

Diophantine Equations*

Florian MeSSner Julian Parsert Jonas Schöpf
Christian Sternagel

May 14, 2024

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

1	Vectors as Lists of Naturals	2
1.1	The Inner Product	3
1.2	The Pointwise Order on Vectors	5
1.3	Pointwise Subtraction	11
1.4	The Lexicographic Order on Vectors	13
1.5	Code Equations	15
2	Homogeneous Linear Diophantine Equations	15
2.1	Further Constraints on Minimal Solutions	18
2.2	Pointwise Restricting Solutions	23
2.3	Special Solutions	31
2.4	Huet's conditions	34
2.5	New conditions: facilitating generation of candidates from right to left	34
3	Minimization	40
3.1	Reverse-Lexicographic Enumeration of Potential Minimal So- lutions	42
3.1.1	Completeness: every minimal solution is generated by <i>solutions</i>	45
3.1.2	Correctness: <i>solutions</i> generates only minimal solutions.	45

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

4	Computing Minimal Complete Sets of Solutions	46
4.1	The Algorithm	49
4.1.1	Correctness: <i>solve</i> generates only minimal solutions. . .	56
4.1.2	Completeness: every minimal solution is generated by <i>solve</i>	58
5	Making the Algorithm More Efficient	59
5.1	Code Generation	68

1 Vectors as Lists of Naturals

```
theory List-Vector
  imports Main
begin
```

```
lemma lex-lengthD:  $(x, y) \in \text{lex } P \implies \text{length } x = \text{length } y$ 
  by (auto simp: lexord-lex)
```

```
lemma lex-take-index:
```

```
  assumes  $(xs, ys) \in \text{lex } r$ 
  obtains  $i$  where  $\text{length } ys = \text{length } xs$ 
    and  $i < \text{length } xs$  and  $\text{take } i \text{ } xs = \text{take } i \text{ } ys$ 
    and  $(xs ! i, ys ! i) \in r$ 
```

```
proof -
```

```
  obtain  $n$   $us$   $x$   $xs'$   $y$   $ys'$  where  $(xs, ys) \in \text{lexn } r$   $n$  and  $\text{length } xs = n$  and  $\text{length } ys = n$ 
    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  $(v::\text{nat}) > w$ 
    and  $(v * b) \bmod a = (w * b) \bmod a$ 
  shows  $((v - w) * b) \bmod a = 0$ 
  using assms by (simp add: mod-eq-dvd-iff-nat algebra-simps)
```

— The 0-vector of length n .

```
abbreviation zeroes ::  $\text{nat} \Rightarrow \text{nat list}$ 
```

```
  where
    zeroes  $n \equiv \text{replicate } n \ 0$ 
```

```
lemma rep-upd-unit:
```

```
  assumes  $x = (\text{zeroes } n)[i := a]$ 
  shows  $\forall j < \text{length } x. (j \neq i \longrightarrow x ! j = 0) \wedge (j = i \longrightarrow x ! j = a)$ 
```

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

definition *nonzero-iff*: $\text{nonzero } xs \longleftrightarrow (\exists x \in \text{set } xs. x \neq 0)$

lemma *nonzero-append* [*simp*]:

$\text{nonzero } (xs @ ys) \longleftrightarrow \text{nonzero } xs \vee \text{nonzero } ys$ **by** (*auto simp: nonzero-iff*)

1.1 The Inner Product

definition *dotprod* :: $\text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{nat}$ (**infixl** \cdot 70)

where

$xs \cdot ys = (\sum_{i < \min(\text{length } xs) (\text{length } ys)}. xs ! i * ys ! i)$

lemma *dotprod-code* [*code*]:

$xs \cdot ys = \text{sum-list } (\text{map } (\lambda(x, y). x * y) (\text{zip } xs \text{ } ys))$

by (*auto simp: dotprod-def sum-list-sum-nth lessThan-atLeast0*)

lemma *dotprod-commute*:

assumes $\text{length } xs = \text{length } ys$

shows $xs \cdot ys = ys \cdot xs$

using *assms* **by** (*auto simp: dotprod-def mult.commute*)

lemma *dotprod-Nil* [*simp*]: $[] \cdot [] = 0$

by (*simp add: dotprod-def*)

lemma *dotprod-Cons* [*simp*]:

$(x \# xs) \cdot (y \# ys) = x * y + xs \cdot ys$

unfolding *dotprod-def* **and** *length-Cons* **and** *min-Suc-Suc* **and** *sum.lessThan-Suc-shift*
by *auto*

lemma *dotprod-1-right* [*simp*]:

$xs \cdot \text{replicate } (\text{length } xs) \ 1 = \text{sum-list } xs$

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

lemma *dotprod-0-right* [*simp*]:

$xs \cdot \text{zeroes } (\text{length } xs) = 0$

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

lemma *dotprod-unit* [*simp*]:

assumes $\text{length } a = n$

and $k < n$

shows $a \cdot (\text{zeroes } n)[k := zk] = a ! k * zk$

using *assms* **by** (*induct a arbitrary: k n*) (*auto split: nat.splits*)

lemma *dotprod-gt0*:

assumes $\text{length } x = \text{length } y$ **and** $\exists i < \text{length } y. x ! i > 0 \wedge y ! i > 0$

shows $x \cdot y > 0$

using *assms* **by** (*induct x y rule: list-induct2*) (*fastforce simp: nth-Cons split: nat.splits*)+

lemma *dotprod-gt0D*:
assumes $\text{length } x = \text{length } y$
and $x \cdot y > 0$
shows $\exists i < \text{length } y. x ! i > 0 \wedge y ! i > 0$
using *assms* **by** (*induct x y rule: list-induct2*) (*auto simp: Ex-less-Suc2*)

lemma *dotprod-gt0-iff* [*iff*]:
assumes $\text{length } x = \text{length } y$
shows $x \cdot y > 0 \iff (\exists i < \text{length } y. x ! i > 0 \wedge y ! i > 0)$
using *assms* **and** *dotprod-gt0D* **and** *dotprod-gt0* **by** *blast*

lemma *dotprod-append*:
assumes $\text{length } a = \text{length } b$
shows $(a @ x) \cdot (b @ y) = a \cdot b + x \cdot y$
using *assms* **by** (*induct a b rule: list-induct2*) *auto*

lemma *dotprod-le-take*:
assumes $\text{length } a = \text{length } b$
and $k \leq \text{length } a$
shows $\text{take } k a \cdot \text{take } k b \leq a \cdot b$
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 $\text{length } a = \text{length } b$
and $k \leq \text{length } a$
shows $\text{drop } k a \cdot \text{drop } k b \leq a \cdot b$
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 $\text{length } x = \text{length } y$
shows $x \cdot y = 0 \iff (\forall i < \text{length } y. x ! i = 0 \vee y ! i = 0)$
using *assms* **by** (*metis dotprod-gt0-iff neq0-conv*)

lemma *dotprod-eq-0-iff*:
assumes $\text{length } x = \text{length } a$
and $0 \notin \text{set } a$
shows $x \cdot a = 0 \iff (\forall e \in \text{set } x. e = 0)$
using *assms* **by** (*fastforce simp: in-set-conv-nth*)

lemma *dotprod-eq-nonzero-iff*:
assumes $a \cdot x = b \cdot y$ **and** $\text{length } x = \text{length } a$ **and** $\text{length } y = \text{length } b$
and $0 \notin \text{set } a$ **and** $0 \notin \text{set } b$
shows $\text{nonzero } x \iff \text{nonzero } y$
using *assms* **by** (*auto simp: nonzero-iff*) (*metis dotprod-commute dotprod-eq-0-iff neq0-conv*)+

lemma *eq-0-iff*:

$xs = \text{zeroes } n \iff \text{length } xs = n \wedge (\forall x \in \text{set } xs. x = 0)$

using *in-set-replicate* [*of - n 0*] **and** *replicate-eqI* [*of xs n 0*] **by** *auto*

lemma *not-nonzero-iff*: $\neg \text{nonzero } x \iff x = \text{zeroes } (\text{length } x)$

by (*auto simp: nonzero-iff replicate-length-same eq-0-iff*)

lemma *neq-0-iff'*:

$xs \neq \text{zeroes } n \iff \text{length } xs \neq n \vee (\exists x \in \text{set } xs. x > 0)$

by (*auto simp: eq-0-iff'*)

lemma *dotprod-pointwise-le*:

assumes $\text{length } as = \text{length } xs$

and $i < \text{length } as$

shows $as ! i * xs ! i \leq as \cdot xs$

proof –

have $as \cdot xs = (\sum i < \min (\text{length } as) (\text{length } xs). as ! i * xs ! i)$

by (*simp add: dotprod-def*)

then show *?thesis*

using *assms* **by** (*auto intro: member-le-sum*)

qed

lemma *replicate-dotprod*:

assumes $\text{length } y = n$

shows $\text{replicate } n \ x \cdot y = x * \text{sum-list } y$

proof –

have $x * (\sum i < \text{length } y. y ! i) = (\sum i < \text{length } y. x * y ! i)$

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* :: $\text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{bool}$ ($- / \leq_v$ - [51, 51] 50)

where

$xs \leq_v ys \iff \text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. xs ! i \leq ys ! i)$

definition *less* :: $\text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{bool}$ ($- / <_v$ - [51, 51] 50)

where

$xs <_v ys \iff xs \leq_v ys \wedge \neg ys \leq_v xs$

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?*]: $\text{length } xs = \text{length } ys \implies \forall i < \text{length } xs. xs ! i \leq ys ! i$
 $\implies xs \leq_v ys$

by (*auto simp: less-eq-def*)

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

lemma *le-list-update* [*simp*]:
assumes $xs \leq_v ys$ **and** $i < \text{length } ys$ **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*: $x \# xs \leq_v y \# ys \iff x \leq y \wedge xs \leq_v ys$
by (*auto simp add: less-eq-def nth-Cons split: nat.splits*)

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

lemma *le-append*:
assumes $\text{length } xs = \text{length } vs$
shows $xs @ ys \leq_v vs @ ws \iff xs \leq_v vs \wedge ys \leq_v ws$
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*:
 $(x \# xs) <_v (y \# ys) \iff \text{length } xs = \text{length } ys \wedge (x \leq y \wedge xs <_v ys \vee x < y \wedge xs \leq_v ys)$
by (*simp add: less-def less-eq-def All-less-Suc2*) (*auto dest: leD*)

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

lemma *less-length* [*dest*]:
assumes $x <_v y$
shows $\text{length } x = \text{length } y$
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 $\text{length } xs = \text{length } vs$
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** $ys \leq_v ws \mid xs \leq_v vs$ **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 $\langle sum-list \ xs < sum-list \ ys \rangle$
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 $\text{length } b = \text{length } w$
shows $v \cdot b \leq w \cdot b$
using *assms* **by** (*simp add: dotprod-le-right dotprod-commute le-length*)

lemma *dotprod-le*:

assumes $x \leq_v u$ **and** $y \leq_v v$
and $\text{length } y = \text{length } x$ **and** $\text{length } v = \text{length } u$
shows $x \cdot y \leq u \cdot v$
using *assms* **by** (*metis dotprod-le-left dotprod-le-right le-length le-trans*)

lemma *dotprod-less-left*:

assumes $\text{length } b = \text{length } w$
and $0 \notin \text{set } b$
and $v <_v w$
shows $v \cdot b < w \cdot b$

proof –

have $\text{length } v = \text{length } w$ **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 $\text{length } y = \text{length } v$
and $x @ y \leq_v w @ v$
shows $y @ x \leq_v v @ w$

proof –

have $\text{length } w = \text{length } x$ **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 $\text{length } y = \text{length } v$
shows $y @ x \leq_v v @ w \longleftrightarrow x @ y \leq_v w @ v$
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 $x <_v (\text{zeroes } n)[i := b]$
shows $x ! i < b \wedge (\forall j < n. j \neq i \longrightarrow x ! j = 0)$

proof

show $x ! i < b$

using *assms less-def* **by** *fastforce*
next
have $x \leq_v (\text{zeroes } n)[i := b]$ **by** (*simp add: assms order-vec.less-imp-le*)
then show $\forall j < n. j \neq i \longrightarrow x ! j = 0$ **by** (*auto simp: less-eq-def*)
qed

lemma *le-sum-list-mono*:
assumes $xs \leq_v ys$
shows $\text{sum-list } xs \leq \text{sum-list } ys$
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 $u \leq_v y$
and $\text{sum-list } u < \text{sum-list } y$
shows $\exists i < \text{length } y. u ! i < y ! i$
proof –
have $\text{length } u = \text{length } y$ **and** $\forall i < \text{length } y. u ! i \leq y ! i$
using $\langle u \leq_v y \rangle$ **by** (*auto simp: less-eq-def*)
then show *?thesis*
using $\langle \text{sum-list } u < \text{sum-list } y \rangle$
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 $\text{sum-list } v < \text{sum-list } w$
using *assms*
proof –
have $\text{length } v = \text{length } w$
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* :: $\text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{nat}$
where
 $\text{maxne0 } x a =$
(if length x = length a \wedge $(\exists i < \text{length } a. x ! i \neq 0)$
then Max {a ! i | i. i < length a \wedge x ! i \neq 0}
else 0)

lemma *maxne0-le-Max*:
 $\text{maxne0 } x a \leq \text{Max } (\text{set } a)$

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-distrib 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 ≤_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  $\forall xi \in \text{set } x. xi \leq \text{Max } (\text{set } y)$ 
  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:
   $\forall x \in \text{set } xs. x \leq b \implies \neg x <_v \text{replicate } (\text{length } xs) b \implies xs = \text{replicate } (\text{length } xs) b$ 
  by (induct xs) (auto simp: less-Cons)

```

```

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

```

```

lemma le-take:
  assumes  $x \leq_v y$  and  $i \leq \text{length } x$  shows  $\text{take } i x \leq_v \text{take } i y$ 
  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  $\{(x, y). x <_v y\} \subseteq \text{measure } \text{sum-list}$ 
  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  $\Rightarrow$  nat list  $\Rightarrow$  nat list (infixl  $-_v$  65)
  where
     $w -_v v = \text{map } (\lambda i. w ! i - v ! i) [0 ..< \text{length } w]$ 

```

```

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*: $\text{map } f [Suc\ m\ ..<\ Suc\ n] = \text{map } (f \circ Suc) [m\ ..<\ n]$

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 $\text{length } w = \text{length } v$

shows $w -_v v = \text{map } (\lambda(x, y). x - y) (\text{zip } w\ v)$

using *assms by (induct rule: list-induct2) simp-all*

lemma *vdiff-dotprod-distr*:

assumes $\text{length } b = \text{length } w$

and $v \leq_v w$

shows $(w -_v v) \cdot b = w \cdot b - v \cdot b$

proof –

have $\text{length } v = \text{length } w$ **and** $\forall i < \text{length } w. v ! i \leq w ! i$

using *assms less-eq-def by auto*

then show *?thesis*

using $\langle \text{length } b = \text{length } w \rangle$

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 $v \leq_v u$

shows $\text{sum-list } (u -_v v) = \text{sum-list } u - \text{sum-list } v$

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 $v \leq_v w$

and $\text{length } v = \text{length } x$

shows $v -_v x \leq_v w$

using *assms by (auto simp add: less-eq-def vdiff-def)*

lemma *mods-with-vec*:

```

assumes  $v <_v w$ 
  and  $0 \notin \text{set } b$ 
  and  $\text{length } b = \text{length } w$ 
  and  $(v \cdot b) \bmod a = (w \cdot b) \bmod a$ 
shows  $((w -_v v) \cdot b) \bmod a = 0$ 
proof –
  have  $*$ :  $v \cdot b < w \cdot b$ 
    using dotprod-less-left and assms by blast
  have  $v \leq_v w$ 
    using assms by auto
  from vdiff-dotprod-distr [OF assms( $\beta$ ) this]
  have  $((w -_v v) \cdot b) \bmod a = (w \cdot b - v \cdot b) \bmod a$ 
    by simp
  also have  $\dots = 0 \bmod a$ 
    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  $0 \notin \text{set } b$ 
    and  $\text{length } b = \text{length } w$ 
    and  $(b \cdot v) \bmod a = (b \cdot w) \bmod a$ 
  shows  $(b \cdot (w -_v v)) \bmod a = 0$ 
  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

$xs <_{lex} ys \equiv (xs, ys) \in \text{lex less-than}$

definition *rlex* (**infix** $<_{rlex}$ *50*)

where

$xs <_{rlex} ys \longleftrightarrow \text{rev } xs <_{lex} \text{rev } ys$

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

$\text{rev } xs \leq_v \text{rev } ys \longleftrightarrow xs \leq_v ys$

proof –

```

{ fix  $i$  assume  $i: i < \text{length } ys$  and [simp]:  $\text{length } xs = \text{length } ys$ 
  and  $\forall i < \text{length } ys. \text{rev } xs ! i \leq \text{rev } ys ! i$ 
  then have  $\text{rev } xs ! (\text{length } ys - i - 1) \leq \text{rev } ys ! (\text{length } ys - i - 1)$  by auto
  then have  $xs ! i \leq ys ! i$  using  $i$  by (auto simp: rev-nth) }
then show ?thesis by (auto simp: less-eq-def rev-nth)

```

qed

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

$rev\ xs <_v rev\ ys \iff xs <_v ys$
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 $\neg ys <_{lex} xs$

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 $\neg ys <_{rlex} xs$

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 $\neg xs <_{lex} ys$
shows $ys = xs \wedge us <_{lex} vs$
using *assms(2,1,3)*
by (*induct xs ys rule: list-induct2*) *auto*

lemma *rlex-Cons*:
 $x \# xs <_{rlex} y \# ys \longleftrightarrow xs <_{rlex} ys \vee ys = xs \wedge x < y$ (**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*:
 $\neg x <_{rlex} x$
by (*induct x*) (*auto simp: rlex-def dest: lex-append-rightD*)

1.5 Code Equations

fun *exists2*
where
 $exists2\ d\ P\ []\ [] \longleftrightarrow False$
 $| exists2\ d\ P\ (x\#\!xs)\ (y\#\!ys) \longleftrightarrow P\ x\ y \vee exists2\ d\ P\ xs\ ys$
 $| exists2\ d\ P\ -\ - \longleftrightarrow d$

lemma *not-le-code [code-unfold]*: $\neg xs \leq_v ys \longleftrightarrow exists2\ True\ (>) xs\ ys$
by (*induct True (>) :: nat \Rightarrow nat \Rightarrow 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 $lcm\ a\ b\ div\ b \leq a$
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 $lcm\ a\ b\ \text{div}\ a \leq b$
by (*metis lcm.commute lcm-div-le*)

lemma *lcm-div-gt-0*:

fixes $a :: nat$
assumes $a > 0$ **and** $b > 0$
shows $lcm\ a\ b\ \text{div}\ a > 0$
proof –
have $lcm\ a\ b = (a * b)\ \text{div}\ (gcd\ a\ b)$
using *lcm-nat-def* **by** *blast*
moreover **have** $\dots > 0$
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 $sum\text{-list}\ (u[i := Suc\ (u\ !\ i)]) = Suc\ (sum\text{-list}\ u)$
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** $\forall x \in A. x < n$
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 $(\exists x \in set\ xs. x\ \text{mod}\ n = 0) \vee$
 $(\exists i < length\ xs. \exists j < length\ xs. i \neq j \wedge (xs\ !\ i)\ \text{mod}\ n = (xs\ !\ j)\ \text{mod}\ n)$
proof –
let $?f = \lambda x. x\ \text{mod}\ n$ **and** $?X = set\ xs$
have $*$: $\forall x \in ?f\ ' ?X. x < n$ **using** $\langle n > 0 \rangle$ **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  $\exists x \in \text{set } xs. x \bmod n = 0$  by (metis imageE lessThan-iff)
    then show ?thesis ..
  next
    assume  $\neg$  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: $a_1x_1 + \dots + a_mx_m = b_1y_1 + \dots + b_ny_n$

```

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

```

```

abbreviation m  $\equiv$  length a
abbreviation n  $\equiv$  length b

```

— The set of all solutions.

```

definition Solutions :: (nat list  $\times$  nat list) set
where

```

$$\text{Solutions} = \{(x, y). a \cdot x = b \cdot y \wedge \text{length } x = m \wedge \text{length } y = n\}$$

lemma in-Solutions-iff:

$$(x, y) \in \text{Solutions} \iff \text{length } x = m \wedge \text{length } y = n \wedge a \cdot x = b \cdot y$$

by (auto simp: Solutions-def)

— The set of pointwise minimal solutions.

```

definition Minimal-Solutions :: (nat list  $\times$  nat list) set
where

```

$$\text{Minimal-Solutions} = \{(x, y) \in \text{Solutions}. \text{nonzero } x \wedge \neg (\exists (u, v) \in \text{Solutions}. \text{nonzero } u \wedge u @ v <_v x @ y)\}$$

```

definition dij :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat

```

```

where
  dij i j = lcm (a ! i) (b ! j) div (a ! i)

```

definition $eij :: nat \Rightarrow nat \Rightarrow nat$

where

$$eij\ i\ j = lcm\ (a\ !\ i)\ (b\ !\ j)\ div\ (b\ !\ j)$$

definition $sij :: nat \Rightarrow nat \Rightarrow (nat\ list \times nat\ list)$

where

$$sij\ i\ j = ((zeroes\ m)[i := dij\ i\ j], (zeroes\ n)[j := eij\ i\ j])$$

2.1 Further Constraints on Minimal Solutions

definition $Ej :: nat \Rightarrow nat\ list \Rightarrow nat\ set$

where

$$Ej\ j\ x = \{ eij\ i\ j - 1 \mid i. i < length\ x \wedge x\ !\ i \geq dij\ i\ j \}$$

definition $Di :: nat \Rightarrow nat\ list \Rightarrow nat\ set$

where

$$Di\ i\ y = \{ dij\ i\ j - 1 \mid j. j < length\ y \wedge y\ !\ j \geq eij\ i\ j \}$$

definition $Di' :: nat \Rightarrow nat\ list \Rightarrow nat\ set$

where

$$Di'\ i\ y = \{ dij\ i\ (j + length\ b - length\ y) - 1 \mid j. j < length\ y \wedge y\ !\ j \geq eij\ i\ (j + length\ b - length\ y) \}$$

lemma *Ej-take-subset*:

$$Ej\ j\ (take\ k\ x) \subseteq Ej\ j\ x$$

by (*auto simp: Ej-def*)

lemma *Di-take-subset*:

$$Di\ i\ (take\ l\ y) \subseteq Di\ i\ y$$

by (*auto simp: Di-def*)

lemma *Di'-drop-subset*:

$$Di'\ i\ (drop\ l\ y) \subseteq Di'\ i\ y$$

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 - ($\lambda i. eij\ i\ j - 1$) ' {0 ..< length x}]]*) (*auto simp: Ej-def*)

lemma *finite-Di*:

$$finite\ (Di\ i\ y)$$

by (*rule finite-subset [of - ($\lambda j. dij\ i\ j - 1$) ' {0 ..< length y}]]*) (*auto simp: Di-def*)

lemma *finite-Di'*:

$$finite\ (Di'\ i\ y)$$

by (*rule finite-subset [of - ($\lambda j. dij\ i\ (j + length\ b - length\ y) - 1$) ' {0 ..< length y}]]*) (*auto simp: Di'-def*)

definition $max-y :: nat\ list \Rightarrow nat \Rightarrow nat$

where

$max-y\ x\ j = (if\ j < n \wedge Ej\ j\ x \neq \{\} \text{ then } Min\ (Ej\ j\ x) \text{ else } Max\ (set\ a))$

definition $max-x :: nat\ list \Rightarrow nat \Rightarrow nat$

where

$max-x\ y\ i = (if\ i < m \wedge Di\ i\ y \neq \{\} \text{ then } Min\ (Di\ i\ y) \text{ else } Max\ (set\ b))$

definition $max-x' :: nat\ list \Rightarrow nat \Rightarrow nat$

where

$max-x'\ y\ i = (if\ i < m \wedge Di'\ i\ y \neq \{\} \text{ then } Min\ (Di'\ i\ y) \text{ else } Max\ (set\ b))$

lemma *Min-Ej-le*:

assumes $j < n$

and $e \in Ej\ j\ x$

and $length\ x \leq m$

shows $Min\ (Ej\ j\ x) \leq Max\ (set\ a)$ (**is** $?m \leq -$)

proof –

have $?m \in Ej\ j\ x$

using *assms* **and** *finite-Ej* **and** *Min-in* **by** *blast*

then obtain i **where**

$i: ?m = eij\ i\ j - 1\ i < length\ x\ x!\ i \geq dij\ i\ j$

by (*auto simp: Ej-def*)

have $lcm\ (a!\ i)\ (b!\ j)\ div\ b!\ j \leq a!\ i$ **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 $e \in Di\ i\ y$

and $length\ y \leq n$

shows $Min\ (Di\ i\ y) \leq Max\ (set\ b)$ (**is** $?m \leq -$)

proof –

have $?m \in Di\ i\ y$

using *assms* **and** *finite-Di* **and** *Min-in* **by** *blast*

then obtain j **where**

$j: ?m = dij\ i\ j - 1\ j < length\ y\ y!\ j \geq eij\ i\ j$

by (*auto simp: Di-def*)

have $lcm\ (a!\ i)\ (b!\ j)\ div\ a!\ i \leq b!\ j$ **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 $e \in Di' i y$
and $length\ y \leq n$
shows $Min\ (Di' i y) \leq Max\ (set\ b)$ (**is** $?m \leq -$)
proof –
have $?m \in Di' i y$
using *assms and finite-Di' and Min-in* **by** *blast*
then obtain j **where**
 $j: ?m = dij\ i\ (j + length\ b - length\ y) - 1\ j < length\ y\ y!\ j \geq eij\ i\ (j + length\ b - length\ y)$
by (*auto simp: Di'-def*)
then have $j + length\ b - length\ y < length\ b$ **using** *assms* **by** *auto*
moreover
have $lcm\ (a!\ i)\ (b!\ (j + length\ b - length\ y))\ div\ a!\ i \leq b!\ (j + length\ b - length\ y)$ **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 $length\ x \leq m$
shows $max-y\ x\ j \leq max-y\ (take\ k\ x)\ j$
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 $length\ y \leq n$
shows $max-x\ y\ i \leq max-x\ (take\ l\ y)\ i$
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 $length\ y \leq n$
shows $max-x'\ y\ i \leq max-x'\ (drop\ l\ y)\ i$
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 $Solutions \equiv hlde-ops.Solutions$

abbreviation $Minimal-Solutions \equiv hlde-ops.Minimal-Solutions$

abbreviation $dij \equiv hlde-ops.dij$

abbreviation $eij \equiv hlde-ops.eij$

abbreviation $sij \equiv hld\text{-ops}.sij$

declare $hld\text{-ops}.dij\text{-def}$ [code]
declare $hld\text{-ops}.eij\text{-def}$ [code]
declare $hld\text{-ops}.sij\text{-def}$ [code]

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

lemma *Minimal-Solutions-imp-Solutions*: $(x, y) \in \text{Minimal-Solutions } a \ b \implies (x, y) \in \text{Solutions } a \ b$
by (*auto simp: hld\text{-ops}.Minimal-Solutions-def*)

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

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

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

then obtain u and v where $(u, v) \in \text{Solutions } a \ b$ and nonzero u
and $uv: u @ v <_v x @ y$ by *blast*
then obtain u' and v' where $(u', v') \in \text{Minimal-Solutions } a \ b$
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*:
 $(x, y) \in \text{Minimal-Solutions } a \ b \implies \text{length } x = \text{length } a \wedge \text{length } y = \text{length } b$
by (*auto simp: hlde-ops.Minimal-Solutions-def hlde-ops.in-Solutions-iff*)

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

lemma *Minimal-Solutions-sym*:
assumes $0 \notin \text{set } a$ and $0 \notin \text{set } b$
shows $(xs, ys) \in \text{Minimal-Solutions } a \ b \longrightarrow (ys, xs) \in \text{Minimal-Solutions } b \ a$
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*: $0 \notin \text{set } a$ $0 \notin \text{set } b$
begin

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

lemma *Minimal-Solutions-min*:
assumes $(x, y) \in \text{Minimal-Solutions}$
and $u @ v <_v x @ y$
and $a \cdot u = b \cdot v$
and [*simp*]: $\text{length } u = m$
and *non0*: nonzero $(u @ v)$
shows *False*
proof –
 have [*simp*]: $\text{length } v = n$ using *assms* by (*force dest: less-appendD Minimal-Solutions-length*)
 have $(u, v) \in \text{Solutions}$ using $\langle a \cdot u = b \cdot v \rangle$ 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  $(x, y) \in \text{Solutions}$ 
    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

- $\forall i < \text{length } u. u ! i \leq y ! i$ and
- $0 < \text{sum-list } u \leq a_k$.

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 ::  $\text{nat list} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat list}$ 
  where
    inc y i u =
      (if  $i < \text{length } y$  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  $(\lambda(y, i, u). \text{max } (\text{length } y) (\text{length } u) - i)$ ) auto

declare inc.simps [simp del]

```

Starting from the 0-vector produce us by iteratively incrementing with respect to y .

```

definition huets-us ::  $\text{nat list} \Rightarrow \text{nat} \Rightarrow \text{nat list}$  (u 1000)
  where
     $\mathbf{u } y i = ((\text{inc } y 0) \rightsquigarrow \text{Suc } i) (\text{zeroes } (\text{length } y))$ 

```

```

lemma huets-us-simps [simp]:
   $\mathbf{u } y 0 = \text{inc } y 0 (\text{zeroes } (\text{length } y))$ 
   $\mathbf{u } y (\text{Suc } i) = \text{inc } y 0 (\mathbf{u } y i)$ 
  by (auto simp: huets-us-def)

```

```

lemma length-inc [simp]:  $\text{length } (\text{inc } y i u) = \text{length } u$ 
  by (induct y i u rule: inc.induct) (simp add: inc.simps)

```

```

lemma length-us [simp]:
   $\text{length } (\mathbf{u } y i) = \text{length } y$ 

```

by (induct i) (simp-all)

inc produces vectors that are pointwise smaller than *y*

lemma *inc-le*:

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

lemma *us-le*:

assumes $\text{length } y > 0$
shows $u \ y \ i \leq_v y$
using *assms* **by** (induct i) (auto simp: inc-le le-length)

lemma *sum-list-inc-le*:

$u \leq_v y \implies \text{sum-list } (\text{inc } y \ i \ u) \leq \text{sum-list } y$
by (induct y i u rule: inc.induct)
(auto simp: inc.simps intro: le-sum-list-mono)

lemma *sum-list-inc-gt0*:

assumes $\text{sum-list } u > 0$ **and** $\text{length } y = \text{length } u$
shows $\text{sum-list } (\text{inc } y \ i \ u) > 0$
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 $\text{length } u = \text{length } y$ **and** $i < \text{length } y$ **and** $y \ ! \ i > 0$ **and** $j \leq i$
shows $\text{sum-list } (\text{inc } y \ j \ u) > 0$
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 $\text{sum-list } y \neq 0$
shows $0 < \text{sum-list } (u \ y \ i)$
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 $\text{length } u = \text{length } y$
shows $\text{sum-list } (\text{inc } y \ i \ u) \leq \text{sum-list } u + 1$

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

lemma *sum-list-us-le*:
 $sum-list (\mathbf{u} \ y \ i) \leq i + 1$
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 $sum-list (\mathbf{u} \ y \ i) \leq k$
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 $\exists j \geq i. j < length \ y \wedge u ! j < y ! j$
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 $sum-list (\mathbf{u} \ y \ i) = i + 1$
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*: $length \ u = length \ y \implies u \leq_v inc \ y \ i \ u$
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 $\mathbf{u} \ y \ i \leq_v \mathbf{u} \ y \ j$
using *assms*

proof (*induct j - i arbitrary: j i*)
case (*Suc n*)
then show *?case*
by (*simp add: Suc.premis 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 $\forall x \in \text{set } xs. x \leq \text{maxne0 } ys \ b$ (**is** $\forall x \in \text{set } xs. x \leq ?m$)
proof (*rule ccontr*)
assume $\neg ?thesis$
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 $\langle a ! k * ?m < a ! k * xs ! k \rangle$ **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:  $\forall u \in \text{set Seq. length } u = \text{length } ys$ 
  by (simp add: Seq-def)
have prop-3:  $\forall u \in \text{set Seq. } u \leq_v ys$ 
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:  $\forall u \in \text{set Seq. sum-list } u > 0$ 
  by (metis Seq-def sl-ys-g-ak gr-implies-not-zero imageE
    set-map sum-list-us-gt0)
have prop-4-2:  $\forall u \in \text{set Seq. sum-list } u \leq a ! k$ 
  by (simp add: Seq-def sum-list-us-bounded)
have prop-5:  $\exists u. \text{length } u = \text{length } ys \wedge u \leq_v ys \wedge \text{sum-list } u > 0 \wedge \text{sum-list } u \leq a ! k$ 
  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  $\exists u \in Us. b \cdot u \text{ mod } a ! k = 0$ 
proof (rule ccontr)
  assume neg-th:  $\neg$  ?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) ( $\exists x \in \text{set Seq-p. } x \text{ mod } (a ! k) = 0$ ) |
    (not-0) ( $\exists i < \text{length Seq-p. } \exists j < \text{length Seq-p. } i \neq j \wedge$ 
       $(\text{Seq-p } ! i) \text{ mod } (a!k) = (\text{Seq-p } ! j) \text{ mod } (a!k)$ )
  using list-mod-cases[of Seq-p] Seq-p-def ak-n0 by auto force
  then show False
proof (cases)
  case eq-0
  have  $\exists u \in \text{set Seq. } b \cdot u \text{ mod } a ! k = 0$ 
    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:  $i < \text{length Seq-p } j < \text{length Seq-p } i \neq j$ 
    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 ≡ v -v w⟩ ass less-vec-sum-list-less by auto
  have prop-4-2: sum-list u ≤ a!k
  proof -
    have u ≤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-neg-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 ≤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 ≤v ys sum-list u > 0 sum-list u ≤ 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 ≤ maxne0 u b * sum-list u
    by (simp add: maxne0-times-sum-list-gt-dotprod u-b-len)
  also have ... ≤ ?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 ≠ 0
  proof -
    have ∃ e ∈ set u. e ≠ 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 ≠ 0⟩ by blast

```

then show *?thesis*
using $\langle 0 < b \cdot u \rangle$ *zk* **by** *auto*
qed
define *z* **where**
z-def: $z = (\text{zeroes } (\text{length } xs))[k := zk]$
then have *zk-zk*: $z ! k = zk$
by (*auto simp add*: $\langle k < \text{length } xs \rangle$)
have $\text{length } z = \text{length } xs$
using *assms z-def* $\langle k < \text{length } xs \rangle$ **by** *auto*
then have *bu-eq-akzk*: $b \cdot u = a ! k * z ! k$
by (*simp add*: $\langle b \cdot u = zk * a ! k \rangle$ *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* $\langle k < \text{length } xs \rangle$ *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 $(z, u) \in \text{Solutions}$
by (*auto simp add*: *bu-eq-akzk* *in-Solutions-iff* *z-def* *u-b-len* $\langle k < \text{length } xs \rangle$ *len*)
moreover have *nonzero* *z*
using $\langle \text{length } z = \text{length } xs \rangle$ **and** $\langle zk \neq 0 \rangle$ **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 $(xs, ys) \in \text{Minimal-Solutions}$
shows $(\forall x \in \text{set } xs. x \leq \text{maxne0 } ys \ b) \wedge (\forall y \in \text{set } ys. y \leq \text{maxne0 } xs \ a)$
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 $(x, y) \in \text{Minimal-Solutions}$
shows $\forall i < \text{length } x. x ! i \leq \text{Max } (\text{set } b)$ **and** $\forall j < \text{length } y. y ! j \leq \text{Max } (\text{set } a)$
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*:

$\text{Minimal-Solutions} = \{(x, y) \in \text{Solutions}.$
 $(x, y) \neq (\text{zeroes } m, \text{zeroes } n) \wedge$
 $x \leq_v \text{replicate } m (\text{Max } (\text{set } b)) \wedge$
 $y \leq_v \text{replicate } n (\text{Max } (\text{set } a)) \wedge$
 $\neg (\exists (u, v) \in \text{Solutions}. \text{nonzero } u \wedge u @ v <_v x @ y)\}$

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 \times nat list) set
where
Special-Solutions = {sij i j | i j. i < m \wedge j < n}

lemma *dij-neq-0*:
assumes i < m
and j < n
shows dij i j \neq 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 \neq 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*:
 $x \in \text{Special-Solutions} \implies x \in \text{Solutions}$
by (auto simp: in-Solutions-iff Special-Solutions-def sij-def dij-def eij-def)

lemma *Special-Solutions-in-Minimal-Solutions*:
assumes (x, y) \in Special-Solutions
shows (x, y) \in Minimal-Solutions
proof (intro Minimal-SolutionsI')
show (x, y) \in 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 $\neg (\exists (u, v) \in \text{Minimal-Solutions}. u @ v <_v x @ y)$
proof
assume $\exists (u, v) \in \text{Minimal-Solutions}. u @ v <_v x @ y$
then obtain u **and** v **where** $uv: (u, v) \in \text{Minimal-Solutions}$ **and** $u @ v <_v x @ y$
and $[simp]: \text{length } u = m \text{ length } v = n$
and $\text{nonzero } u$ **by** $(\text{auto simp: Minimal-Solutions-def in-Solutions-iff})$
then consider $u <_v x$ **and** $v \leq_v y \mid v <_v y$ **and** $u \leq_v x$ **by** $(\text{auto elim: less-append-cases})$
then show False
proof (cases)
case 1
then obtain i **and** j **where** $ij: i < m \ j < n$
and $\text{less-dij}: u ! i < \text{dij } i \ j$
and $u \leq_v (\text{zeroes } m)[i := \text{dij } i \ j]$
and $v \leq_v (\text{zeroes } n)[j := \text{eij } i \ j]$
using assms **by** $(\text{auto simp: Special-Solutions-def sij-def unit-less})$
then have $u: u = (\text{zeroes } m)[i := u ! i]$ **and** $v: v = (\text{zeroes } n)[j := v ! j]$
by $(\text{auto simp: less-eq-def list-eq-iff-nth-eq})$
 $(\text{metis le-zero-eq length-list-update length-replicate rep-upd-unit})+$
then have $u ! i > 0$ **using** $\langle \text{nonzero } u \rangle$ **and** ij
by $(\text{metis gr-implies-not0 neq0-conv unit-less zero-less})$

define c **where** $c = a ! i * u ! i$
then have $ac: a ! i \ \text{dvd} \ c$ **by** simp

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

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

moreover have *nonzero v*
using $\langle \text{nonzero } u \rangle$ **and** $\langle (u, v) \in \text{Minimal-Solutions} \rangle$
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*: $b ! j \text{ dvd } c$ **by** *simp*

have $a \cdot u = b \cdot v$ **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*: $a ! i \text{ dvd } c$ **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 $c < \text{lcm } (a ! i) (b ! j)$ **by** (*auto simp: eij-def c-def*)
moreover have $\text{lcm } (a ! i) (b ! j) \text{ dvd } c$ **by** (*simp add: ac bc*)
moreover have $c > 0$ **using** $\langle v ! j > 0 \rangle$ **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 $(x, y) \in \text{Solutions} - \text{Special-Solutions}$
and *ij*: $i < m \ j < n$
and $x ! i \geq dij i j$ **and** $y ! j \geq eij i j$
shows $(x, y) \notin \text{Minimal-Solutions}$
proof
assume *min*: $(x, y) \in \text{Minimal-Solutions}$
moreover have *sij* $i j \in \text{Solutions}$
using *ij* **by** (*intro Special-Solutions-in-Solutions*) (*auto simp: Special-Solutions-def*)
moreover have (*case sij i j of (u, v) $\Rightarrow u @ v$*) $<_v x @ y$
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) $\Rightarrow \text{nonzero } u$*)
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* $xs\ ys \longleftrightarrow (\forall x \in \text{set } xs. x \leq \text{maxne0 } ys\ b)$

definition *cond-B* $x \longleftrightarrow$
 $(\forall k \leq m. \text{take } k\ a \cdot \text{take } k\ x \leq b \cdot \text{map } (\text{max-y } (\text{take } k\ x))\ [0 ..< n])$

definition *boundr* $x\ y \longleftrightarrow (\forall j < n. y\ !\ j \leq \text{max-y } x\ j)$

definition *cond-D* $x\ y \longleftrightarrow (\forall l \leq n. \text{take } l\ b \cdot \text{take } l\ y \leq a \cdot x)$

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

definition *subdprodr* $y \longleftrightarrow$
 $(\forall l \leq n. \text{take } l\ b \cdot \text{take } l\ y \leq a \cdot \text{map } (\text{max-x } (\text{take } l\ y))\ [0 ..< m])$

definition *subdprodl* $x\ y \longleftrightarrow (\forall k \leq m. \text{take } k\ a \cdot \text{take } k\ x \leq b \cdot y)$

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

lemma *boundr*:

assumes *min*: $(x, y) \in \text{Minimal-Solutions}$

and $(x, y) \notin \text{Special-Solutions}$

shows *boundr* $x\ y$

proof (*unfold boundr-def, intro allI impI*)

fix j

assume *ass*: $j < n$

have *ln*: $m = \text{length } x \wedge n = \text{length } y$

using *assms Minimal-Solutions-def in-Solutions-iff min* **by** *auto*

have *is-sol*: $(x, y) \in \text{Solutions}$

using *assms Minimal-Solutions-def min* **by** *auto*

have *j-less-l*: $j < n$

using *assms ass le-less-trans* **by** *linarith*

consider (*notemp*) $Ej\ j\ x \neq \{\}$ | (*empty*) $Ej\ j\ x = \{\}$

by *blast*

then show $y\ !\ j \leq \text{max-y } x\ j$

proof (*cases*)

case *notemp*

have *max-y-def*: $\text{max-y } x\ j = \text{Min } (Ej\ j\ x)$

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'*: $\forall e \in Ej\ j\ x. (\exists i < \text{length } x. x\ !\ i \geq \text{dij } i\ j \wedge \text{eij } i\ j - 1 = e)$

using *Ej-def [of j x]* **by** *auto*

```

then have  $\exists i < \text{length } x. x ! i \geq \text{dij } i j \wedge \text{eij } i j - 1 = \text{Min } (Ej j x)$ 
  using notemp Min-in e-def' fin-e by blast
then obtain i where
  i:  $i < \text{length } x \wedge x ! i \geq \text{dij } i j \wedge \text{eij } i j - 1 = \text{Min } (Ej j x)$ 
  by blast
show ?thesis
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  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  $(u, v) \Rightarrow u @ v \leq_v x @ y$ 
    by (force simp: max-y-def)
  then have cs: case sij i j of  $(u, v) \Rightarrow u @ v <_v x @ y$ 
  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:  $\text{sij } i j = (u, v) \wedge u @ v <_v x @ y$ 
  by blast
  have dij-gt0:  $\text{dij } i j > 0$ 
    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  $i < \text{length } (\text{zeroes } m)$  by (simp add: i(1) ln)
    then show  $\exists i \in \text{set } u. i \neq 0$ 
      by (metis (no-types) Pair-inject dij-gt0 set-update-memI sij-def u-v(1))
  neq0-conv)
  qed
  then have sij i j  $\in \text{Solutions}$ 
    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  $\forall y \in \text{set } y. y \leq \text{Max } (\text{set } a)$ 
    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:  $(x, y) \in \text{Minimal-Solutions}$ 
    and  $(x, y) \notin \text{Special-Solutions}$ 
  shows boundl x y
proof (unfold boundl-def, intro allI impI)

```

```

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

```

```

    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  $\forall x \in \text{set } x. x \leq \text{Max } (\text{set } b)$ 
  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 $(x, y) \in \text{Solutions}$
 shows *cond-D* $x\ y$
 using assms and dotprod-le-take by (auto simp: cond-D-def in-Solutions-iff)

lemma *Solution-imp-subprodl*:
 assumes $(x, y) \in \text{Solutions}$
 shows *subprodl* $x\ y$
 using assms and dotprod-le-take
 by (auto simp: subprodl-def in-Solutions-iff) metis

theorem *conds*:
 assumes *min*: $(x, y) \in \text{Minimal-Solutions}$
 shows *cond-A*: *cond-A* $x\ y$
 and *cond-B*: $(x, y) \notin \text{Special-Solutions} \implies \text{cond-B } x$
 and $(x, y) \notin \text{Special-Solutions} \implies \text{boundr } x\ y$
 and *cond-D*: *cond-D* $x\ y$
 and *subprodr*: $(x, y) \notin \text{Special-Solutions} \implies \text{subprodr } y$
 and *subprodl*: *subprodl* $x\ y$
proof –
 have *sol*: $a \cdot x = b \cdot y$ and *ln*: $m = \text{length } x \wedge n = \text{length } y$
 using *min* by (auto simp: Minimal-Solutions-def in-Solutions-iff)
 then have $\forall i < m. x ! i \leq \text{maxne0 } y\ b$
 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 $(x, y) \notin \text{Special-Solutions} \implies \text{cond-B } x$
proof (unfold *cond-B-def*, intro allI impI)
 fix k assume *non-spec*: $(x, y) \notin \text{Special-Solutions}$ and $k: k \leq m$
 from k have *take k a* \cdot *take k x $\leq a \cdot x$
 using *dotprod-le-take ln* by blast
 also have $\dots = b \cdot y$ by *fact*
 also have *map-b-dot-p*: $\dots \leq b \cdot \text{map } (\text{max-}y\ x) [0..<n]$ (is $\leq - b \cdot ?nt$)
 using *non-spec* and *less-eq-def* and *ln* and *boundr* and *min**

by (*fastforce intro!: dotprod-le-right simp: boundr-def*)
 also have $\dots \leq b \cdot \text{map } (\text{max-}y \text{ (take } k \text{ } x)) [0..<n]$ (*is - \leq - \cdot ?t*)
 proof -
 have $\forall j < n. ?nt!j \leq ?t!j$
 using *min and ln and max-y-le-take and k by auto*
 then have $?nt \leq_v ?t$
 using *less-eq-def by auto*
 then show *?thesis*
 by (*simp add: dotprod-le-right*)
 qed
 finally show $\text{take } k \text{ } a \cdot \text{take } k \text{ } x \leq b \cdot \text{map } (\text{max-}y \text{ (take } k \text{ } x)) [0..<n]$
 by (*auto simp: cond-B-def*)
 qed

show $(x, y) \notin \text{Special-Solutions} \implies \text{subdprodr } y$
 proof (*unfold subdprodr-def, intro allI impI*)
 fix l assume *non-spec: (x, y) \notin Special-Solutions and $l: l \leq n$*
 from l have $\text{take } l \text{ } b \cdot \text{take } l \text{ } y \leq b \cdot y$
 using *dotprod-le-take ln by blast*
 also have $\dots = a \cdot x$ by (*simp add: sol*)
 also have $\text{map-}b\text{-dot-}p: \dots \leq a \cdot \text{map } (\text{max-}x \text{ } y) [0..<m]$ (*is - \leq - $a \cdot$?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 $\dots \leq a \cdot \text{map } (\text{max-}x \text{ (take } l \text{ } y)) [0..<m]$ (*is - \leq - \cdot ?t*)
 proof -
 have $\forall i < m. ?nt ! i \leq ?t ! i$
 using *min and ln and max-x-le-take and l by auto*
 then have $?nt \leq_v ?t$
 using *less-eq-def by auto*
 then show *?thesis*
 by (*simp add: dotprod-le-right*)
 qed
 finally show $\text{take } l \text{ } b \cdot \text{take } l \text{ } y \leq a \cdot \text{map } (\text{max-}x \text{ (take } l \text{ } y)) [0..<m]$
 by (*auto simp: cond-B-def*)
 qed

show $(x, y) \notin \text{Special-Solutions} \implies \text{boundr } x \text{ } y$
 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 $u \leq_v x$
 shows $Ej \text{ } j \text{ } u \subseteq Ej \text{ } j \text{ } x$

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 $\text{length } x \leq m$

shows $\text{max-y } u \ j \geq \text{max-y } x \ j$

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 $\text{Di } i \ v \subseteq \text{Di } i \ y$

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 $\text{length } y \leq n$

shows $\text{max-x } v \ i \geq \text{max-x } y \ i$

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*:

$\text{sorted-wrt } P \ xs \implies \text{sorted-wrt } P \ (\text{filter } Q \ xs)$

by (induct *xs*) (auto)

lemma *sorted-wrt-map-mono*:

assumes $\text{sorted-wrt } Q \ xs$

and $\bigwedge x \ y. Q \ x \ y \implies P \ (f \ x) \ (f \ y)$

shows $\text{sorted-wrt } P \ (\text{map } f \ xs)$

using *assms* by (induct *xs*) (auto)

lemma *sorted-wrt-concat-map-map*:

assumes $\text{sorted-wrt } Q \ xs$

and $\text{sorted-wrt } Q \ ys$

and $\bigwedge a \ x \ y. Q \ x \ y \implies P \ (f \ x \ a) \ (f \ y \ a)$

and $\bigwedge x \ y \ u \ v. x \in \text{set } xs \implies y \in \text{set } xs \implies Q \ u \ v \implies P \ (f \ x \ u) \ (f \ y \ v)$

shows $\text{sorted-wrt } P \ [f \ x \ y \ . \ y \leftarrow ys, \ x \leftarrow xs]$

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 $\bigwedge x. x \in \text{set } xs \implies \text{sorted-wrt } P$ (*map* h (f x))
and $\bigwedge x y u v. P$ (h x) (h y) $\implies x \in \text{set } xs \implies y \in \text{set } xs \implies u \in \text{set } (f$ $x)$
 $\implies v \in \text{set } (f$ $y) \implies P$ (h u) (h v)
shows *sorted-wrt* P (*concat* (*map* (*map* $h \circ f$) 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*:
 $xs \neq [] \implies \text{sorted-wrt } P$ $xs \implies \text{sorted-wrt } P$ (*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 ($x \# xs$) = $x \# \text{filter}$ (P x) (*minimize-wrt* P xs)

lemma *minimize-wrt-subset*: set (*minimize-wrt* P xs) $\subseteq \text{set } xs$
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 $xs \implies \text{sorted-wrt } Q$ (*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 $y \implies \neg Q$ y x
and *sorted-wrt* Q xs
and $x \in \text{set}$ (*minimize-wrt* P xs)
and $\neg P$ y x **and** Q y x **and** $y \in \text{set } xs$ **and** $y \neq x$

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

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

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

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

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

```

lemma minimize-wrt-append:
  assumes  $\forall x \in \text{set } xs. \forall y \in \text{set } (xs @ ys). P y x$ 
  shows minimize-wrt  $P (xs @ ys) = xs @ \text{filter } (\lambda y. \forall x \in \text{set } xs. P x y)$  (minimize-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:  $xs \neq [] \implies f (xs ! 0) \neq [] \implies \text{concat } (\text{map } f xs) ! 0 =$ 
 $f (xs ! 0) ! 0$ 
by (induct xs) (auto simp: nth-append)

```

3.1 Reverse-Lexicographic Enumeration of Potential Minimal Solutions

```

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

```

```

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

```

```

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

```

```

lemma less-imp-rlex2:  $\neg (\text{case } x \text{ of } (x, y) \Rightarrow \lambda(u, v). \neg x @ y <_v u @ v) y \implies$ 
 $x <_{\text{rlex2}} y$ 
  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 ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list list}$ 
  where
     $\text{gen } B 0 = [[]]$ 
     $|\ \text{gen } B (\text{Suc } n) = [x \# xs \ . \ xs \leftarrow \text{gen } B n, x \leftarrow [0 ..< B + 1]]$ 

```

```

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

```

```

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

```

definition $minimize = minimize-wrt (\lambda(x, y) (u, v). \neg x @ y <_v u @ v)$

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 $set-gen: set (gen B n) = \{xs. length xs = n \wedge (\forall i < n. xs ! i \leq B)\}$ (is -
= ?A n)

proof (induct n)

case [simp]: (Suc n)

{ fix xs assume $xs \in ?A (Suc n)$

then have $xs \in set (gen B (Suc n))$

by (cases xs) (force simp: All-less-Suc2)+ }

then show ?case by (auto simp: less-Suc-eq-0-disj)

qed simp

abbreviation $gen2 A B m n \equiv [(x, y) . y \leftarrow gen B n, x \leftarrow gen A m]$

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 $gen-ne [simp]: gen B n \neq []$ by (induct n) auto

lemma $gen2-ne: gen2 A B m n \neq []$ 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 $check-generate a b \equiv check a b (generate (Max (set b)) (Max (set a)) (length a) (length b))$

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 $in-tl-gen2: x \in set (tl (gen2 A B m n)) \implies x \in set (gen2 A B m n)$

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*:

$set\ (gen2\ A\ B\ m\ n) = \{(x, y). length\ x = m \wedge length\ y = n \wedge (\forall i < m. x\ !\ i \leq A) \wedge (\forall j < n. y\ !\ j \leq B)\}$
by (*auto simp: set-gen*)

lemma *gen2-unique*:

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

lemma *zeroes-ni-tl-gen2*:

$(zeroes\ m, zeroes\ n) \notin set\ (tl\ (gen2\ A\ B\ m\ n))$

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*:

$set\ (generate\ A\ B\ m\ n) = \{(x, y). (x, y) \neq (zeroes\ m, zeroes\ n) \wedge (x, y) \in set\ (gen2\ A\ B\ m\ n)\}$

proof

show $set\ (generate\ A\ B\ m\ n) \subseteq \{(x, y). (x, y) \neq (zeroes\ m, zeroes\ n) \wedge (x, y) \in set\ (gen2\ A\ B\ m\ n)\}$
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 $set\ (gen2\ A\ B\ m\ n) = set\ (generate\ A\ B\ m\ n) \cup \{(zeroes\ m, zeroes\ n)\}$

by (*metis Un-empty-right generate-def Un-insert-right gen2-ne calculation list.exhaust-sel list.simps(15)*)

ultimately show $\{(x, y). (x, y) \neq (zeroes\ m, zeroes\ n) \wedge (x, y) \in set\ (gen2\ A\ B\ m\ n)\}$

$\subseteq set\ (generate\ A\ B\ m\ n)$

by *blast*

qed

lemma *set-check-generate*:

$set\ (check-generate\ a\ b) = \{(x, y). (x, y) \neq (zeroes\ (length\ a), zeroes\ (length\ b)) \wedge length\ x = length\ a \wedge length\ y = length\ b \wedge a \cdot x = b \cdot y \wedge (\forall i < length\ a. x\ !\ i \leq Max\ (set\ b)) \wedge (\forall j < length\ b. y\ !\ j \leq Max\ (set\ a))\}$

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*)) =
 $\{(x, y) \in \text{set } (\text{check-generate } a \ b). \neg (\exists (u, v) \in \text{set } (\text{check-generate } a \ b). u \ @ \ v <_v x \ @ \ y)\}$
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*) =
 $\{(x, y) \in \text{set } (\text{check-generate } a \ b). \neg (\exists (u, v) \in \text{set } (\text{check-generate } a \ b). u \ @ \ v <_v x \ @ \ y)\}$
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 \subseteq *set* (*solutions* *a b*)
proof (*rule subrelI*)
let *?A* = *Max* (*set* *b*) **and** *?B* = *Max* (*set* *a*)
fix *x y* **assume** *min*: $(x, y) \in \text{Minimal-Solutions}$
then have $(x, y) \in \text{set } (\text{check } a \ b \ (\text{generate } ?A \ ?B \ m \ n))$
by (*auto simp: Minimal-Solutions-alt-def set-check-generate less-eq-def in-Solutions-iff*)
moreover have $\forall (u, v) \in \text{set } (\text{check } a \ b \ (\text{generate } ?A \ ?B \ m \ n)). \neg u \ @ \ v <_v x \ @ \ y$
using *min* **and** *no0*
by (*auto simp: check-def set-generate neq-0-iff' set-gen nonzero-iff dest!: Minimal-Solutions-min*)
ultimately show $(x, y) \in \text{set } (\text{solutions } a \ b)$ **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*) \subseteq *Minimal-Solutions*
proof (*rule subrelI*)
fix *x y* **assume** *sol*: $(x, y) \in \text{set } (\text{solutions } a \ b)$
show $(x, y) \in \text{Minimal-Solutions}$
proof (*rule Minimal-SolutionsI'*)
show $*$: $(x, y) \in \text{Solutions}$
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  $\neg (\exists (u, v) \in \text{Minimal-Solutions}. u @ v <_v x @ y)$ 
proof
  have min-cg:  $(x, y) \in \text{set } (\text{minimize } (\text{check-generate } a \ b))$ 
  using sol by (auto simp: solutions-def)
  note * = in-minimize-wrt-False [OF - sorted-wrt-check-generate min-cg
  [unfolded minimize-def]]

  assume  $\exists (u, v) \in \text{Minimal-Solutions}. u @ v <_v x @ y$ 
  then obtain u and v where  $(u, v) \in \text{Minimal-Solutions}$  and less:  $u @ v <_v$ 
 $x @ y$  by blast
  then have  $(u, v) \in \text{set } (\text{solutions } a \ b)$  by (auto intro: solutions-complete
  [THEN subsetD])
  then have  $(u, v) \in \text{set } (\text{check-generate } a \ b)$ 
  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]:  $\text{set } (\text{solutions } a \ b) = \text{Minimal-Solutions}$ 
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*: $(\forall i \leq \text{Suc } n. P \ i) \longleftrightarrow P \ 0 \wedge (\forall i \leq n. P \ (\text{Suc } i))$
by (*metis less-Suc-eq-0-disj nat-less-le order-refl*)

lemma *concat-map-filter-filter*:

assumes $\bigwedge x. x \in \text{set } xs \implies \neg Q \ x \implies \text{filter } P \ (f \ x) = []$

shows $\text{concat } (\text{map } (\text{filter } P \circ f) \ (\text{filter } Q \ xs)) = \text{concat } (\text{map } (\text{filter } P \circ f) \ xs)$

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

lemma *filter-pairs-conj*:

$\text{filter } (\lambda(x, y). P \ x \ y \wedge Q \ y) \ xs = \text{filter } (\lambda(x, y). P \ x \ y) \ (\text{filter } (Q \circ \text{snd}) \ xs)$

by (*induct xs*) *auto*

lemma *concat-map-filter*:

$\text{concat } (\text{map } f \ (\text{filter } P \ xs)) = \text{concat } (\text{map } (\lambda x. \text{if } P \ x \ \text{then } f \ x \ \text{else } []) \ xs)$

```

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* $A B a b = [(xs, ys). ys \leftarrow \text{alls } B b, xs \leftarrow \text{alls } A a]$

lemma *alls2-ne* [*simp*]:

$\text{alls2 } A B a b \neq []$

by (*auto simp: alls2-def*) (*metis alls-ne list.set-intros(1) neq-Nil-conv surj-pair*)

lemma *set-alls2*:

$\text{set } (\text{alls2 } A B a b) = \{((x, s), (y, t)). \text{length } x = \text{length } a \wedge \text{length } y = \text{length } b$

\wedge

$(\forall i < \text{length } a. x ! i \leq A) \wedge (\forall j < \text{length } b. y ! j \leq B) \wedge s = a \cdot x \wedge t = b \cdot y\}$

by (*auto simp: alls2-def set-alls*)

lemma *alls2-nth0* [*simp*]: $\text{alls2 } A B as bs ! 0 = ((\text{zeroes } (\text{length } as), 0), (\text{zeroes } (\text{length } bs), 0))$

by (*auto simp: alls2-def concat-map-nth0*)

lemma *alls2-Cons-tl-conv*: $\text{alls2 } A B as bs =$

$((\text{zeroes } (\text{length } as), 0), (\text{zeroes } (\text{length } bs), 0)) \# \text{tl } (\text{alls2 } A B as bs)$

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

$\text{gen2 } A B a b \equiv \text{map } (\lambda(x, y). (\text{fst } x, \text{fst } y)) (\text{alls2 } A B a b)$

lemma *sorted-wrt-gen2*:

$\text{sorted-wrt } (<_{\text{rlex2}}) (\text{gen2 } A B a b)$

apply (*rule sorted-wrt-map-mono* [of $\lambda(x, y) (u, v). (\text{fst } x, \text{fst } y) <_{\text{rlex2}} (\text{fst } u, \text{fst } v)$])

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

$\text{generate}' A B a b = \text{tl } (\text{map } (\lambda(x, y). (\text{fst } x, \text{fst } y)) (\text{alls2 } A B a b))$

lemma *sorted-wrt-generate'*:

$\text{sorted-wrt } (<_{\text{rlex2}}) (\text{generate}' A B a b)$

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*]: $gen2\ m\ n\ b\ c\ \neq\ []$ **by** *auto*

lemma *in-generate'*: $x \in set\ (generate'\ m\ n\ c\ b) \implies x \in set\ (gen2\ m\ n\ c\ b)$
unfolding *generate'-def* **by** (*rule list.set-sel*) *simp*

definition *cond-cons* $P = (\lambda(ys, s).\ case\ ys\ of\ [] \Rightarrow True\ |\ ys \Rightarrow P\ ys\ s)$

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
 $suffs\ P\ as\ (xs,\ s) \longleftrightarrow$
 $length\ xs = length\ as \wedge$
 $s = as \cdot xs \wedge$
 $(\forall i \leq length\ xs.\ cond-cons\ P\ (drop\ i\ xs,\ drop\ i\ as \cdot drop\ i\ xs))$

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

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

lemma *suffs-Cons*:
 $suffs\ P\ (a\ \#\ as)\ (x\ \#\ xs,\ s) \longleftrightarrow$
 $s = a * x + as \cdot xs \wedge cond-cons\ P\ (x\ \#\ xs,\ s) \wedge suffs\ P\ as\ (xs,\ as \cdot xs)$
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* \leq *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* \times *nat list*) *list*

where

special-solutions = [*sij a b i j* . *i* \leftarrow [*0* ..< *length a*], *j* \leftarrow [*0* ..< *length b*]]

definition *big-e* :: *nat list* \Rightarrow *nat* \Rightarrow *nat list*

where

big-e *x j* = map (λi . *eij a b i j* - 1) (filter (λi . *x* ! *i* \geq *dij a b i j*) [*0* ..< *length x*])

definition *big-d* :: *nat list* \Rightarrow *nat* \Rightarrow *nat list*

where

big-d *y i* = map (λj . *dij a b i j* - 1) (filter (λj . *y* ! *j* \geq *eij a b i j*) [*0* ..< *length y*])

definition *big-d'* :: *nat list* \Rightarrow *nat* \Rightarrow *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 (λj . *dij a b i* (*j* + *k*) - 1) (filter (λj . *y* ! *j* \geq *eij a b i* (*j* + *k*)) [*0* ..< *length y*]))))

definition *max-y-impl* :: *nat list* \Rightarrow *nat* \Rightarrow *nat*

where

max-y-impl *x j* =
 (if *j* < *length b* \wedge *big-e* *x j* \neq [] then *Min* (set (*big-e* *x j*))
 else *Max* (set *a*))

definition *max-x-impl* :: *nat list* \Rightarrow *nat* \Rightarrow *nat*

where

max-x-impl *y i* =
 (if *i* < *length a* \wedge *big-d* *y i* \neq [] then *Min* (set (*big-d* *y i*))
 else *Max* (set *b*))

definition *max-x-impl'* :: *nat list* \Rightarrow *nat* \Rightarrow *nat*

where

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

definition $cond-a :: nat list \Rightarrow nat list \Rightarrow bool$

where

$cond-a xs ys \longleftrightarrow (\forall x \in set xs. x \leq maxne0 ys b)$

definition $cond-b :: nat list \Rightarrow bool$

where

$cond-b xs \longleftrightarrow (\forall k \leq length a.$
 $take k a \cdot take k xs \leq b \cdot (map (max-y-impl (take k xs)) [0 ..< length b]))$

definition $boundr-impl :: nat list \Rightarrow nat list \Rightarrow bool$

where

$boundr-impl x y \longleftrightarrow (\forall j < length b. y ! j \leq max-y-impl x j)$

definition $cond-d :: nat list \Rightarrow nat list \Rightarrow bool$

where

$cond-d xs ys \longleftrightarrow (\forall l \leq length b. take l b \cdot take l ys \leq a \cdot xs)$

definition $subdprodr-impl :: nat list \Rightarrow bool$

where

$subdprodr-impl ys \longleftrightarrow (\forall l \leq length b.$
 $take l b \cdot take l ys \leq a \cdot map (max-x-impl (take l ys)) [0 ..< length a])$

definition $subdprodl-impl :: nat list \Rightarrow nat list \Rightarrow bool$

where

$subdprodl-impl x y \longleftrightarrow (\forall k \leq length a. take k a \cdot take k x \leq b \cdot y)$

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

definition $static-bounds$

where

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

definition $check-cond =$

$(\lambda(x, y). static-bounds x y \wedge a \cdot x = b \cdot y \wedge boundr-impl x y \wedge subdprodl-impl x y \wedge subdprodr-impl y)$

definition $check' = filter check-cond$

definition $non-special-solutions =$

(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 ($<_{rl_{ex2}}$) (*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 \leq *length b* \implies *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* \leq *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*) *subprodr-impl* [*simp*]: *subprodr-impl a b y* = *subprodr y*

using *max-x-impl* **by** (*auto simp: subprodr-impl-def subprodr-def presburger+*)

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

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

lemma (in *hlde*) *check* [*simp*]:
check' $a\ b =$
filter ($\lambda(x, y). \text{static-bounds}\ a\ b\ x\ y \wedge a \cdot x = b \cdot y \wedge \text{boundr}\ x\ y \wedge$
subdprodl $x\ y \wedge$
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*: $u \leq_v x\ v \leq_v y$
and *len*: $\text{length}\ x = m\ \text{length}\ y = n$
shows *cond-B* $x \implies \text{cond-B}\ u$
and *boundr* $x\ y \implies \text{boundr}\ u\ v$
and $a \cdot u = b \cdot v \implies \text{cond-D}\ x\ y \implies \text{cond-D}\ u\ v$
and $a \cdot u = b \cdot v \implies \text{subdprodl}\ x\ y \implies \text{subdprodl}\ u\ v$
and *subdprodr* $y \implies \text{subdprodr}\ v$

proof –

assume *B*: *cond-B* x
have $\text{length}\ u = m$ **using** *len* **and** *le* **by** (*auto*)
show *cond-B* u
proof (*unfold* *cond-B-def*, *intro* *allI* *impI*)
fix k
assume $k: k \leq m$
moreover **have** $*$: $\text{take}\ k\ u \leq_v \text{take}\ k\ x$ **if** $k \leq m$ **for** k
using *le* **and** *that* **by** (*intro* *le-take*) (*auto* *simp*: *len*)
ultimately **have** $\text{take}\ k\ a \cdot \text{take}\ k\ u \leq \text{take}\ k\ a \cdot \text{take}\ k\ x$
by (*intro* *dotprod-le-right*) (*auto* *simp*: *len*)
also **have** $\dots \leq b \cdot \text{map}\ (\text{max-y}\ (\text{take}\ k\ x))\ [0..<n]$
using k **and** *B* **by** (*auto* *simp*: *cond-B-def*)
also **have** $\dots \leq b \cdot \text{map}\ (\text{max-y}\ (\text{take}\ k\ u))\ [0..<n]$
using *le-imp-max-y-ge* [*OF* $*$] [*OF* k]
using k **by** (*auto* *simp*: *len* *intro!*: *dotprod-le-right* *less-eqI*)
finally **show** $\text{take}\ k\ a \cdot \text{take}\ k\ u \leq b \cdot \text{map}\ (\text{max-y}\ (\text{take}\ k\ u))\ [0..<n]$.

qed

next

assume *subdprodr*: *subdprodr* y
have $\text{length}\ v = n$ **using** *len* **and** *le* **by** (*auto*)
show *subdprodr* v
proof (*unfold* *subdprodr-def*, *intro* *allI* *impI*)
fix l
assume $l: l \leq n$
moreover **have** $*$: $\text{take}\ l\ v \leq_v \text{take}\ l\ y$ **if** $l \leq n$ **for** l

```

    using le and that by (intro le-take) (auto simp: len)
  ultimately have  $\text{take } l \ b \cdot \text{take } l \ v \leq \text{take } l \ b \cdot \text{take } l \ y$ 
    by (intro dotprod-le-right) (auto simp: len)
  also have  $\dots \leq a \cdot \text{map } (\text{max-}x \ (\text{take } l \ y)) \ [0..<m]$ 
    using l and subdprodr by (auto simp: subdprodr-def)
  also have  $\dots \leq a \cdot \text{map } (\text{max-}x \ (\text{take } l \ v)) \ [0..<m]$ 
    using le-imp-max-x-ge [OF * [OF l]]
    using l by (auto simp: len intro!: dotprod-le-right less-eqI)
  finally show  $\text{take } l \ b \cdot \text{take } l \ v \leq a \cdot \text{map } (\text{max-}x \ (\text{take } l \ v)) \ [0..<m]$ .
qed
next
  assume C: boundr x y
  show boundr u v
    using le-imp-max-y-ge [OF  $\langle u \leq_v x \rangle$ ] 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)  $\subseteq$  Special-Solutions
    by (auto simp: Special-Solutions-def special-solutions-def) (blast)
  moreover have Special-Solutions  $\subseteq$  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) =  $\{(x, y). x \leq_v \text{replicate } (\text{length } a) \ A \wedge y \leq_v \text{replicate } (\text{length } b) \ B\}$ 
  (is ?L = ?R)
proof (intro equalityI subrelI)
  fix xs ys assume  $(xs, ys) \in ?R$ 
  then have  $\forall x \in \text{set } xs. x \leq A$  and  $\forall y \in \text{set } ys. y \leq B$ 
    and length xs = length a and length ys = length b
    by (auto simp: less-eq-def in-set-conv-nth)
  then have  $((xs, a \cdot xs), (ys, b \cdot ys)) \in \text{set } (\text{alls2 } A \ B \ a \ b)$  by (auto simp: set-alls2)
  then have  $(\lambda(x, y). (\text{fst } x, \text{fst } y)) ((xs, a \cdot xs), (ys, b \cdot ys)) \in (\lambda(x, y). (\text{fst } x, \text{fst } y)) \text{' set } (\text{alls2 } A \ B \ a \ b)$ 
    by (intro imageI)

```

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

lemma *set-gen2'*:
 $(\lambda(x, y). (fst\ x, fst\ y)) \text{ ` } set\ (alls2\ A\ B\ a\ b) =$
 $\{(x, y). x \leq_v replicate\ (length\ a)\ A \wedge y \leq_v replicate\ (length\ b)\ B\}$
using *set-gen2* **by** *simp*

lemma (*in hlde*) *in-non-special-solutions*:
assumes $(x, y) \in set\ (non-special-solutions\ a\ b)$
shows $(x, y) \in Solutions$
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 $generate\ A\ B\ a\ b\ !\ i \neq generate\ A\ B\ a\ b\ !\ j$
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 $gen2\ A\ B\ a\ b\ !\ i \neq gen2\ A\ B\ a\ b\ !\ j$
using *sorted-wrt-nth-less [OF sorted-wrt-gen2 assms]*
by (*auto simp: rlex2-irrefl*)

lemma *zeroes-ni-generate'*:
 $(zeroes\ (length\ a), zeroes\ (length\ b)) \notin set\ (generate'\ A\ B\ a\ b)$
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'*:
 $set\ (generate'\ A\ B\ a\ b) =$
 $\{(x, y). (x, y) \neq (zeroes\ (length\ a), zeroes\ (length\ b)) \wedge (x, y) \in set\ (gen2\ A\ B\ a\ b)\}$
proof
show $set\ (generate'\ A\ B\ a\ b)$
 $\subseteq \{(x, y). (x, y) \neq (zeroes\ (length\ a), zeroes\ (length\ b)) \wedge (x, y) \in set\ (gen2\ A\ B\ a\ b)\}$
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 $\text{set } (\text{gen2 } A \ B \ a \ b) = \text{set } (\text{tl } (\text{gen2 } A \ B \ a \ b)) \cup \{(\text{zeroes } (\text{length } a), \text{ zeroes } (\text{length } b))\}$
by (*metis Un-empty-right Un-insert-right gen2-ne calculation list.exhaust-sel list.simps(15)*)
ultimately show $\{(x, y). (x, y) \neq (\text{zeroes } (\text{length } a), \text{ zeroes } (\text{length } b)) \wedge (x, y) \in \text{set } (\text{gen2 } A \ B \ a \ b)\}$
 $\subseteq \text{set } (\text{generate}' \ A \ B \ a \ b)$
unfolding *generate'-def* **by** *blast*
qed

lemma *set-generate''*:
 $\text{set } (\text{generate}' \ A \ B \ a \ b) =$
 $\{(x, y). (x, y) \neq (\text{zeroes } (\text{length } a), \text{ zeroes } (\text{length } b)) \wedge x \leq_v \text{replicate } (\text{length } a) \ A \wedge y \leq_v \text{replicate } (\text{length } b) \ B\}$
by (*simp add: set-generate' set-gen2'*)

lemma (*in hlde*) *zeroes-ni-non-special-solutions*:
shows $(\text{zeroes } m, \text{ zeroes } n) \notin \text{set } (\text{non-special-solutions } a \ b)$

proof –

define *All-lex* **where**
 $\text{All-lex}: \text{All-lex} = \text{gen2 } (\text{Max } (\text{set } b)) \ (\text{Max } (\text{set } a)) \ a \ b$
define *z* **where** $z: z = (\text{zeroes } m, \text{ zeroes } n)$
have $\text{set } (\text{non-special-solutions } a \ b) \subseteq \text{set } (\text{tl } (\text{All-lex}))$
by (*auto simp: All-lex generate'-def non-special-solutions-def minimize-def dest: minimize-wrtD*)
moreover have $z \notin \text{set } (\text{tl } (\text{All-lex}))$
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 $\text{set } (\text{solve } a \ b) \subseteq \text{Minimal-Solutions}$

proof (*rule subrelI*)

let $?a = \text{Max } (\text{set } a)$ **and** $?b = \text{Max } (\text{set } b)$

fix $x \ y$

assume $\text{ass}: (x, y) \in \text{set } (\text{solve } a \ b)$

then consider $(x, y) \in \text{set } (\text{special-solutions } a \ b) \mid (x, y) \in \text{set } (\text{non-special-solutions } a \ b)$

unfolding *solve-def* **and** *set-append* **by** *blast*

then show $(x, y) \in \text{Minimal-Solutions}$

proof (*cases*)

case *1*

then have $(x, y) \in \text{Special-Solutions}$

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 convA/B/x/convD/y ∧
  subprodl x y ∧
  subprodr y]
case 2
then have conds: ∀ e ∈ set x. e ≤ Max (set b) boundr x y
  subprodl x y subprodr 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 convA/B/x/convD/y
convA/D/x/y ∧
  subprodl x y ∧
  subprodr 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
  subprodl u v subprodr 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  $\Rightarrow$  nat  $\Rightarrow$  bool
  and B :: nat
  assumes bound:  $\bigwedge x xs s. x > B \Longrightarrow C (x \# xs) s = False$ 
  and cond-antimono:  $\bigwedge x x' xs s s'. C (x \# xs) s \Longrightarrow x' \leq x \Longrightarrow s' \leq s \Longrightarrow C (x' \# xs) s'$ 
begin

```

```

function incs :: nat  $\Rightarrow$  nat  $\Rightarrow$  (nat list  $\times$  nat)  $\Rightarrow$  (nat list  $\times$  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 ( $\lambda(a, x, xs, s). B + 1 - x$ ), rule wf-measure, case-tac x > B)
  (use bound in auto)

```

```

declare incs.simps [simp del]

```

```

lemma in-incs:
  assumes (ys, t)  $\in$  set (incs a x (xs, s))
  shows length ys = length xs + 1  $\wedge$  t = s + hd ys * a  $\wedge$  tl ys = xs  $\wedge$  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]:  $x > B \Longrightarrow incs a x (xs, s) = []$ 
  by (induct a x (xs, s) rule: incs.induct) (simp add: incs.simps bound)

```

```

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

```

```

show incs a x xss = ( $\lambda(xs, s).$  filter (cond-cons C) (map ( $\lambda x.$  ( $x \# xs, s + a * x$ )) [ $x .. < B + 1$ ])) 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  $\Rightarrow$  (nat list  $\times$  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)  $\in$  set (gen-check as)
  shows length ys = length as
  using assms
proof (induct as arbitrary: ys s)
  case (Cons a as)
  have  $\exists (la, t) \in$  set (gen-check as). (ys, s)  $\in$  set (incs a 0 (la, t))
    using Cons.prem1 by auto
  moreover obtain la t where (la, t)  $\in$  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.prem1 by fastforce
  ultimately show ?case by simp
qed (auto)

```

```

lemma in-gen-check:
  assumes (xs, s)  $\in$  set (gen-check as)
  shows length xs = length as  $\wedge$  s = as  $\cdot$  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  $\forall j \leq \text{length } xs. \text{drop } j \text{ } xs \neq [] \longrightarrow C (\text{drop } j \text{ } xs) (s - \text{take } j \text{ } as \cdot \text{take } j \text{ } xs)$ 
  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 C2 B2 for C2 B2 +
  fixes C1 and B1
  assumes cond1:  $\bigwedge b \text{ ys}. \text{ys} \in \text{fst} \text{ ' set } (c2.\text{gen-check } b) \implies \text{bounded-gen-check}$ 
  (C1 b ys) (B1 b)
begin

definition generate-check a b =
  [(xs, ys). ys  $\leftarrow$  c2.gen-check b, xs  $\leftarrow$  bounded-gen-check.gen-check (C1 b (fst ys))
  a]

lemma generate-check-filter-conv:
  generate-check a b = [(xs, ys).
    ys  $\leftarrow$  filter (suffs C2 b) (alls B2 b),
    xs  $\leftarrow$  filter (suffs (C1 b (fst ys)) a) (alls (B1 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)  $\leftarrow$  alls2 (B1 b) B2 a b. suffs (C1 b (fst ys)) a xs
   $\wedge$  suffs C2 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 (C1 b (zeroes (length b))) a (zeroes (length a), 0)
  and suffs C2 b (zeroes (length b), 0)
  shows tl (generate-check a b) = [(xs, ys)  $\leftarrow$  tl (alls2 (B1 b) B2 a b). suffs (C1
  b (fst ys)) a xs  $\wedge$  suffs C2 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

```

$cond1\ ys\ []\ s \longleftrightarrow True$
 $| cond1\ ys\ (x\ \# \ xs)\ s \longleftrightarrow s \leq b \cdot ys \wedge x \leq maxne0-impl\ ys\ b$

lemma *max-x-impl'-conv:*

$i < length\ a \implies length\ y = length\ b \implies max-x-impl'\ a\ b\ y\ i = max-x-impl\ a\ b\ y\ i$
by (*auto simp: max-x-impl'-def max-x-impl-def Let-def big-d'-def big-d-def*)

fun *cond2*

where

$cond2\ []\ s \longleftrightarrow True$
 $| cond2\ (y\ \# \ ys)\ s \longleftrightarrow y \leq Max\ (set\ a) \wedge s \leq a \cdot map\ (max-x-impl'\ a\ b\ (y\ \# \ ys))\ [0\ ..<\ length\ a]$

lemma *le-imp-big-d'-subset:*

assumes $v \leq_v\ y$
shows $set\ (big-d'\ a\ b\ v\ i) \subseteq set\ (big-d'\ a\ b\ y\ i)$
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 - ($\lambda j. dij\ a\ b\ i\ (j + length\ b - length\ y) - 1$) ' {0 ..< length\ y}]]*)
(auto simp: Let-def big-d'-def)

lemma *Min-big-d'-le:*

assumes $i < length\ a$
and $big-d'\ a\ b\ y\ i \neq []$
and $length\ y \leq length\ b$
shows $Min\ (set\ (big-d'\ a\ b\ y\ i)) \leq Max\ (set\ b)\ (is\ ?m \leq -)$
proof –
have $?m \in set\ (big-d'\ a\ b\ y\ i)$
using *assms and finite-big-d' and Min-in by auto*
then obtain j **where**
 $j: ?m = dij\ a\ b\ i\ (j + length\ b - length\ y) - 1\ j < length\ y\ y!\ j \geq eij\ a\ b\ i\ (j + length\ b - length\ y)$
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 $lcm\ (a!\ i)\ (b!\ (j + length\ b - length\ y))\ div\ a!\ i \leq b!\ (j + length\ b - length\ y)$ **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  $i < \text{length } a$ 
  shows  $\text{max-x-impl}' a b v i \geq \text{max-x-impl}' a b y i$ 
  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
 $\lambda b. \text{Max (set b)}$ 
  defines c2-gen-check = c12.c2.gen-check and c2-incs = c12.c2.incs
    and c12-generate-check = c12.generate-check
proof -
  { fix  $x \text{ } xs \text{ } s$  assume  $\text{Max (set a)} < x$ 
    then have  $\text{cond2 } a \text{ } b \text{ } (x \# xs) \text{ } s = \text{False}$  by (auto) }
  note 1 = this

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

interpret c2: bounded-generate-check  $\text{cond2 } a \text{ } b \text{ } \text{Max (set a)}$  by (standard) fact+

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

  { fix  $b \text{ } ys \text{ } x \text{ } x' \text{ } xs \text{ } s \text{ } s'$  assume  $ys \in \text{fst } ' \text{set } (c2.\text{gen-check } b)$  and  $\text{cond1 } b \text{ } ys \text{ } (x \# xs) \text{ } s$ 
    and  $x' \leq x$  and  $s' \leq s$ 
    then have  $\text{cond1 } b \text{ } ys \text{ } (x' \# xs) \text{ } s'$  by auto }
  note 4 = this

  show bounded-generate-check ( $\text{cond2 } a \text{ } b$ ) ( $\text{Max (set a)}$ ) cond1 ( $\lambda b. \text{Max (set b)}$ )
    using 1 and 2 and 3 and 4 by (unfold-locales) metis+
qed

```


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

definition *fast-filter* $a\ b = \text{filter } (\text{post-cond } a\ b) (\text{map } (\lambda(x, y). (\text{fst } x, \text{fst } y)) (\text{tl } (\text{c12-generate-check } a\ b\ a\ b)))$

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 $\forall y \in \text{set } aa. y \leq \text{maxne0-impl } aaa\ b$
and *length* $aa = \text{length } a$
and $a \cdot aa = b \cdot aaa$
shows *suffs* (*cond1* $b\ aaa$) a ($aa, b \cdot aaa$)
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* $ys = \text{length } b$
shows *suffs* (*cond2* $a\ b$) b ($ys, b \cdot ys$) \longleftrightarrow
 $(\forall y \in \text{set } ys. y \leq \text{Max } (\text{set } a)) \wedge \text{subdprodr-impl } a\ b\ ys$
(is ?L \longleftrightarrow ?R)

proof
assume $*$: $?L$
then have $\forall y \in \text{set } ys. y \leq \text{Max } (\text{set } a)$
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** $l: l \leq \text{length } b$
have *take* $l\ b \cdot \text{take } l\ ys \leq b \cdot ys$
using l **and** *assms* **by** (*simp add: dotprod-le-take*)
also have $\dots \leq a \cdot \text{map } (\text{max-x-impl}'\ a\ b\ ys) [0 ..< \text{length } a]$
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: subprod1-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*]: $x > B \implies \text{incs } a \ x \ (xs, s) = []$

```

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.