *j* < *n*
shows *dij i j* ≠ *0*
proof −
have *a* ! *i* > *0* **and** *b* ! *j* > *0*
using *assms* **and** *no0* **by** (*simp-all add*: *in-set-conv-nth*)
then have *dij i j* > *0*
using *lcm-div-gt-0* [*of a* ! *i b* ! *j*] **by** (*simp add*: *dij-def*)
then show *?thesis* **by** *simp*
qed

lemma *eij-neq-0*:
assumes *i* < *m*
and *j* < *n*
shows *eij i j* ≠ *0*
proof −
have *a* ! *i* > *0* **and** *b* ! *j* > *0*
using *assms* **and** *no0* **by** (*simp-all add*: *in-set-conv-nth*)
then have *eij i j* > *0*
using *lcm-div-gt-0*[*of b* ! *j a* ! *i*] **by** (*simp add*: *eij-def lcm.commute*)
then show *?thesis*
by *simp*
qed

lemma *Special-Solutions-in-Solutions*:
x ∈ *Special-Solutions* ⟹ *x* ∈ *Solutions*
by (*auto simp*: *in-Solutions-iff Special-Solutions-def sij-def dij-def eij-def*)

lemma *Special-Solutions-in-Minimal-Solutions*:
assumes (*x*, *y*) ∈ *Special-Solutions*
shows (*x*, *y*) ∈ *Minimal-Solutions*
proof (*intro Minimal-SolutionsI′*)
show (*x*, *y*) ∈ *Solutions* **by** (*fact Special-Solutions-in-Solutions* [*OF assms*])
then have [*simp*]: *length x* = *m length y* = *n* **by** (*auto simp*: *in-Solutions-iff*)
show *nonzero x* **using** *assms* **and** *dij-neq-0*
by (*auto simp*: *Special-Solutions-def sij-def nonzero-iff*)
(*metis length-replicate set-update-memI*)

```
  show ¬ (∃(u, v)∈Minimal-Solutions. u @ v <v x @ y)
  proof
    assume ∃(u, v)∈Minimal-Solutions. u @ v <v x @ y
    then obtain u and v where uv: (u, v) ∈ Minimal-Solutions and u @ v <v x @ y
      and [simp]: length u = m length v = n
      and nonzero u by (auto simp: Minimal-Solutions-def in-Solutions-iff)
     then consider u <v x and v ≤v y | v <v y and u ≤v x by (auto elim: less-append-cases)
    then show False
    proof (cases)
      case 1
      then obtain i and j where ij: i < m j < n
        and less-dij: u ! i < dij i j
        and u ≤v (zeroes m)[i := dij i j]
        and v ≤v (zeroes n)[j := eij i j]
        using assms by (auto simp: Special-Solutions-def sij-def unit-less)
      then have u: u = (zeroes m)[i := u ! i] and v: v = (zeroes n)[j := v ! j]
        by (auto simp: less-eq-def list-eq-iff-nth-eq)
          (metis le-zero-eq length-list-update length-replicate rep-upd-unit)+
      then have u ! i > 0 using ‹nonzero u› and ij
        by (metis gr-implies-not0 neq0-conv unit-less zero-less)

      define c where c = a ! i * u ! i
      then have ac: a ! i dvd c by simp

     have a · u = b · v using uv by (auto simp: Minimal-Solutions-def in-Solutions-iff)
      then have c = b ! j * v ! j
         using ij unfolding c-def by (subst (asm) u, subst (asm)v, subst u, subst v) auto
      then have bc: b ! j dvd c by simp

      have a ! i * u ! i < a ! i * dij i j
        using less-dij and no0 and ij by (auto simp: in-set-conv-nth)
      then have c < lcm (a ! i) (b ! j) by (auto simp: dij-def c-def)
      moreover have lcm (a ! i) (b ! j) dvd c by (simp add: ac bc)
       moreover have c > 0 using ‹u ! i > 0› and no0 and ij by (auto simp: c-def in-set-conv-nth)
      ultimately show False using ac and bc by (auto dest: nat-dvd-not-less)
    next
      case 2
      then obtain i and j where ij: i < m j < n
        and less-dij: v ! j < eij i j
        and u ≤v (zeroes m)[i := dij i j]
        and v ≤v (zeroes n)[j := eij i j]
        using assms by (auto simp: Special-Solutions-def sij-def unit-less)
      then have u: u = (zeroes m)[i := u ! i] and v: v = (zeroes n)[j := v ! j]
        by (auto simp: less-eq-def list-eq-iff-nth-eq)
          (metis le-zero-eq length-list-update length-replicate rep-upd-unit)+
```

```
      moreover have nonzero v
        using ‹nonzero u› and ‹(u, v) ∈ Minimal-Solutions›
          and Minimal-Solutions-imp-Solutions Solutions-snd-not-0 by blast
      ultimately have v ! j > 0 using ij
        by (metis gr-implies-not0 neq0-conv unit-less zero-less)

      define c where c = b ! j ∗ v ! j
      then have bc: b ! j dvd c by simp

    have a · u = b · v using uv by (auto simp: Minimal-Solutions-def in-Solutions-iff)
      then have c = a ! i ∗ u ! i
        using ij unfolding c-def by (subst (asm) u, subst (asm)v, subst u, subst
v) auto
      then have ac: a ! i dvd c by simp

      have b ! j ∗ v ! j < b ! j ∗ eij i j
        using less-dij and no0 and ij by (auto simp: in-set-conv-nth)
      then have c < lcm (a ! i) (b ! j) by (auto simp: eij-def c-def)
      moreover have lcm (a ! i) (b ! j) dvd c by (simp add: ac bc)
       moreover have c > 0 using ‹v ! j > 0› and no0 and ij by (auto simp:
c-def in-set-conv-nth)
      ultimately show False using ac and bc by (auto dest: nat-dvd-not-less)
    qed
  qed
qed


lemma non-special-solution-non-minimal:
  assumes (x, y) ∈ Solutions − Special-Solutions
    and ij: i < m j < n
    and x ! i ≥ dij i j and y ! j ≥ eij i j
  shows (x, y) ∉ Minimal-Solutions
proof
  assume min: (x, y) ∈ Minimal-Solutions
  moreover have sij i j ∈ Solutions
   using ij by (intro Special-Solutions-in-Solutions) (auto simp: Special-Solutions-def)
  moreover have (case sij i j of (u, v) ⇒ u @ v) <_v x @ y
    using assms and min
    apply (cases sij i j)
    apply (auto simp: sij-def Special-Solutions-def)
   by (metis List-Vector.le0 Minimal-Solutions-length le-append le-list-update less-append
order-vec.dual-order.strict-iff-order same-append-eq)
  moreover have (case sij i j of (u, v) ⇒ nonzero u)
    apply (auto simp: sij-def)
    by (metis dij-neq-0 ij length-replicate nonzero-iff set-update-memI)
  ultimately show False
    by (auto simp: Minimal-Solutions-def)
qed
```

2.4 Huet's conditions

definition *cond-A xs ys* ⟷ (∀ *x*∈*set xs*. *x* ≤ *maxne0 ys b*)

definition *cond-B x* ⟷
(∀ *k*≤*m*. *take k a* • *take k x* ≤ *b* • *map* (*max-y* (*take k x*)) [*0* ..< *n*])

definition *boundr x y* ⟷ (∀ *j*<*n*. *y* ! *j* ≤ *max-y x j*)

definition *cond-D x y* ⟷ (∀ *l*≤*n*. *take l b* • *take l y* ≤ *a* • *x*)

2.5 New conditions: facilitating generation of candidates from right to left

definition *subdprodr y* ⟷
(∀ *l*≤*n*. *take l b* • *take l y* ≤ *a* • *map* (*max-x* (*take l y*)) [*0* ..< *m*])

definition *subdprodl x y* ⟷ (∀ *k*≤*m*. *take k a* • *take k x* ≤ *b* • *y*)

definition *boundl x y* ⟷ (∀ *i*<*m*. *x* ! *i* ≤ *max-x y i*)

lemma *boundr*:
  assumes *min*: (*x*, *y*) ∈ *Minimal-Solutions*
    and (*x*, *y*) ∉ *Special-Solutions*
  shows *boundr x y*
proof (*unfold boundr-def*, *intro allI impI*)
  fix *j*
  assume *ass*: *j* < *n*
  have *ln*: *m* = *length x* ∧ *n* = *length y*
    using *assms Minimal-Solutions-def in-Solutions-iff min* **by** *auto*
  have *is-sol*: (*x*, *y*) ∈ *Solutions*
    using *assms Minimal-Solutions-def min* **by** *auto*
  have *j-less-l*: *j* < *n*
    using *assms ass le-less-trans* **by** *linarith*
  consider (*notemp*) *Ej j x* ≠ {} | (*empty*) *Ej j x* = {}
    by *blast*
  then show *y* ! *j* ≤ *max-y x j*
  proof (*cases*)
    case *notemp*
    have *max-y-def*: *max-y x j* = *Min* (*Ej j x*)
      using *j-less-l max-y-def notemp* **by** *auto*
    have *fin-e*: *finite* (*Ej j x*)
      using *finite-Ej* [*of j x*] **by** *auto*
    have *e-def'*: ∀ *e* ∈ *Ej j x*. (∃ *i*<*length x*. *x* ! *i* ≥ *dij i j* ∧ *eij i j* − *1* = *e*)
      using *Ej-def* [*of j x*] **by** *auto*

```
    then have ∃i<length x. x ! i ≥ dij i j ∧ eij i j − 1 = Min (Ej j x)
      using notemp Min-in e-def′ fin-e by blast
    then obtain i where
      i: i < length x x ! i ≥ dij i j eij i j − 1 = Min (Ej j x)
      by blast
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      with non-special-solution-non-minimal [of x y i j]
        and i and ln and assms and is-sol and j-less-l
      have case sij i j of (u, v) ⇒ u @ v ≤v x @ y
        by (force simp: max-y-def)
      then have cs:case sij i j of (u, v) ⇒ u @ v <v x @ y
      using assms by(auto simp: Special-Solutions-def) (metis append-eq-append-conv
          i(1) j-less-l length-list-update length-replicate sij-def
          order-vec.le-neq-trans ln prod.sel(1))
      then obtain u v where
        u-v: sij i j = (u, v) u @ v <v x @ y
        by blast
      have dij-gt0: dij i j > 0
        using assms(1) assms(2) dij-neq-0 i(1) j-less-l ln by auto
      then have not-0-u: nonzero u
      proof (unfold nonzero-iff)
        have i < length (zeroes m) by (simp add: i(1) ln)
        then show ∃i∈set u. i ≠ 0
            by (metis (no-types) Pair-inject dij-gt0 set-update-memI sij-def u-v(1)
neq0-conv)
      qed
      then have sij i j ∈ Solutions
        by (metis (mono-tags, lifting) Special-Solutions-def i(1)
          Special-Solutions-in-Solutions j-less-l ln mem-Collect-eq u-v(1))
      then show False
        using assms cs u-v not-0-u Minimal-Solutions-def min by auto
    qed
  next
    case empty
    have ∀y∈set y. y ≤ Max (set a)
      using assms and max-coeff-bound and maxne0-le-Max
      using le-trans by blast
    then  show ?thesis
      using empty j-less-l ln max-y-def by auto
  qed
qed

lemma boundl:
  assumes min: (x, y) ∈ Minimal-Solutions
    and (x, y) ∉ Special-Solutions
  shows boundl x y
proof (unfold boundl-def, intro allI impI)
```

```
  fix i
  assume ass: i < m
  have ln: n = length y ∧ m = length x
    using assms Minimal-Solutions-def in-Solutions-iff min by auto
  have is-sol: (x, y) ∈ Solutions
    using assms Minimal-Solutions-def min by auto
  have i-less-l: i < m
    using assms ass le-less-trans by linarith
  consider (notemp) Di i y ≠ {}  | (empty)  Di i y = {}
    by blast
  then show x ! i ≤ max-x y i
  proof (cases)
    case notemp
    have max-x-def: max-x y i =  Min (Di i y)
      using i-less-l max-x-def notemp by auto
    have fin-e: finite (Di i y)
      using finite-Di [of i y] by auto
    have e-def': ∀ e ∈ Di i y. (∃ j<length y. y ! j ≥ eij i j ∧ dij i j − 1 = e)
      using Di-def [of i y] by auto
    then have ∃ j<length y. y ! j ≥ eij i j ∧ dij i j − 1 = Min (Di i y)
      using notemp Min-in e-def' fin-e by blast
    then obtain j where
      j: j < length y y ! j ≥ eij i j dij i j − 1 = Min (Di i y)
      by blast
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      with non-special-solution-non-minimal [of x y i j]
        and j and ln and assms and is-sol and i-less-l
      have case sij i j of (u, v) ⇒ u @ v ≤_v x @ y
        by (force simp: max-x-def)
      then have cs: case sij i j of (u, v) ⇒ u @ v <_v x @ y
      using assms by(auto simp: Special-Solutions-def) (metis append-eq-append-conv
          j(1) i-less-l length-list-update length-replicate sij-def
          order-vec.le-neq-trans ln prod.sel(1))
      then obtain u v where
        u-v: sij i j = (u, v) u @ v <_v x @ y
        by blast
      have dij-gt0: dij i j > 0
        using assms(1) assms(2) dij-neq-0 j(1) i-less-l ln by auto
      then have not-0-u: nonzero u
      proof (unfold nonzero-iff)
        have i < length (zeroes m)
          using ass by simp
        then show ∃ i∈set u. i ≠ 0
            by (metis (no-types) Pair-inject dij-gt0 set-update-memI sij-def u-v(1)
neq0-conv)
      qed
      then have sij i j ∈ Solutions
```

```
        by (metis (mono-tags, lifting) Special-Solutions-def j(1)
            Special-Solutions-in-Solutions i-less-l ln mem-Collect-eq u-v(1))
      then show False
        using assms cs u-v not-0-u Minimal-Solutions-def min by auto
    qed
  next
    case empty
    have ∀ x∈set x. x ≤ Max (set b)
      using assms and max-coeff-bound and maxne0-le-Max
      using le-trans by blast
    then  show ?thesis
      using empty i-less-l ln max-x-def by auto
  qed
qed

lemma Solution-imp-cond-D:
  assumes (x, y) ∈ Solutions
  shows cond-D x y
  using assms and dotprod-le-take by (auto simp: cond-D-def in-Solutions-iff)

lemma Solution-imp-subdprodl:
  assumes (x, y) ∈ Solutions
  shows subdprodl x y
  using assms and dotprod-le-take
  by (auto simp: subdprodl-def in-Solutions-iff) metis

theorem conds:
  assumes min: (x, y) ∈ Minimal-Solutions
  shows cond-A: cond-A x y
    and cond-B: (x, y) ∉ Special-Solutions ⟹ cond-B x
    and (x, y) ∉ Special-Solutions ⟹ boundr x y
    and cond-D: cond-D x y
    and subdprodr: (x, y) ∉ Special-Solutions ⟹ subdprodr y
    and subdprodl: subdprodl x y
proof –
  have sol: a • x = b • y and ln: m = length x ∧ n = length y
    using min by (auto simp: Minimal-Solutions-def in-Solutions-iff)
  then have ∀ i<m. x ! i ≤ maxne0 y b
    by (metis min max-coeff-bound-right nth-mem)
  then show cond-A x y
    using min and le-less-trans by (auto simp: cond-A-def max-coeff-bound)
  show (x, y) ∉ Special-Solutions ⟹ cond-B x
  proof (unfold cond-B-def, intro allI impI)
    fix k assume non-spec: (x, y) ∉ Special-Solutions and k: k ≤ m
    from k have take k a • take k x ≤ a • x
      using dotprod-le-take ln by blast
    also have ... = b • y by fact
    also have map-b-dot-p: ... ≤ b • map (max-y x) [0..<n] (is - ≤ - b • ?nt)
      using non-spec and less-eq-def and ln and boundr and min
```

```
      by (fastforce intro!: dotprod-le-right simp: boundr-def)
    also have ... ≤ b · map (max-y (take k x)) [0..<n] (is - ≤ - · ?t)
    proof –
      have ∀ j<n. ?nt!j ≤ ?t!j
        using min and ln and max-y-le-take and k by auto
      then have ?nt ≤v ?t
        using less-eq-def by auto
      then show ?thesis
        by (simp add: dotprod-le-right)
    qed
    finally show take k a · take k x ≤ b · map (max-y (take k x)) [0..<n]
      by (auto simp: cond-B-def)
  qed

  show (x, y) ∉ Special-Solutions ⟹ subdprodr y
  proof (unfold subdprodr-def, intro allI impI)
    fix l assume non-spec: (x, y) ∉ Special-Solutions and l: l ≤ n
    from l have take l b · take l y ≤ b · y
      using dotprod-le-take ln by blast
    also have ... = a · x by (simp add: sol)
    also have map-b-dot-p: ... ≤ a · map (max-x y) [0..<m] (is - ≤ - a · ?nt)
      using non-spec and less-eq-def and ln and boundl and min
      by (fastforce intro!: dotprod-le-right simp: boundl-def)
    also have ... ≤ a · map (max-x (take l y)) [0..<m] (is - ≤ - · ?t)
    proof –
      have ∀ i<m. ?nt ! i ≤ ?t ! i
        using min and ln and max-x-le-take and l by auto
      then have ?nt ≤v ?t
        using less-eq-def by auto
      then show ?thesis
        by (simp add: dotprod-le-right)
    qed
    finally show take l b · take l y ≤ a · map (max-x (take l y)) [0..<m]
      by (auto simp: cond-B-def)
  qed

  show (x, y) ∉ Special-Solutions ⟹ boundr x y
    using boundr [of x y] and min by blast

  show cond-D x y
    using ln and dotprod-le-take and sol by (auto simp: cond-D-def)

  show subdprodl x y
    using ln and dotprod-le-take and sol by (force simp: subdprodl-def)
qed

lemma le-imp-Ej-subset:
  assumes u ≤v x
  shows Ej j u ⊆ Ej j x
```

using *assms* **and** *le-trans* **by** (*force simp*: *Ej-def less-eq-def dij-def eij-def*)

lemma *le-imp-max-y-ge*:
assumes $u \leq_v x$
and *length* $x \leq m$
shows *max-y* u $j \geq$ *max-y* x j
using *assms* **and** *le-imp-Ej-subset* **and** *Min-Ej-le* [*of* j, *OF* - - *assms*(*2*)]
by (*metis Min.subset-imp Min-in emptyE finite-Ej max-y-def order-refl subsetCE*)

lemma *le-imp-Di-subset*:
assumes $v \leq_v y$
shows *Di* i $v \subseteq$ *Di* i y
using *assms* **and** *le-trans* **by** (*force simp*: *Di-def less-eq-def dij-def eij-def*)

lemma *le-imp-max-x-ge*:
assumes $v \leq_v y$
and *length* $y \leq n$
shows *max-x* v $i \geq$ *max-x* y i
using *assms* **and** *le-imp-Di-subset* **and** *Min-Di-le* [*of* i, *OF* - - *assms*(*2*)]
by (*metis Min.subset-imp Min-in emptyE finite-Di max-x-def order-refl subsetCE*)

end

end

theory *Sorted-Wrt*
imports *Main*
begin

lemma *sorted-wrt-filter*:
sorted-wrt P xs $\Longrightarrow$ *sorted-wrt P* (*filter Q xs*)
by (*induct xs*) (*auto*)

lemma *sorted-wrt-map-mono*:
assumes *sorted-wrt Q xs*
and $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow P\ (f\ x)\ (f\ y)$
shows *sorted-wrt P* (*map f xs*)
using *assms* **by** (*induct xs*) (*auto*)

lemma *sorted-wrt-concat-map-map*:
assumes *sorted-wrt Q xs*
and *sorted-wrt Q ys*
and $\bigwedge a\ x\ y.\ Q\ x\ y \Longrightarrow P\ (f\ x\ a)\ (f\ y\ a)$
and $\bigwedge x\ y\ u\ v.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow Q\ u\ v \Longrightarrow P\ (f\ x\ u)\ (f\ y\ v)$
shows *sorted-wrt P* $[f\ x\ y\ .\ y \leftarrow ys,\ x \leftarrow xs]$
using *assms* **by** (*induct ys*)
(*auto simp*: *sorted-wrt-append intro*: *sorted-wrt-map-mono* [*of Q*])

lemma *sorted-wrt-concat-map*:
assumes *sorted-wrt P* (*map h xs*)
and $\bigwedge x.\ x \in set\ xs \Longrightarrow sorted\text{-}wrt\ P\ (map\ h\ (f\ x))$
and $\bigwedge x\ y\ u\ v.\ P\ (h\ x)\ (h\ y) \Longrightarrow x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow u \in set\ (f\ x) \Longrightarrow v \in set\ (f\ y) \Longrightarrow P\ (h\ u)\ (h\ v)$
shows *sorted-wrt P* (*concat* (*map* (*map h* ∘ *f*) *xs*))
using *assms* **by** (*induct xs*) (*auto simp*: *sorted-wrt-append*)

lemma *sorted-wrt-map-distr*:
assumes *sorted-wrt* ($\lambda x\ y.\ P\ x\ y$) (*map f xs*)
shows *sorted-wrt* ($\lambda x\ y.\ P\ (f\ x)\ (f\ y)$) *xs*
using *assms*
by (*induct xs*) (*auto*)

lemma *sorted-wrt-tl*:
$xs \neq [] \Longrightarrow sorted\text{-}wrt\ P\ xs \Longrightarrow sorted\text{-}wrt\ P\ (tl\ xs)$
by (*cases xs*) (*auto*)

end

3 Minimization

theory *Minimize-Wrt*
imports *Sorted-Wrt*
begin

fun *minimize-wrt*
where
minimize-wrt P [] = []
| *minimize-wrt P* (*x* # *xs*) = *x* # *filter* (*P x*) (*minimize-wrt P xs*)

lemma *minimize-wrt-subset*: *set* (*minimize-wrt P xs*) ⊆ *set xs*
by (*induct xs*) *auto*

lemmas *minimize-wrtD* = *minimize-wrt-subset* [*THEN subsetD*]

lemma *sorted-wrt-minimize-wrt*:
sorted-wrt P (*minimize-wrt P xs*)
by (*induct xs*) (*auto simp*: *sorted-wrt-filter*)

lemma *sorted-wrt-imp-sorted-wrt-minimize-wrt*:
sorted-wrt Q xs ⟹ *sorted-wrt Q* (*minimize-wrt P xs*)
by (*induct xs*) (*auto simp*: *sorted-wrt-filter dest*: *minimize-wrtD*)

lemma *in-minimize-wrt-False*:
assumes $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow \neg\ Q\ y\ x$
and *sorted-wrt Q xs*
and *x* ∈ *set* (*minimize-wrt P xs*)
and ¬ *P y x* **and** *Q y x* **and** *y* ∈ *set xs* **and** *y* ≠ *x*

shows *False*
using *assms* **by** (*induct xs*) (*auto dest*: *minimize-wrtD*)

lemma *in-minimize-wrtI*:
assumes $x \in$ *set xs*
and $\forall$ *y*$\in$*set xs*. *P y x*
shows $x \in$ *set* (*minimize-wrt P xs*)
using *assms* **by** (*induct xs*) *auto*

lemma *minimize-wrt-eq*:
assumes *distinct xs* **and** $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow P\ x\ y \longleftrightarrow Q\ x\ y$
$\vee\ x = y$
shows *minimize-wrt P xs* = *minimize-wrt Q xs*
using *assms* **by** (*induct xs*) (*auto*, *metis contra-subsetD filter-cong minimize-wrt-subset*)

lemma *minimize-wrt-ni*:
assumes $x \in$ *set xs*
and $x \notin$ *set* (*minimize-wrt Q xs*)
shows $\exists\, y \in set\ xs.\ (\neg\ Q\ y\ x) \wedge x \neq y$
using *assms* **by** (*induct xs*) (*auto*)

lemma *in-minimize-wrtD*:
assumes $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow \neg\ Q\ y\ x$
and *sorted-wrt Q xs*
and $x \in$ *set* (*minimize-wrt P xs*)
and $\bigwedge x\ y.\ \neg\ P\ x\ y \Longrightarrow Q\ x\ y$
and $\bigwedge x.\ P\ x\ x$
shows $x \in set\ xs \wedge (\forall\, y {\in} set\ xs.\ P\ y\ x)$
using *in-minimize-wrt-False* [*OF assms*(*1*−*3*)] **and** *minimize-wrt-subset* [*of P*
xs] **and** *assms*(*3*−*5*)
by *blast*

lemma *in-minimize-wrt-iff*:
assumes $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow \neg\ Q\ y\ x$
and *sorted-wrt Q xs*
and $\bigwedge x\ y.\ \neg\ P\ x\ y \Longrightarrow Q\ x\ y$
and $\bigwedge x.\ P\ x\ x$
shows $x \in$ *set* (*minimize-wrt P xs*) $\longleftrightarrow x \in set\ xs \wedge (\forall\, y {\in} set\ xs.\ P\ y\ x)$
using *assms* **and** *in-minimize-wrtD* [*of Q xs x P*, *OF assms*(*1*,*2*) - *assms*(*3*,*4*)]
by (*blast intro*: *in-minimize-wrtI*)

lemma *set-minimize-wrt*:
assumes $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow \neg\ Q\ y\ x$
and *sorted-wrt Q xs*
and $\bigwedge x\ y.\ \neg\ P\ x\ y \Longrightarrow Q\ x\ y$
and $\bigwedge x.\ P\ x\ x$
shows *set* (*minimize-wrt P xs*) = $\{x \in set\ xs.\ \forall\, y {\in} set\ xs.\ P\ y\ x\}$
by (*auto simp*: *in-minimize-wrt-iff* [*OF assms*])

lemma *minimize-wrt-append*:
assumes $\forall x \in set\ xs.\ \forall y \in set\ (xs\ @\ ys).\ P\ y\ x$
shows $minimize\text{-}wrt\ P\ (xs\ @\ ys) = xs\ @\ filter\ (\lambda y.\ \forall x \in set\ xs.\ P\ x\ y)\ (minimize\text{-}wrt\ P\ ys)$
using *assms* **by** (*induct xs*) (*auto intro*: *filter-cong*)

end

theory *Simple-Algorithm*
imports
Linear-Diophantine-Equations
Minimize-Wrt
begin

lemma *concat-map-nth0*: $xs \neq [] \Longrightarrow f\ (xs\ !\ 0) \neq [] \Longrightarrow concat\ (map\ f\ xs)\ !\ 0 = f\ (xs\ !\ 0)\ !\ 0$
by (*induct xs*) (*auto simp*: *nth-append*)

3.1 Reverse-Lexicographic Enumeration of Potential Minimal Solutions

fun *rlex2* :: $(nat\ list \times nat\ list) \Rightarrow (nat\ list \times nat\ list) \Rightarrow bool$ (**infix** $<_{rlex2}$ *50*)
where
$(xs,\ ys) <_{rlex2} (us,\ vs) \longleftrightarrow xs\ @\ ys <_{rlex}\ us\ @\ vs$

lemma *rlex2-irrefl*:
$\neg\ x <_{rlex2} x$
by (*cases x*) (*auto simp*: *rlex-irrefl*)

lemma *rlex2-not-sym*: $x <_{rlex2} y \Longrightarrow \neg\ y <_{rlex2} x$
using *rlex-not-sym* **by** (*cases x*; *cases y*; *simp*)

lemma *less-imp-rlex2*: $\neg\ (case\ x\ of\ (x,\ y) \Rightarrow \lambda(u,\ v).\ \neg\ x\ @\ y <_v u\ @\ v)\ y \Longrightarrow x <_{rlex2} y$
using *less-imp-rlex* **by** (*cases x*; *cases y*; *auto*)

Generate all lists (of natural numbers) of length n with elements bounded by B.

fun *gen* :: $nat \Rightarrow nat \Rightarrow nat\ list\ list$
where
$gen\ B\ 0 = [[]]$
$|\ gen\ B\ (Suc\ n) = [x\#xs\ .\ xs \leftarrow gen\ B\ n,\ x \leftarrow [0\ ..<\ B + 1]]$

definition $generate\ A\ B\ m\ n = tl\ [(x,\ y)\ .\ y \leftarrow gen\ B\ n,\ x \leftarrow gen\ A\ m]$

definition $check\ a\ b = filter\ (\lambda(x,\ y).\ a \cdot x = b \cdot y)$

definition *minimize* = *minimize-wrt* (λ(*x*, *y*) (*u*, *v*). ¬ *x* @ *y* $<_v$ *u* @ *v*)

definition *solutions a b* =
(*let A* = *Max* (*set b*); *B* = *Max* (*set a*); *m* = *length a*; *n* = *length b*
in minimize (*check a b* (*generate A B m n*)))

lemma *set-gen*: *set* (*gen B n*) = {*xs*. *length xs* = *n* ∧ (∀ *i*<*n*. *xs* ! *i* ≤ *B*)} (**is** - = *?A n*)
proof (*induct n*)
  case [*simp*]: (*Suc n*)
  { fix *xs* **assume** *xs* ∈ *?A* (*Suc n*)
    then have *xs* ∈ *set* (*gen B* (*Suc n*))
      by (*cases xs*) (*force simp*: *All-less-Suc2*)+ **}**
  then show *?case* **by** (*auto simp*: *less-Suc-eq-0-disj*)
qed *simp*

abbreviation *gen2 A B m n* ≡ [(*x*, *y*) . *y* ← *gen B n*, *x* ← *gen A m*]

lemma *sorted-wrt-gen*:
  sorted-wrt ($<_{rlex}$) (*gen B n*)
by (*induction n*)
   (*auto simp*: *rlex-Cons sorted-wrt-append sorted-wrt-map rlex-irrefl*
    intro!: *sorted-wrt-concat-map* [**where** *h* = *id*, *simplified*])

lemma *sorted-wrt-gen2*: *sorted-wrt* ($<_{rlex2}$) (*gen2 A B m n*)
  by (*intro sorted-wrt-concat-map-map* [**where** *Q* = ($<_{rlex}$)] *sorted-wrt-gen*)
   (*auto simp*: *set-gen rlex-def intro*: *lex-append-leftI lex-append-rightI*)

lemma *gen-ne* [*simp*]: *gen B n* ≠ [] **by** (*induct n*) *auto*

lemma *gen2-ne*: *gen2 A B m n* ≠ [] **by** *auto*

lemma *sorted-wrt-generate*: *sorted-wrt* ($<_{rlex2}$) (*generate A B m n*)
  by (*auto simp*: *generate-def intro*: *sorted-wrt-tl sorted-wrt-gen2*)

abbreviation *check-generate a b* ≡ *check a b* (*generate* (*Max* (*set b*)) (*Max* (*set a*)) (*length a*) (*length b*))

lemma *sorted-wrt-check-generate*: *sorted-wrt* ($<_{rlex2}$) (*check-generate a b*)
  by (*auto simp*: *check-def intro*: *sorted-wrt-filter sorted-wrt-generate*)

lemma *in-tl-gen2*: *x* ∈ *set* (*tl* (*gen2 A B m n*)) ⟹ *x* ∈ *set* (*gen2 A B m n*)
  by (*rule list.set-sel*) *simp*

lemma *gen-nth0* [*simp*]: *gen B n* ! *0* = *zeroes n*
  by (*induct n*) (*auto simp*: *nth-append concat-map-nth0*)

lemma *gen2-nth0* [*simp*]:
  gen2 A B m n ! *0* = (*zeroes m*, *zeroes n*)

by (*auto simp*: *concat-map-nth0*)

lemma *set-gen2*:
  set (*gen2 A B m n*) = {(*x*, *y*). *length x* = *m* ∧ *length y* = *n* ∧ (∀ *i*<*m*. *x* ! *i* ≤ *A*) ∧ (∀ *j*<*n*. *y* ! *j* ≤ *B*)}
  by (*auto simp*: *set-gen*)

lemma *gen2-unique*:
  assumes *i* < *j*
    and *j* < *length* (*gen2 A B m n*)
  shows *gen2 A B m n* ! *i* ≠ *gen2 A B m n* ! *j*
  using *sorted-wrt-nth-less* [*OF sorted-wrt-gen2 assms*]
  by (*auto simp*: *rlex2-irrefl*)

lemma *zeroes-ni-tl-gen2*:
  (*zeroes m*, *zeroes n*) ∉ *set* (*tl* (*gen2 A B m n*))
proof −
  have *gen2 A B m n* ! *0* = (*zeroes m*, *zeroes n*) **by** (*auto simp*: *generate-def*)
  with *gen2-unique*[*of 0 - A m B n*] **show** *?thesis*
    by (*metis* (*no-types*, *lifting*) *Suc-eq-plus1 in-set-conv-nth length-tl less-diff-conv nth-tl zero-less-Suc*)
qed

lemma *set-generate*:
  set (*generate A B m n*) = {(*x*, *y*). (*x*, *y*) ≠ (*zeroes m*, *zeroes n*) ∧ (*x*, *y*) ∈ *set* (*gen2 A B m n*)}
proof
  show *set* (*generate A B m n*)
    ⊆ {(*x*, *y*).(*x*, *y*) ≠ (*zeroes m*, *zeroes n*) ∧ (*x*, *y*) ∈ *set* (*gen2 A B m n*)}
     using *in-tl-gen2* **and** *mem-Collect-eq* **and** *zeroes-ni-tl-gen2* **by** (*auto simp*: *generate-def*)
next
  have (*zeroes m*, *zeroes n*) = *hd* (*gen2 A B m n*)
    by (*simp add*: *hd-conv-nth*)
  moreover have *set* (*gen2 A B m n*) = *set* (*generate A B m n*) ∪ {(*zeroes m*, *zeroes n*)}
   by (*metis Un-empty-right generate-def Un-insert-right gen2-ne calculation list.exhaust-sel list.simps*(*15*))
  ultimately show {(*x*, *y*). (*x*, *y*) ≠ (*zeroes m*, *zeroes n*) ∧ (*x*, *y*) ∈ *set* (*gen2 A B m n*)}
    ⊆ *set* (*generate A B m n*)
    by *blast*
qed

lemma *set-check-generate*:
  set (*check-generate a b*) = {(*x*, *y*).
    (*x*, *y*) ≠ (*zeroes* (*length a*), *zeroes* (*length b*)) ∧
    length x = *length a* ∧ *length y* = *length b* ∧ *a* · *x* = *b* · *y* ∧
    (∀ *i*<*length a*. *x* ! *i* ≤ *Max* (*set b*)) ∧ (∀ *j*<*length b*. *y* ! *j* ≤ *Max* (*set a*))}

unfolding *check-def* **and** *set-filter* **and** *set-generate* **and** *set-gen2* **by** *auto*

lemma *set-minimize-check-generate*:
set (*minimize* (*check-generate a b*)) =
{(*x*, *y*)∈*set* (*check-generate a b*). ¬ (∃ (*u*, *v*)∈*set* (*check-generate a b*). *u* @ *v* $<_v$ *x* @ *y*)}
unfolding *minimize-def*
by (*subst set-minimize-wrt* [*OF* - *sorted-wrt-check-generate*]) (*auto dest*: *rlex-not-sym less-imp-rlex*)

lemma *set-solutions-iff*:
set (*solutions a b*) =
{(*x*, *y*) ∈ *set* (*check-generate a b*). ¬ (∃ (*u*, *v*)∈*set* (*check-generate a b*). *u* @ *v* $<_v$ *x* @ *y*)}
by (*auto simp*: *solutions-def set-minimize-check-generate*)

3.1.1 Completeness: every minimal solution is generated by *solutions*

lemma (**in** *hlde*) *solutions-complete*:
Minimal-Solutions ⊆ *set* (*solutions a b*)
proof (*rule subrelI*)
let *?A* = *Max* (*set b*) **and** *?B* = *Max* (*set a*)
fix *x y* **assume** *min*: (*x*, *y*) ∈ *Minimal-Solutions*
then have (*x*, *y*) ∈ *set* (*check a b* (*generate ?A ?B m n*))
by (*auto simp*: *Minimal-Solutions-alt-def set-check-generate less-eq-def in-Solutions-iff*)
moreover have ∀ (*u*, *v*) ∈ *set* (*check a b* (*generate ?A ?B m n*)). ¬ *u* @ *v* $<_v$ *x* @ *y*
using *min* **and** *no0*
by (*auto simp*: *check-def set-generate neq-0-iff′ set-gen nonzero-iff dest*!: *Minimal-Solutions-min*)
ultimately show (*x*, *y*) ∈ *set* (*solutions a b*) **by** (*auto simp*: *set-solutions-iff*)
qed

3.1.2 Correctness: *solutions* generates only minimal solutions.

lemma (**in** *hlde*) *solutions-sound*:
set (*solutions a b*) ⊆ *Minimal-Solutions*
proof (*rule subrelI*)
fix *x y* **assume** *sol*: (*x*, *y*) ∈ *set* (*solutions a b*)
show (*x*, *y*) ∈ *Minimal-Solutions*
proof (*rule Minimal-SolutionsI′*)
show ∗: (*x*, *y*) ∈ *Solutions*
using *sol* **by** (*auto simp*: *set-solutions-iff in-Solutions-iff check-def set-generate set-gen*)
show *nonzero x*
using *sol* **and** *nonzero-iff* **and** *replicate-eqI* **and** *nonzero-Solutions-iff* [*OF* ∗]
by (*fastforce simp*: *solutions-def minimize-def check-def set-generate set-gen dest*!: *minimize-wrtD*)

show ¬ (∃(u, v)∈Minimal-Solutions. u @ v $<_v$ x @ y)
proof
have min-cg: (x, y) ∈ set (minimize (check-generate a b))
using sol **by** (auto simp: solutions-def)
note ∗ = in-minimize-wrt-False [OF - sorted-wrt-check-generate min-cg [unfolded minimize-def]]

assume ∃(u, v)∈Minimal-Solutions. u @ v $<_v$ x @ y
then obtain u **and** v **where** (u, v) ∈ Minimal-Solutions **and** less: u @ v $<_v$ x @ y **by** blast
then have (u, v) ∈ set (solutions a b) **by** (auto intro: solutions-complete [THEN subsetD])
then have (u, v) ∈ set (check-generate a b)
by (auto simp: solutions-def minimize-def dest: minimize-wrtD)
from ∗ [OF - - - this] **and** less **show** False
using less-imp-rlex **and** rlex-not-sym **by** force
qed
qed
qed

lemma (**in** hlde) set-solutions [simp]: set (solutions a b) = Minimal-Solutions
using solutions-sound **and** solutions-complete **by** blast

end

4 Computing Minimal Complete Sets of Solutions

theory Algorithm
imports Simple-Algorithm
begin

lemma all-Suc-le-conv: (∀ i≤Suc n. P i) ⟷ P 0 ∧ (∀ i≤n. P (Suc i))
by (metis less-Suc-eq-0-disj nat-less-le order-refl)

lemma concat-map-filter-filter:
assumes ⋀x. x ∈ set xs ⟹ ¬ Q x ⟹ filter P (f x) = []
shows concat (map (filter P ∘ f) (filter Q xs)) = concat (map (filter P ∘ f) xs)
using assms **by** (induct xs) simp-all

lemma filter-pairs-conj:
filter (λ(x, y). P x y ∧ Q y) xs = filter (λ(x, y). P x y) (filter (Q ∘ snd) xs)
by (induct xs) auto

lemma concat-map-filter:
concat (map f (filter P xs)) = concat (map (λx. if P x then f x else []) xs)

by (*induct xs*) *simp-all*

fun *alls*
  where
    alls B [] = [([], *0*)]
  | *alls B* (*a* # *as*) = [(*x* # *xs*, *s* + *a* ∗ *x*). (*xs*, *s*) ← *alls B as*, *x* ← [*0* ..< *B* + *1*]]

lemma *alls-ne* [*simp*]:
  alls B as ≠ []
  by (*induct as*)
   (*auto*, *metis* (*no-types*, *lifting*) *append-is-Nil-conv case-prod-conv list.set-intros*(*1*)
     neq-Nil-conv old.prod.exhaust)

lemma *set-alls*: *set* (*alls B a*) =
  {(*x*, *s*). *length x* = *length a* ∧ (∀ *i*<*length a*. *x* ! *i* ≤ *B*) ∧ *s* = *a* · *x*}
    (**is** *?L a* = *?R a*)
proof
  show *?L a* ⊆ *?R a* **by** (*induct a*) (*auto simp*: *nth-Cons split*: *nat.splits*)
next
  show *?R a* ⊆ *?L a*
  proof (*induct a*)
    case (*Cons a as*)
    show *?case*
    proof
      fix *xs′* **assume** *xs′* ∈ *?R* (*a* # *as*)
      then obtain *x* **and** *xs* **where** [*simp*]: *xs′* = (*x* # *xs*, (*a* # *as*) · (*x* # *xs*))
        and *length as* = *length xs*
        and *B*: *x* ≤ *B* ∀ *i*<*length as*. *xs* ! *i* ≤ *B*
        by (*cases xs′*, *case-tac a*) (*auto simp*: *All-less-Suc2*)
      then have (*xs*, *as* · *xs*) ∈ *?L as* **using** *Cons* **by** *auto*
      then show *xs′* ∈ *?L* (*a* # *as*)
        using *B*
        apply *auto*
        apply (*rule bexI* [*of* - (*xs*, *as* · *xs*)])
         apply *auto*
        done
    qed
  qed *auto*
qed

lemma *alls-nth0* [*simp*]: *alls A as* ! *0* = (*zeroes* (*length as*), *0*)
  by (*induct as*) (*auto simp*: *nth-append concat-map-nth0*)

lemma *alls-Cons-tl-conv*: *alls A as* = (*zeroes* (*length as*), *0*) # *tl* (*alls A as*)
  by (*rule nth-equalityI*) (*auto simp*: *nth-Cons nth-tl split*: *nat.splits*)

lemma *sorted-wrt-alls*:
  sorted-wrt ($<_{rlex}$) (*map fst* (*alls B xs*))
  by (*induct xs*) (*auto simp*: *map-concat rlex-Cons sorted-wrt-append*

intro!: *sorted-wrt-concat-map sorted-wrt-map-mono* [*of* (<)])

definition *alls2 A B a b* = [(*xs*, *ys*). *ys* ← *alls B b*, *xs* ← *alls A a*]

lemma *alls2-ne* [*simp*]:
alls2 A B a b ≠ []
by (*auto simp*: *alls2-def*) (*metis alls-ne list.set-intros*(*1*) *neq-Nil-conv surj-pair*)

lemma *set-alls2*:
set (*alls2 A B a b*) = {((*x*, *s*), (*y*, *t*)). *length x* = *length a* ∧ *length y* = *length b* ∧
(∀ *i*<*length a*. *x* ! *i* ≤ *A*) ∧ (∀ *j*<*length b*. *y* ! *j* ≤ *B*) ∧ *s* = *a* · *x* ∧ *t* = *b* · *y*}
by (*auto simp*: *alls2-def set-alls*)

lemma *alls2-nth0* [*simp*]: *alls2 A B as bs* ! *0* = ((*zeroes* (*length as*), *0*), (*zeroes* (*length bs*), *0*))
by (*auto simp*: *alls2-def concat-map-nth0*)

lemma *alls2-Cons-tl-conv*: *alls2 A B as bs* =
((*zeroes* (*length as*), *0*), (*zeroes* (*length bs*), *0*)) # *tl* (*alls2 A B as bs*)
apply (*rule nth-equalityI*)
apply (*auto simp*: *alls2-def nth-Cons nth-tl length-concat concat-map-nth0 split*: *nat.splits*)
apply (*cases alls B bs*; *simp*)
done

abbreviation *gen2*
where
gen2 A B a b ≡ *map* (λ(*x*, *y*). (*fst x*, *fst y*)) (*alls2 A B a b*)

lemma *sorted-wrt-gen2*:
sorted-wrt ($<_{rlex2}$) (*gen2 A B a b*)
apply (*rule sorted-wrt-map-mono* [*of* λ(*x*, *y*) (*u*, *v*). (*fst x*, *fst y*) $<_{rlex2}$ (*fst u*, *fst v*)])
apply (*auto simp*: *alls2-def map-concat*)
apply (*fold rlex2.simps*)
apply (*rule sorted-wrt-concat-map-map*)
apply (*rule sorted-wrt-map-distr*, *rule sorted-wrt-alls*)
apply (*rule sorted-wrt-map-distr*, *rule sorted-wrt-alls*)
apply (*auto simp*: *rlex-def set-alls intro*: *lex-append-leftI lex-append-rightI*)
done

definition *generate'*
where
generate' A B a b = *tl* (*map* (λ(*x*, *y*). (*fst x*, *fst y*)) (*alls2 A B a b*))

lemma *sorted-wrt-generate'*:
sorted-wrt ($<_{rlex2}$) (*generate' A B a b*)
by (*auto simp*: *generate'-def sorted-wrt-gen2 sorted-wrt-tl*)

lemma *gen2-nth0* [*simp*]:
  gen2 A B a b ! 0 = (zeroes (length a), zeroes (length b))
  by *auto*

lemma *gen2-ne* [*simp*, *intro*]: *gen2 m n b c ≠ []* **by** *auto*

lemma *in-generate′*: *x ∈ set (generate′ m n c b) ⟹ x ∈ set (gen2 m n c b)*
  unfolding *generate′-def* **by** (*rule list.set-sel*) *simp*

definition *cond-cons P = (λ(ys, s). case ys of [] ⇒ True | ys ⇒ P ys s)*

lemma *cond-cons-simp* [*simp*]:
  cond-cons P ([], s) = True
  cond-cons P (x # xs, s) = P (x # xs) s
  by (*auto simp*: *cond-cons-def*)

fun *suffs*
  where
    suffs P as (xs, s) ⟷
      length xs = length as ∧
      s = as • xs ∧
      (∀ i≤length xs. cond-cons P (drop i xs, drop i as • drop i xs))
declare *suffs.simps* [*simp del*]

lemma *suffs-Nil* [*simp*]: *suffs P [] ([], s) ⟷ s = 0*
  by (*auto simp*: *suffs.simps*)

lemma *suffs-Cons*:
  suffs P (a # as) (x # xs, s) ⟷
    s = a ∗ x + as • xs ∧ cond-cons P (x # xs, s) ∧ suffs P as (xs, as • xs)
  apply (*auto simp*: *suffs.simps cond-cons-def split*: *list.splits*)
    apply *force*
   apply (*metis Suc-le-mono drop-Suc-Cons*)
  by (*metis One-nat-def Suc-le-mono Suc-pred dotprod-Cons drop-Cons′ le-0-eq not-le-imp-less*)

4.1 The Algorithm

fun *maxne0-impl*
  where
    maxne0-impl [] a = 0
  | *maxne0-impl x [] = 0*
  | *maxne0-impl (x#xs) (a#as) = (if x > 0 then max a (maxne0-impl xs as) else maxne0-impl xs as)*

lemma *maxne0-impl*:
  assumes *length x = length a*
  shows *maxne0-impl x a = maxne0 x a*

using *assms* **by** (*induct x a rule*: *list-induct2*) (*auto*)

lemma *maxne0-impl-le*:
maxne0-impl x a ≤ *Max* (*set* (*a*::*nat list*))
apply (*induct x a rule*: *maxne0-impl.induct*)
apply (*auto simp add*: *max.coboundedI2*)
by (*metis List.finite-set Max-insert Nat.le0 le-max-iff-disj maxne0-impl.elims maxne0-impl.simps*(*2*) *set-empty*)

context
fixes *a b* :: *nat list*
begin

definition *special-solutions* :: (*nat list* × *nat list*) *list*
where
special-solutions = [*sij a b i j* . *i* ← [*0* ..< *length a*], *j* ← [*0* ..< *length b*]]

definition *big-e* :: *nat list* ⇒ *nat* ⇒ *nat list*
where
big-e x j = *map* (λ*i*. *eij a b i j* − *1*) (*filter* (λ*i*. *x* ! *i* ≥ *dij a b i j*) [*0* ..< *length x*])

definition *big-d* :: *nat list* ⇒ *nat* ⇒ *nat list*
where
big-d y i = *map* (λ*j*. *dij a b i j* − *1*) (*filter* (λ*j*. *y* ! *j* ≥ *eij a b i j*) [*0* ..< *length y*])

definition *big-d′* :: *nat list* ⇒ *nat* ⇒ *nat list*
where
big-d′ y i =
(*let l* = *length y*; *n* = *length b in*
if l > *n then* [] *else*
(*let k* = *n* − *l in*
map (λ*j*. *dij a b i* (*j* + *k*) − *1*) (*filter* (λ*j*. *y* ! *j* ≥ *eij a b i* (*j* + *k*)) [*0* ..< *length y*])))

definition *max-y-impl* :: *nat list* ⇒ *nat* ⇒ *nat*
where
max-y-impl x j =
(*if j* < *length b* ∧ *big-e x j* ≠ [] *then Min* (*set* (*big-e x j*))
else Max (*set a*))

definition *max-x-impl* :: *nat list* ⇒ *nat* ⇒ *nat*
where
max-x-impl y i =
(*if i* < *length a* ∧ *big-d y i* ≠ [] *then Min* (*set* (*big-d y i*))
else Max (*set b*))

definition *max-x-impl′* :: *nat list* ⇒ *nat* ⇒ *nat*

where
$max\text{-}x\text{-}impl'\ y\ i =$
$(if\ i < length\ a \wedge big\text{-}d'\ y\ i \neq []\ then\ Min\ (set\ (big\text{-}d'\ y\ i))$
$else\ Max\ (set\ b))$

definition $cond\text{-}a :: nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$cond\text{-}a\ xs\ ys \longleftrightarrow (\forall x{\in}set\ xs.\ x \leq maxne0\ ys\ b)$

definition $cond\text{-}b :: nat\ list \Rightarrow bool$
where
$cond\text{-}b\ xs \longleftrightarrow (\forall k{\leq}length\ a.$
$take\ k\ a \cdot take\ k\ xs \leq b \cdot (map\ (max\text{-}y\text{-}impl\ (take\ k\ xs))\ [0\ ..{<}\ length\ b]))$

definition $boundr\text{-}impl :: nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$boundr\text{-}impl\ x\ y \longleftrightarrow (\forall j{<}length\ b.\ y\ !\ j \leq max\text{-}y\text{-}impl\ x\ j)$

definition $cond\text{-}d :: nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$cond\text{-}d\ xs\ ys \longleftrightarrow (\forall l{\leq}length\ b.\ take\ l\ b \cdot take\ l\ ys \leq a \cdot xs)$

definition $subdprodr\text{-}impl :: nat\ list \Rightarrow bool$
where
$subdprodr\text{-}impl\ ys \longleftrightarrow (\forall l{\leq}length\ b.$
$take\ l\ b \cdot take\ l\ ys \leq a \cdot map\ (max\text{-}x\text{-}impl\ (take\ l\ ys))\ [0\ ..{<}\ length\ a])$

definition $subdprodl\text{-}impl :: nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$subdprodl\text{-}impl\ x\ y \longleftrightarrow (\forall k{\leq}length\ a.\ take\ k\ a \cdot take\ k\ x \leq b \cdot y)$

definition $boundl\text{-}impl\ x\ y \longleftrightarrow (\forall i{<}length\ a.\ x\ !\ i \leq max\text{-}x\text{-}impl\ y\ i)$

definition $static\text{-}bounds$
where
$static\text{-}bounds\ x\ y \longleftrightarrow$
$(let\ mx = maxne0\text{-}impl\ y\ b;\ my = maxne0\text{-}impl\ x\ a\ in$
$(\forall x{\in}set\ x.\ x \leq mx) \wedge (\forall y{\in}set\ y.\ y \leq my))$

definition $check\text{-}cond =$
$(\lambda(x,\ y).\ static\text{-}bounds\ x\ y \wedge a \cdot x = b \cdot y \wedge boundr\text{-}impl\ x\ y \wedge subdprodl\text{-}impl\ x$
$y \wedge subdprodr\text{-}impl\ y)$

definition $check' = filter\ check\text{-}cond$

definition $non\text{-}special\text{-}solutions =$
$(let\ A = Max\ (set\ b);\ B = Max\ (set\ a)$
$in\ minimize\ (check'\ (generate'\ A\ B\ a\ b)))$

definition *solve* = *special-solutions* @ *non-special-solutions*

end

lemma *sorted-wrt-check-generate′*:
sorted-wrt ($<_{rlex2}$) (*check′ a b* (*generate′ A B a b*))
by (*auto simp*: *check′-def intro*!: *sorted-wrt-filter sorted-wrt-generate′ sorted-wrt-tl*)

lemma *big-e*:
set (*big-e a b xs j*) = *hlde-ops.Ej a b j xs*
by (*auto simp*: *hlde-ops.Ej-def big-e-def*)

lemma *big-d*:
set (*big-d a b ys i*) = *hlde-ops.Di a b i ys*
by (*auto simp*: *hlde-ops.Di-def big-d-def*)

lemma *big-d′*:
length ys ≤ *length b* ⟹ *set* (*big-d′ a b ys i*) = *hlde-ops.Di′ a b i ys*
by (*auto simp*: *hlde-ops.Di′-def big-d′-def Let-def*)

lemma *max-y-impl*:
max-y-impl a b x j = *hlde-ops.max-y a b x j*
by (*simp add*: *max-y-impl-def big-e hlde-ops.max-y-def set-empty* [*symmetric*])

lemma *max-x-impl*:
max-x-impl a b y i = *hlde-ops.max-x a b y i*
by (*simp add*: *max-x-impl-def big-d hlde-ops.max-x-def set-empty* [*symmetric*])

lemma *max-x-impl′*:
assumes *length y* ≤ *length b*
shows *max-x-impl′ a b y i* = *hlde-ops.max-x′ a b y i*
by (*simp add*: *max-x-impl′-def big-d′* [*OF assms*] *hlde-ops.max-x′-def set-empty* [*symmetric*])

lemma (**in** *hlde*) *cond-a* [*simp*]: *cond-a b x y* = *cond-A x y*
by (*simp add*: *cond-a-def cond-A-def*)

lemma (**in** *hlde*) *cond-b* [*simp*]: *cond-b a b x* = *cond-B x*
using *max-y-impl* **by** (*auto simp*: *cond-b-def cond-B-def*) *presburger*+

lemma (**in** *hlde*) *boundr-impl* [*simp*]: *boundr-impl a b x y* = *boundr x y*
by (*simp add*: *boundr-impl-def boundr-def max-y-impl*)

lemma (**in** *hlde*) *cond-d* [*simp*]: *cond-d a b x y* = *cond-D x y*
by (*simp add*: *cond-d-def cond-D-def*)

lemma (**in** *hlde*) *subdprodr-impl* [*simp*]: *subdprodr-impl a b y* = *subdprodr y*
using *max-x-impl* **by** (*auto simp*: *subdprodr-impl-def subdprodr-def*) *presburger*+

lemma (**in** *hlde*) *subdprodl-impl* [*simp*]: *subdprodl-impl a b x y* = *subdprodl x y*
  by (*simp add*: *subdprodl-impl-def subdprodl-def*)

lemma (**in** *hlde*) *cond-bound-impl* [*simp*]: *boundl-impl a b x y* = *boundl x y*
  by (*simp add*: *boundl-impl-def boundl-def max-x-impl*)

lemma (**in** *hlde*) *check* [*simp*]:
  check′ a b =
    filter (λ(*x*, *y*). *static-bounds a b x y* ∧ *a* • *x* = *b* • *y* ∧ *boundr x y* ∧
    subdprodl x y ∧
    subdprodr y)
  by (*simp add*: *check′-def check-cond-def*)

conditions B, C, and D from Huet as well as "subdprodr" and "subdprodl" are preserved by smaller solutions

lemma (**in** *hlde*) *le-imp-conds*:
  assumes *le*: $u \le_v x$ $v \le_v y$
    and *len*: *length x* = *m length y* = *n*
  shows *cond-B x* ⟹ *cond-B u*
    and *boundr x y* ⟹ *boundr u v*
    and *a* • *u* = *b* • *v* ⟹ *cond-D x y* ⟹ *cond-D u v*
    and *a* • *u* = *b* • *v* ⟹ *subdprodl x y* ⟹ *subdprodl u v*
    and *subdprodr y* ⟹ *subdprodr v*
proof −
  assume *B*: *cond-B x*
  have *length u* = *m* **using** *len* **and** *le* **by** (*auto*)
  show *cond-B u*
  proof (*unfold cond-B-def*, *intro allI impI*)
    fix *k*
    assume *k*: *k* ≤ *m*
    moreover have ∗: *take k u* $\le_v$ *take k x* **if** *k* ≤ *m* **for** *k*
      using *le* **and** *that* **by** (*intro le-take*) (*auto simp*: *len*)
    ultimately have *take k a* • *take k u* ≤ *take k a* • *take k x*
      by (*intro dotprod-le-right*) (*auto simp*: *len*)
    also have . . . ≤ *b* • *map* (*max-y* (*take k x*)) [*0*..<*n*]
      using *k* **and** *B* **by** (*auto simp*: *cond-B-def*)
    also have . . . ≤ *b* • *map* (*max-y* (*take k u*)) [*0*..<*n*]
      using *le-imp-max-y-ge* [*OF* ∗ [*OF k*]]
      using *k* **by** (*auto simp*: *len intro*!: *dotprod-le-right less-eqI*)
    finally show *take k a* • *take k u* ≤ *b* • *map* (*max-y* (*take k u*)) [*0*..<*n*] **.**
  qed
next
  assume *subdprodr*: *subdprodr y*
  have *length v* = *n* **using** *len* **and** *le* **by** (*auto*)
  show *subdprodr v*
  proof (*unfold subdprodr-def*, *intro allI impI*)
    fix *l*
    assume *l*: *l* ≤ *n*
    moreover have ∗: *take l v* $\le_v$ *take l y* **if** *l* ≤ *n* **for** *l*

using *le* **and** *that* **by** (*intro le-take*) (*auto simp*: *len*)
ultimately have *take l b* $\cdot$ *take l v* $\leq$ *take l b* $\cdot$ *take l y*
by (*intro dotprod-le-right*) (*auto simp*: *len*)
also have ... $\leq$ *a* $\cdot$ *map* (*max-x* (*take l y*)) [*0*..<*m*]
using *l* **and** *subdprodr* **by** (*auto simp*: *subdprodr-def*)
also have ... $\leq$ *a* $\cdot$ *map* (*max-x* (*take l v*)) [*0*..<*m*]
using *le-imp-max-x-ge* [*OF* ∗ [*OF l*]]
using *l* **by** (*auto simp*: *len intro*!: *dotprod-le-right less-eqI*)
finally show *take l b* $\cdot$ *take l v* $\leq$ *a* $\cdot$ *map* (*max-x* (*take l v*)) [*0*..<*m*] **.**
qed
next
assume *C*: *boundr x y*
show *boundr u v*
using *le-imp-max-y-ge* [*OF* ‹$u \leq_v x$›] **and** *C* **and** *le*
by (*auto simp*: *boundr-def len less-eq-def*) (*meson order-trans*)
next
assume *a* $\cdot$ *u* = *b* $\cdot$ *v* **and** *cond-D x y*
then show *cond-D u v*
using *le* **by** (*auto simp*: *cond-D-def len le-length intro*: *dotprod-le-take*)
next
assume *a* $\cdot$ *u* = *b* $\cdot$ *v* **and** *subdprodl x y*
then show *subdprodl u v*
using *le* **by** (*metis subdprodl-def dotprod-le-take le-length len*(*1*))
qed

lemma (**in** *hlde*) *special-solutions* [*simp*]:
shows *set* (*special-solutions a b*) = *Special-Solutions*
proof −
have *set* (*special-solutions a b*) ⊆ *Special-Solutions*
by (*auto simp*: *Special-Solutions-def special-solutions-def*) (*blast*)
moreover have *Special-Solutions* ⊆ *set* (*special-solutions a b*)
by (*auto simp*: *Special-Solutions-def special-solutions-def*)
ultimately show *?thesis* **..**
qed

lemma *set-gen2*:
set (*gen2 A B a b*) = {(*x*, *y*). $x \leq_v$ *replicate* (*length a*) *A* ∧ $y \leq_v$ *replicate* (*length b*) *B*}
(**is** *?L* = *?R*)
proof (*intro equalityI subrelI*)
fix *xs ys* **assume** (*xs*, *ys*) ∈ *?R*
then have ∀ *x*∈*set xs*. *x* ≤ *A* **and** ∀ *y*∈*set ys*. *y* ≤ *B*
and *length xs* = *length a* **and** *length ys* = *length b*
by (*auto simp*: *less-eq-def in-set-conv-nth*)
then have ((*xs*, *a* $\cdot$ *xs*), (*ys*, *b* $\cdot$ *ys*)) ∈ *set* (*alls2 A B a b*) **by** (*auto simp*: *set-alls2*)
then have (λ(*x*, *y*). (*fst x*, *fst y*)) ((*xs*, *a* $\cdot$ *xs*), (*ys*, *b* $\cdot$ *ys*)) ∈ (λ(*x*, *y*). (*fst x*, *fst y*)) ‘ *set* (*alls2 A B a b*)
by (*intro imageI*)

then show (*xs*, *ys*) ∈ *?L* **by** *simp*
qed (*auto simp*: *less-eq-def set-alls2*)

lemma *set-gen2′*:
(λ(*x*, *y*). (*fst x*, *fst y*)) ‘ *set* (*alls2 A B a b*) =
{(*x*, *y*). $x \leq_v$ *replicate* (*length a*) *A* ∧ $y \leq_v$ *replicate* (*length b*) *B*}
using *set-gen2* **by** *simp*

lemma (**in** *hlde*) *in-non-special-solutions*:
assumes (*x*, *y*) ∈ *set* (*non-special-solutions a b*)
shows (*x*, *y*) ∈ *Solutions*
using *assms*
by (*auto dest*!: *minimize-wrtD in-generate′*
simp: *non-special-solutions-def in-Solutions-iff minimize-def set-alls2*)

lemma *generate-unique*:
assumes *i* < *j*
and *j* < *length* (*generate A B a b*)
shows *generate A B a b* ! *i* ≠ *generate A B a b* ! *j*
using *sorted-wrt-nth-less* [*OF sorted-wrt-generate assms*]
by (*auto simp*: *rlex2-irrefl*)

lemma *gen2-unique*:
assumes *i* < *j*
and *j* < *length* (*gen2 A B a b*)
shows *gen2 A B a b* ! *i* ≠ *gen2 A B a b* ! *j*
using *sorted-wrt-nth-less* [*OF sorted-wrt-gen2 assms*]
by (*auto simp*: *rlex2-irrefl*)

lemma *zeroes-ni-generate′*:
(*zeroes* (*length a*), *zeroes* (*length b*)) ∉ *set* (*generate′ A B a b*)
proof −
have *gen2 A B a b* ! *0* = (*zeroes* (*length a*), *zeroes* (*length b*)) **by** (*auto*)
with *gen2-unique* [*of 0 - A B a b*] **show** *?thesis*
by (*auto simp*: *in-set-conv-nth nth-tl generate′-def*)
(*metis One-nat-def Suc-eq-plus1 less-diff-conv zero-less-Suc*)
qed

lemma *set-generate′*:
set (*generate′ A B a b*) =
{(*x*, *y*). (*x*, *y*) ≠ (*zeroes* (*length a*), *zeroes* (*length b*)) ∧ (*x*, *y*) ∈ *set* (*gen2 A B a b*)}
proof
show *set* (*generate′ A B a b*)
⊆ {(*x*, *y*).(*x*, *y*) ≠ (*zeroes* (*length a*), *zeroes* (*length b*)) ∧ (*x*, *y*) ∈ *set* (*gen2 A B a b*)}
using *in-generate′* **and** *mem-Collect-eq* **and** *zeroes-ni-generate′* **by** (*auto*)
next
have (*zeroes* (*length a*), *zeroes* (*length b*)) = *hd* (*gen2 A B a b*)

by (*simp add*: *hd-conv-nth*)
moreover have *set* (*gen2 A B a b*) = *set* (*tl* (*gen2 A B a b*)) ∪ {(*zeroes* (*length a*), *zeroes* (*length b*))}
by (*metis Un-empty-right Un-insert-right gen2-ne calculation list.exhaust-sel list.simps*(*15*))
ultimately show {(*x*, *y*). (*x*, *y*) ≠ (*zeroes* (*length a*), *zeroes* (*length b*)) ∧ (*x*, *y*) ∈ *set* (*gen2 A B a b*)}
⊆ *set* (*generate′ A B a b*)
unfolding *generate′-def* **by** *blast*
qed

lemma *set-generate′′*:
set (*generate′ A B a b*) =
{(*x*, *y*). (*x*, *y*) ≠ (*zeroes* (*length a*), *zeroes* (*length b*)) ∧ $x \le_v$ *replicate* (*length a*) *A* ∧ $y \le_v$ *replicate* (*length b*) *B*}
by (*simp add*: *set-generate′ set-gen2′*)

lemma (**in** *hlde*) *zeroes-ni-non-special-solutions*:
shows (*zeroes m*, *zeroes n*) ∉ *set* (*non-special-solutions a b*)
proof −
define *All-lex* **where**
All-lex: *All-lex* = *gen2* (*Max* (*set b*)) (*Max* (*set a*)) *a b*
define *z* **where** *z*: *z* = (*zeroes m*, *zeroes n*)
have *set* (*non-special-solutions a b*) ⊆ *set* (*tl* (*All-lex*))
by (*auto simp*: *All-lex generate′-def non-special-solutions-def minimize-def dest*: *minimize-wrtD*)
moreover have *z* ∉ *set* (*tl* (*All-lex*))
using *zeroes-ni-generate′ All-lex z* **by** (*auto simp*: *generate′-def*)
ultimately show *?thesis*
using *z* **by** *blast*
qed

4.1.1 Correctness: *solve* generates only minimal solutions.

lemma (**in** *hlde*) *solve-subset-Minimal-Solutions*:
shows *set* (*solve a b*) ⊆ *Minimal-Solutions*
proof (*rule subrelI*)
let *?a* = *Max* (*set a*) **and** *?b* = *Max* (*set b*)
fix *x y*
assume *ass*: (*x*, *y*) ∈ *set* (*solve a b*)
then consider (*x*, *y*) ∈ *set* (*special-solutions a b*) | (*x*, *y*) ∈ *set* (*non-special-solutions a b*)
unfolding *solve-def* **and** *set-append* **by** *blast*
then show (*x*, *y*) ∈ *Minimal-Solutions*
proof (*cases*)
case *1*
then have (*x*, *y*) ∈ *Special-Solutions*
unfolding *special-solutions* **.**
then show *?thesis*

```
      by (simp add: Special-Solutions-in-Minimal-Solutions)
  next
    let ?xs = [(x, y) ← generate′ ?b ?a a b.
      static-bounds a b x y ∧ a • x = b • y ∧ boundr x y ∧ cond-B x ∧ cond-D x y ∧
      subdprodl x y ∧
      subdprodr y]
    case 2
    then have conds: ∀ e∈set x. e ≤ Max (set b) boundr x y
      subdprodl x y subdprodr y
      and xs: (x, y) ∈ set (minimize ?xs)
      by (auto simp: non-special-solutions-def minimize-def set-alls2
        dest!: minimize-wrtD in-generate′)
        (metis in-set-conv-nth)
    have sol: (x, y) ∈ Solutions
     using ass by (auto simp: solve-def Special-Solutions-in-Solutions in-non-special-solutions)
    then have len: length x = m length y = n by (auto simp: Solutions-def)
    have nonzero x
      using sol Solutions-snd-not-0 [of y x]
     by (metis 2 eq-0-iff len nonzero-Solutions-iff nonzero-iff zeroes-ni-non-special-solutions)
    moreover have ¬ (∃ (u, v) ∈ Minimal-Solutions. u @ v <_v x @ y)
    proof
      let ?P = λ(x, y) (u, v). ¬ x @ y <_v u @ v
     let ?Q = (λ(x, y). static-bounds a b x y ∧ a • x = b • y ∧ boundr x y ∧ cond-B
x ∧ cond-D x y ∧
        subdprodl x y ∧
        subdprodr y)
      note sorted = sorted-wrt-generate′ [THEN sorted-wrt-filter, of ?Q ?b ?a a b]
     note * = in-minimize-wrt-False [OF - sorted, of (x, y) ?P, OF - xs [unfolded
minimize-def]]

      assume ∃ (u, v)∈Minimal-Solutions. u @ v <_v x @ y
      then obtain u and v where
        uv: (u, v) ∈ Minimal-Solutions and less: u @ v <_v x @ y by blast
      from uv and less have le: u ≤_v x v ≤_v y and sol′: a • u = b • v
        and nonzero: nonzero u
     using sol by (auto simp: Minimal-Solutions-def Solutions-def elim!: less-append-cases)

      with le-imp-conds(2,4,5) [OF le] and conds(2−)
      have conds′: ∀ e∈set u. e ≤ Max (set b) boundr u v
        subdprodl u v subdprodr v
         using conds(1,3,4) by (auto simp: len less-eq-def) (metis in-set-conv-nth
le-trans len(1))
      moreover have static-bounds a b u v
        using max-coeff-bound [OF uv] and Minimal-Solutions-length [OF uv]
        by (auto simp: static-bounds-def maxne0-impl)
      moreover have x ≤_v replicate m ?b
        using xs set-generate′ [of Max (set b) Max (set a) a b]
          cond-A-def conds(1) le-replicateI len by metis
      moreover have y ≤_v replicate n ?a
```

```
      using xs by (auto simp: less-eqI minimize-def set-generate′ set-alls2 dest!:
minimize-wrtD)
    ultimately have (u, v) ∈ set ?xs
    using sol′ and set-generate′′ [of ?b ?a a b] and uv [THEN Minimal-Solutions-imp-Solutions]
and nonzero
      by (simp add: set-gen2) (metis in-set-replicate le order-vec.dual-order.trans
nonzero-iff)
    from ∗ [OF - - - this] and less show False
      using less-imp-rlex and rlex-not-sym by force
  qed
  ultimately show ?thesis by (simp add: Minimal-SolutionsI′ sol)
 qed
qed
```

4.1.2 Completeness: every minimal solution is generated by *solve*

```
lemma (in hlde) Minimal-Solutions-subset-solve:
 shows Minimal-Solutions ⊆ set (solve a b)
proof (rule subrelI)
 fix x y
 assume min: (x, y) ∈ Minimal-Solutions
 then have sol: a • x = b • y length x = m length y = n
  and [dest]: x = zeroes m =⇒ y = zeroes n =⇒ False
  by (auto simp: Minimal-Solutions-def Solutions-def nonzero-iff)
 consider (special) (x, y) ∈ Special-Solutions
  | (not-special) (x, y) ∉ Special-Solutions by blast
 then show (x, y) ∈ set (solve a b)
 proof (cases)
  case special
  then show ?thesis
   by (simp add: no0 solve-def)
 next
  define all where all = generate′ (Max (set b)) (Max (set a)) a b
  have ∗: ∀ (u, v) ∈ set (check′ a b all). ¬ u @ v <v x @ y
   using min and no0
  by (auto simp: all-def set-generate′′ neq-0-iff′ nonzero-iff dest!: Minimal-Solutions-min)

  case not-special
  from conds [OF min] and not-special
  have (x, y) ∈ set (check′ a b all)
   using max-coeff-bound [OF min] and maxne0-le-Max
    and Minimal-Solutions-length [OF min]
  apply (auto simp: sol all-def set-generate′′ cond-A-def less-eq-def static-bounds-def
maxne0-impl)
    apply (metis le-trans nth-mem sol(2))
    by (metis le-trans nth-mem sol(3))
  from in-minimize-wrtI [OF this, of λ(x, y) (u, v). ¬ x @ y <v u @ v] ∗
  have (x, y) ∈ set (non-special-solutions a b)
   by (auto simp: non-special-solutions-def minimize-def all-def)
```

then show *?thesis*
by (*simp add*: *solve-def*)
qed
qed

The main correctness and completeness result of our algorithm.

lemma (**in** *hlde*) *solve* [*simp*]:
shows *set* (*solve a b*) = *Minimal-Solutions*
using *Minimal-Solutions-subset-solve* **and** *solve-subset-Minimal-Solutions* **by** *blast*

5 Making the Algorithm More Efficient

locale *bounded-gen-check* =
fixes C :: *nat list* ⇒ *nat* ⇒ *bool*
and B :: *nat*
assumes *bound*: $\bigwedge x\ xs\ s.\ x > B \Longrightarrow C\ (x \# xs)\ s = False$
and *cond-antimono*: $\bigwedge x\ x'\ xs\ s\ s'.\ C\ (x \# xs)\ s \Longrightarrow x' \le x \Longrightarrow s' \le s \Longrightarrow C\ (x' \# xs)\ s'$
begin

function *incs* :: *nat* ⇒ *nat* ⇒ (*nat list* × *nat*) ⇒ (*nat list* × *nat*) *list*
where
incs a x (*xs*, *s*) =
(*let t* = *s* + *a* ∗ *x in*
if C (*x* # *xs*) *t then* (*x* # *xs*, *t*) # *incs a* (*Suc x*) (*xs*, *s*) *else* [])
by (*auto*)
termination
by (*relation measure* (λ(*a*, *x*, *xs*, *s*). *B* + *1* − *x*), *rule wf-measure*, *case-tac x* > *B*)
(*use bound* **in** *auto*)
declare *incs.simps* [*simp del*]

lemma *in-incs*:
assumes (*ys*, *t*) ∈ *set* (*incs a x* (*xs*, *s*))
shows *length ys* = *length xs* + *1* ∧ *t* = *s* + *hd ys* ∗ *a* ∧ *tl ys* = *xs* ∧ *C ys t*
using *assms*
by (*induct a x* (*xs*, *s*) *arbitrary*: *ys t rule*: *incs.induct*)
(*subst* (*asm*) (*2*) *incs.simps*, *auto simp*: *Let-def*)

lemma *incs-Nil* [*simp*]: *x* > *B* ⟹ *incs a x* (*xs*, *s*) = []
by (*induct a x* (*xs*, *s*) *rule*: *incs.induct*) (*simp add*: *incs.simps bound*)

lemma *incs-filter*:
assumes *x* ≤ *B*
shows *incs a x* = (λ(*xs*, *s*). *filter* (*cond-cons C*) (*map* (λ*x*. (*x* # *xs*, *s* + *a* ∗ *x*)) [*x* ..< *B* + *1*]))
proof
fix *xss*

show *incs a x xss* = (λ(*xs*, *s*). *filter* (*cond-cons C*) (*map* (λ*x*. (*x* # *xs*, *s* + *a* ∗ *x*)) [*x* ..< *B* + *1*])) *xss*
using *assms*
proof (*induct a x xss rule*: *incs.induct*)
case (*1 a x xs s*)
then show *?case*
by (*unfold incs.simps* [*of a x*], *cases x* = *B*)
(*auto simp*: *filter-empty-conv Let-def cond-cons-def upt-conv-Cons intro*: *cond-antimono*)
qed
qed

fun *gen-check* :: *nat list* ⇒ (*nat list* × *nat*) *list*
where
gen-check [] = [([], *0*)]
| *gen-check* (*a* # *as*) = *concat* (*map* (*incs a 0*) (*gen-check as*))

lemma *gen-check-len*:
assumes (*ys*, *s*) ∈ *set* (*gen-check as*)
shows *length ys* = *length as*
using *assms*
proof (*induct as arbitrary*: *ys s*)
case (*Cons a as*)
have ∃(*la*,*t*) ∈ *set* (*gen-check as*). (*ys*, *s*) ∈ *set* (*incs a 0* (*la*,*t*))
using *Cons.prems*(*1*) **by** *auto*
moreover obtain *la t* **where** (*la*,*t*) ∈ *set* (*gen-check as*)
using *calculation* **by** *auto*
moreover have *length ys* = *length la* + *1*
using *calculation*
by (*metis* (*no-types*, *lifting*) *Cons.hyps case-prodE in-incs*)
moreover have *length la* = *length as*
using *calculation*
using *Cons.hyps Cons.prems* **by** *fastforce*
ultimately show *?case* **by** *simp*
qed (*auto*)

lemma *in-gen-check*:
assumes (*xs*, *s*) ∈ *set* (*gen-check as*)
shows *length xs* = *length as* ∧ *s* = *as* · *xs*
using *assms*
apply (*induct as arbitrary*: *xs s*)
apply (*auto simp*: *in-incs*)
apply (*case-tac xs*)
apply (*auto dest*: *in-incs*)
done

lemma *gen-check-filter*:
gen-check as = *filter* (*suffs C as*) (*alls B as*)
proof (*induct as*)

```
next
  case (Cons a as)
  have filter (suffs C (a # as)) (alls B (a # as)) =
    filter (λ(xs, s). cond-cons C (xs, s) ∧ suffs C as (tl xs, as · tl xs)) (alls B (a #
as))
    by (intro filter-cong [OF refl])
      (auto simp: set-alls suffs.simps all-Suc-le-conv ac-simps split: list.splits)
  also have ... =
     concat (map (λ(xs, s). filter (cond-cons C) (map (λx. (x # xs, s + a ∗ x))
[0..<B + 1]))
      (filter (suffs C as) (alls B as)))
    unfolding alls.simps
    unfolding filter-concat
    unfolding map-map
    by (subst concat-map-filter-filter [symmetric, where Q = suffs C as])
      (auto simp: set-alls intro!: arg-cong [of - - concat] filter-cong)
  finally have ∗: filter (suffs C (a # as)) (alls B (a # as)) =
    concat (map (λ(xs, s).
       filter (cond-cons C) (map (λx. (x # xs, s + a ∗ x)) [0..<B + 1])) (filter
(suffs C as) (alls B as))) .
  have gen-check (a # as) = filter (suffs C (a # as)) (alls B (a # as))
    unfolding ∗
    by (simp add: incs-filter [OF zero-le] Cons)
  then show ?case by simp
qed simp

lemma in-gen-check-cond:
  assumes (xs, s) ∈ set (gen-check as)
  shows ∀j≤length xs. drop j xs ≠ [] ⟶ C (drop j xs) (s − take j as · take j xs)
  using assms
  apply (induct as arbitrary: xs s)
   apply auto
  apply (case-tac xs)
   apply auto
  apply (case-tac j)
   apply (auto dest: in-incs)
  done

lemma sorted-gen-check:
  sorted-wrt (<_rlex) (map fst (gen-check xs))
proof −
  have sort-map: sorted-wrt (λx y. x <_rlex y) (map fst (alls B xs))
    using sorted-wrt-alls by auto
  then have sorted-wrt (λx y. fst x <_rlex fst y) (alls B xs)
    using sorted-wrt-map-distr [of (<_rlex) fst alls B xs]
    by (auto)
  then have sorted-wrt (λx y. fst x <_rlex fst y) (filter (suffs C xs) (alls B xs))
    using sorted-wrt-alls sorted-wrt-filter sorted-wrt-map
    by blast
```

then show *?thesis*
using *gen-check-filter*
by (*simp add*: *case-prod-unfold sorted-wrt-map-mono*)
qed

end

locale *bounded-generate-check* =
c2: *bounded-gen-check* C_2 B_2 **for** C_2 B_2 +
fixes C_1 **and** B_1
assumes *cond1*: $\bigwedge b$ *ys*. *ys* ∈ *fst* ‘ *set* (*c2.gen-check b*) ⟹ *bounded-gen-check* (C_1 *b ys*) (B_1 *b*)
begin

definition *generate-check a b* =
[(*xs*, *ys*). *ys* ← *c2.gen-check b*, *xs* ← *bounded-gen-check.gen-check* (C_1 *b* (*fst ys*)) *a*]

lemma *generate-check-filter-conv*:
generate-check a b = [(*xs*, *ys*).
ys ← *filter* (*suffs* C_2 *b*) (*alls* B_2 *b*),
xs ← *filter* (*suffs* (C_1 *b* (*fst ys*)) *a*) (*alls* (B_1 *b*) *a*)]
using *bounded-gen-check.gen-check-filter* [*OF cond1*]
by (*force simp*: *generate-check-def c2.gen-check-filter intro*!: *arg-cong* [*of - - concat*] *map-cong*)

lemma *generate-check-filter*:
generate-check a b = [(*xs*, *ys*) ← *alls2* (B_1 *b*) B_2 *a b*. *suffs* (C_1 *b* (*fst ys*)) *a xs* ∧ *suffs* C_2 *b ys*]
by (*auto intro*: *arg-cong* [*of - - concat*]
simp: *generate-check-filter-conv alls2-def filter-concat concat-map-filter filter-map o-def*)

lemma *tl-generate-check-filter*:
assumes *suffs* (C_1 *b* (*zeroes* (*length b*))) *a* (*zeroes* (*length a*), *0*)
and *suffs* C_2 *b* (*zeroes* (*length b*), *0*)
shows *tl* (*generate-check a b*) = [(*xs*, *ys*) ← *tl* (*alls2* (B_1 *b*) B_2 *a b*). *suffs* (C_1 *b* (*fst ys*)) *a xs* ∧ *suffs* C_2 *b ys*]
using *assms*
by (*unfold generate-check-filter*, *subst* (*1 2*) *alls2-Cons-tl-conv*) *auto*

end

context
fixes *a b* :: *nat list*
begin

fun *cond1*
where

cond1 ys [] *s* ⟷ *True*
| *cond1 ys* (*x* # *xs*) *s* ⟷ *s* ≤ *b* · *ys* ∧ *x* ≤ *maxne0-impl ys b*

lemma *max-x-impl′-conv*:
i < *length a* ⟹ *length y* = *length b* ⟹ *max-x-impl′ a b y i* = *max-x-impl a b y i*
by (*auto simp*: *max-x-impl′-def max-x-impl-def Let-def big-d′-def big-d-def*)

fun *cond2*
where
cond2 [] *s* ⟷ *True*
| *cond2* (*y* # *ys*) *s* ⟷ *y* ≤ *Max* (*set a*) ∧ *s* ≤ *a* · *map* (*max-x-impl′ a b* (*y* # *ys*)) [*0* ..< *length a*]

lemma *le-imp-big-d′-subset*:
assumes $v \leq_v y$
shows *set* (*big-d′ a b v i*) ⊆ *set* (*big-d′ a b y i*)
using *assms* **and** *le-trans*
by (*auto simp*: *Let-def big-d′-def less-eq-def hlde-ops.dij-def hlde-ops.eij-def*)

lemma *finite-big-d′*:
finite (*set* (*big-d′ a b y i*))
by (*rule finite-subset* [*of* - (λ*j*. *dij a b i* (*j* + *length b* − *length y*) − *1*) ‘ {*0* ..< *length y*}])
(*auto simp*: *Let-def big-d′-def*)

lemma *Min-big-d′-le*:
assumes *i* < *length a*
and *big-d′ a b y i* ≠ []
and *length y* ≤ *length b*
shows *Min* (*set* (*big-d′ a b y i*)) ≤ *Max* (*set b*) (**is** *?m* ≤ -)
proof −
have *?m* ∈ *set* (*big-d′ a b y i*)
using *assms* **and** *finite-big-d′* **and** *Min-in* **by** *auto*
then obtain *j* **where**
j: *?m* = *dij a b i* (*j* + *length b* − *length y*) − *1 j* < *length y y* ! *j* ≥ *eij a b i* (*j* + *length b* − *length y*)
by (*auto simp*: *big-d′-def Let-def split*: *if-splits*)
then have *j* + *length b* − *length y* < *length b*
using *assms* **by** *auto*
moreover
have *lcm* (*a* ! *i*) (*b* ! (*j* + *length b* − *length y*)) *div a* ! *i* ≤ *b* ! (*j* + *length b* − *length y*) **by** (*rule lcm-div-le′*)
ultimately show *?thesis*
using *j* **and** *assms*
by (*auto simp*: *hlde-ops.dij-def*)
(*meson List.finite-set Max-ge diff-le-self le-trans less-le-trans nth-mem*)
qed

lemma *le-imp-max-x-impl′-ge*:
  assumes $v \leq_v y$
    and $i <$ *length a*
  shows *max-x-impl′ a b v i* $\geq$ *max-x-impl′ a b y i*
  using *assms* **and** *le-imp-big-d′-subset* [*OF assms*(*1*), *of i*]
    and *Min-in* [*OF finite-big-d′*, *of y i*]
    and *finite-big-d′* **and** *Min-le*
  by (*auto simp*: *max-x-impl′-def Let-def intro*!: *Min-big-d′-le* [*of i y*])
    (*fastforce simp*: *big-d′-def intro*: *leI*)

end

global-interpretation *c12*: *bounded-generate-check* (*cond2 a b*) *Max* (*set a*) *cond1* *λb. Max* (*set b*)
  defines *c2-gen-check* = *c12.c2.gen-check* **and** *c2-incs* = *c12.c2.incs*
    and *c12-generate-check* = *c12.generate-check*
proof −
  { fix *x xs s* **assume** *Max* (*set a*) < *x*
    then have *cond2 a b* (*x* # *xs*) *s* = *False* **by** (*auto*) **}**
  note *1* = *this*

  { fix *x x′ xs s s′* **assume** *cond2 a b* (*x* # *xs*) *s* **and** *x′* ≤ *x* **and** *s′* ≤ *s*
      moreover have *map* (*max-x-impl′ a b* (*x* # *xs*)) [*0..<length a*] $\leq_v$ *map* (*max-x-impl′ a b* (*x′* # *xs*)) [*0..<length a*]
      using *le-imp-max-x-impl′-ge* [*of x′ # xs x # xs*] **and** ‹*x′* ≤ *x*›
      by (*auto simp*: *le-Cons less-eq-def All-less-Suc2*)
    ultimately have *cond2 a b* (*x′* # *xs*) *s′*
      by (*auto simp*: *le-Cons*) (*metis dotprod-le-right le-trans length-map map-nth*)
}
  note *2* = *this*

  interpret *c2*: *bounded-gen-check cond2 a b Max* (*set a*) **by** (*standard*) *fact*+

  { fix *b ys x xs s* **assume** *ys* ∈ *fst ‘ set* (*c2.gen-check b*) **and** *Max* (*set b*) < *x*
  then have *cond1 b ys* (*x* # *xs*) *s* = *False*
   by (*auto dest*!: *c2.in-gen-check*) (*metis leD less-le-trans maxne0-impl maxne0-le-Max*)
}
  note *3* = *this*

  { fix *b ys x x′ xs s s′* **assume** *ys* ∈ *fst ‘ set* (*c2.gen-check b*) **and** *cond1 b ys* (*x* # *xs*) *s*
      and *x′* ≤ *x* **and** *s′* ≤ *s*
    then have *cond1 b ys* (*x′* # *xs*) *s′* **by** *auto* **}**
  note *4* = *this*

  show *bounded-generate-check* (*cond2 a b*) (*Max* (*set a*)) *cond1* (*λb. Max* (*set b*))
    using *1* **and** *2* **and** *3* **and** *4* **by** (*unfold-locales*) *metis*+
qed

definition *post-cond a b* = (λ(*x*, *y*). *static-bounds a b x y* ∧ *a* · *x* = *b* · *y* ∧ *boundr-impl a b x y*)

definition *fast-filter a b* =
  filter (*post-cond a b*) (*map* (λ(*x*, *y*). (*fst x*, *fst y*)) (*tl* (*c12-generate-check a b a b*)))

```
lemma cond1-cond2-zeroes:
  shows suffs (cond1 b (zeroes (length b))) a (zeroes (length a), 0)
    and suffs (cond2 a b) b (zeroes (length b), 0)
   apply (auto simp: suffs.simps cond-cons-def split: list.splits)
     apply (metis dotprod-0-right length-drop)
    apply (metis Cons-replicate-eq Nat.le0)
   apply (metis Cons-replicate-eq Nat.le0)
  by (metis Nat.le0 dotprod-0-right length-drop)

lemma suffs-cond1I:
  assumes ∀y∈set aa. y ≤ maxne0-impl aaa b
    and length aa = length a
    and a · aa = b · aaa
  shows suffs (cond1 b aaa) a (aa, b · aaa)
  using assms
  apply (auto simp: suffs.simps cond-cons-def split: list.splits)
   apply (metis dotprod-le-drop)
  by (metis in-set-dropD list.set-intros(1))

lemma suffs-cond2-conv:
  assumes length ys = length b
  shows suffs (cond2 a b) b (ys, b · ys) ⟷
    (∀y∈set ys. y ≤ Max (set a)) ∧ subdprodr-impl a b ys
    (is ?L ⟷ ?R)
proof
  assume ∗: ?L
  then have ∀y∈set ys. y ≤ Max (set a)
    apply (auto simp: suffs.simps cond-cons-def in-set-conv-nth split: list.splits)
    apply (auto simp: hd-drop-conv-nth [symmetric])
    apply (case-tac drop i ys)
      apply simp-all
    using less-or-eq-imp-le by blast
  moreover
  { fix l assume l: l ≤ length b
    have take l b · take l ys ≤ b · ys
      using l and assms by (simp add: dotprod-le-take)
    also have ... ≤ a · map (max-x-impl′ a b ys) [0 ..< length a]
      using ∗ apply (auto simp: suffs.simps cond-cons-def split: list.splits)
      apply (drule-tac x = 0 in spec)
        apply (cases ys)
       apply auto
      done
```

also have . . . = *a* · *map* (*max-x-impl a b ys*) [*0* ..< *length a*]
using *max-x-impl′-conv* [*OF* - *assms*, *of* - *a*]
by (*metis* (*mono-tags*, *lifting*) *atLeastLessThan-iff map-eq-conv set-upt*)
also have . . . ≤ *a* · *map* (*max-x-impl a b* (*take l ys*)) [*0* ..< *length a*]
unfolding *max-x-impl* **using** *hlde-ops.max-x-le-take* [*OF eq-imp-le*, *OF assms*, *of a*]
by (*intro dotprod-le-right*) (*auto simp*: *less-eq-def*)
finally have *take l b* · *take l ys* ≤ *a* · *map* (*max-x-impl a b* (*take l ys*)) [*0* ..< *length a*] **.**
}
ultimately show *?R* **by** (*auto simp*: *subdprodr-impl-def*)
next
assume ∗: *?R*
then have ∀ *y*∈*set ys*. *y* ≤ *Max* (*set a*) **and** *subdprodr-impl a b ys* **by** *auto*
moreover
{ fix *i* **assume** *i*: *i* ≤ *length b*
have *drop i b* · *drop i ys* ≤ *b* · *ys*
using *i* **and** *assms* **by** (*simp add*: *dotprod-le-drop*)
also have . . . ≤ *a* · *map* (*max-x-impl a b ys*) [*0* ..< *length a*]
using ∗ **and** *assms* **by** (*auto simp*: *subdprodr-impl-def*)
also have . . . = *a* · *map* (*max-x-impl′ a b ys*) [*0* ..< *length a*]
using *max-x-impl′-conv* [*OF* - *assms*, *of* - *a*]
by (*metis* (*mono-tags*, *lifting*) *atLeastLessThan-iff map-eq-conv set-upt*)
also have . . . ≤ *a* · *map* (*max-x-impl′ a b* (*drop i ys*)) [*0* ..< *length a*]
using *hlde-ops.max-x′-le-drop* [*OF eq-imp-le*, *OF assms*, *of a*]
by (*intro dotprod-le-right*) (*auto simp*: *less-eq-def max-x-impl′ i assms*)
finally have *drop i b* · *drop i ys* ≤ *a* · *map* (*max-x-impl′ a b* (*drop i ys*)) [*0* ..< *length a*] **.**
}
ultimately show *?L*
using *assms*
apply (*auto simp*: *suffs.simps cond-cons-def split*: *list.splits*)
apply (*metis in-set-dropD list.set-intros*(*1*))
apply *force*
done
qed

lemma *suffs-cond2I*:
assumes ∀ *y*∈*set aaa*. *y* ≤ *Max* (*set a*)
and *length aaa* = *length b*
and *subdprodr-impl a b aaa*
shows *suffs* (*cond2 a b*) *b* (*aaa*, *b* · *aaa*)
using *assms* **by** (*subst suffs-cond2-conv*) *simp-all*

lemma *check-cond-conv*:
assumes (*x*, *y*) ∈ *set* (*alls2* (*Max* (*set b*)) (*Max* (*set a*)) *a b*)
shows *check-cond a b* (*fst x*, *fst y*) ⟷
static-bounds a b (*fst x*) (*fst y*) ∧ *a* · *fst x* = *b* · *fst y* ∧ *boundr-impl a b* (*fst x*) (*fst y*) ∧

suffs (*cond1 b* (*fst y*)) *a x* ∧
suffs (*cond2 a b*) *b y*
using *assms*
apply (*cases x*; *cases y*; *auto simp*: *static-bounds-def check-cond-def set-alls2 split*: *list.splits*)
apply (*auto intro*: *suffs-cond1I suffs-cond2I simp*: *subdprodl-impl-def suffs-cond2-conv*)
apply (*metis in-set-conv-nth*)
by (*metis dotprod-le-take*)

lemma *tune*:
check′ a b (*generate′* (*Max* (*set b*)) (*Max* (*set a*)) *a b*) = *fast-filter a b*
using *cond1-cond2-zeroes*
by (*auto simp*: *c12.tl-generate-check-filter check′-def generate′-def map-tl* [*symmetric*] *filter-map post-cond-def fast-filter-def*
intro!: *map-cong filter-cong dest*: *list.set-sel*(*2*) [*THEN check-cond-conv*, *OF alls2-ne*])

locale *bounded-incs* =
fixes *cond* :: *nat list* ⇒ *nat* ⇒ *bool*
and *B* :: *nat*
assumes *bound*: ⋀*x xs s*. *x* > *B* ⟹ *cond* (*x* # *xs*) *s* = *False*
begin

function *incs* :: *nat* ⇒ *nat* ⇒ (*nat list* × *nat*) ⇒ (*nat list* × *nat*) *list*
where
incs a x (*xs*, *s*) =
(*let t* = *s* + *a* ∗ *x in*
if cond (*x* # *xs*) *t then* (*x* # *xs*, *t*) # *incs a* (*Suc x*) (*xs*, *s*) *else* [])
by (*auto*)
termination
by (*relation measure* (λ(*a*, *x*, *xs*, *s*). *B* + *1* − *x*), *rule wf-measure*, *case-tac x* > *B*)
(*use bound* **in** *auto*)
declare *incs.simps* [*simp del*]

lemma *in-incs*:
assumes (*ys*, *t*) ∈ *set* (*incs a x* (*xs*, *s*))
shows *length ys* = *length xs* + *1* ∧ *t* = *s* + *hd ys* ∗ *a* ∧ *tl ys* = *xs* ∧ *cond ys t*
using *assms*
by (*induct a x* (*xs*, *s*) *arbitrary*: *ys t rule*: *incs.induct*)
(*subst* (*asm*) (*2*) *incs.simps*, *auto simp*: *Let-def*)

lemma *incs-Nil* [*simp*]: *x* > *B* ⟹ *incs a x* (*xs*, *s*) = []
by (*induct a x* (*xs*, *s*) *rule*: *incs.induct*) (*auto simp*: *Let-def incs.simps bound*)

end

global-interpretation *incs1*:
bounded-incs (*cond1 b ys*) (*Max* (*set b*))

for *b ys* :: *nat list*
defines *c1-incs* = *incs1.incs*
proof
fix *x xs s*
assume *Max* (*set b*) < *x*
then show *cond1 b ys* (*x* # *xs*) *s* = *False*
using *maxne0-impl-le* [*of ys b*] **by** *auto*
qed

fun *c1-gen-check*
where
c1-gen-check b ys [] = [([], *0*)]
| *c1-gen-check b ys* (*a* # *as*) = *concat* (*map* (*c1-incs b ys a 0*) (*c1-gen-check b ys as*))

definition *generate-check a b* = [(*xs*, *ys*). *ys* ← *c2-gen-check a b b*, *xs* ← *c1-gen-check b* (*fst ys*) *a*]

lemma *c1-gen-check-conv*:
assumes (*ys*, *s*) ∈ *set* (*c2-gen-check a b b*)
shows *c1-gen-check b ys a* = *bounded-gen-check.gen-check* (*cond1 b ys*) *a*
proof −
interpret *c1*: *bounded-gen-check* (*cond1 b ys*) *Max* (*set b*)
by (*unfold-locales*) (*auto*, *meson leD less-le-trans maxne0-impl-le*)
have *eq*: *c1-incs b ys a1 0* (*a*, *ba*) = *c1.incs a1 0* (*a*, *ba*) **if** (*a*, *ba*) ∈ *set* (*c1.gen-check a2*)
for *a a1 a2 ba*
using *that*
by (*induct rule*: *c1.incs.induct*)
(*auto dest*!: *c1.in-gen-check simp*: *Let-def incs1.incs.simps c1.incs.simps*)
show *?thesis*
by (*induct a*) (*auto intro*!: *arg-cong* [*of* - - *concat*] *dest*: *eq*)
qed

5.1 Code Generation

lemma *solve-efficient* [*code*]:
solve a b = *special-solutions a b* @ *minimize* (*fast-filter a b*)
by (*auto simp*: *solve-def non-special-solutions-def tune*)

lemma *c12-generate-check-code* [*code-unfold*]:
c12-generate-check a b a b = *generate-check a b*
by (*auto simp*: *generate-check-def c12.generate-check-def c1-gen-check-conv intro*!: *arg-cong* [*of* - - *concat*])

end

References

[1] G. Huet. An algorithm to generate the basis of solutions to homogeneous linear diophantine equations. *Information Processing Letters*, 7(3):144–147, 1978.