Andrew's Monotone Chain Convex Hull Algorithm

Arthur Freitas Ramos
David Barros Hulak
Ruy J. G. B. de Queiroz

May 12, 2026

Abstract

This development formalizes the executable core of Andrew's monotone chain convex hull algorithm [1]. The algorithm sorts planar points lexicographically, removes duplicates, computes lower and upper hull chains by a stack scan that removes non-left turns, and concatenates the two chains with their repeated endpoints removed. The formalization proves the stack-scan turn invariant, subset and distinctness properties for the computed chains, ordered-chain facts for the lower and upper scans, a distinctness criterion for the final concatenation, length bounds, support-function invariants for the lower and upper scans, and the real-coordinate theorem that the computed output has the same convex hull as the input. The final correctness theorem states the specification explicitly: the returned vertices are input points, every input point lies in the convex hull of the returned vertices, and the returned vertices have exactly the same convex hull as the input. It also proves irredundancy of the returned vertex set: deleting any returned point changes the convex hull of the returned set. The executable algorithm is polymorphic over ordered integral domains because the scan only uses lexicographic ordering and signs of orientation determinants; the geometric theorem is stated for real coordinates because the convexity and separating-hyperplane libraries use real vector spaces. Thus the integer examples exercise the executable code, while the real examples instantiate the geometric correctness theorem. The support-function argument shows that points removed by a scan are dominated in the relevant support directions; a separating hyperplane theorem then yields convex-hull coverage, and strict supporting directions expose each returned vertex for the irredundancy result. The development proves functional correctness, but does not formalize the asymptotic running time. AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

Contents

1 Andrew's Monotone Chain Convex Hull Algorithm

theory *Andrew-Monotone-Chain*
imports
HOL−Analysis.Convex
HOL−Analysis.Convex-Euclidean-Space
HOL−Library.Product-Lexorder
HOL−Library.Sublist
begin

This theory formalizes the executable core of Andrew's monotone chain convex hull algorithm. Points are sorted lexicographically, duplicate points are removed, and two stack scans compute the lower and upper chains. The final hull is the usual concatenation of the two chains with their repeated endpoints removed.

The scan is stored internally with the top of the stack at the head of the list. Reversing the stack gives the geometric left-to-right order of a lower scan, or the right-to-left order of an upper scan.

The main correctness theorem combines executable invariants with a real-coordinate support-function argument. Since the algorithm only returns input points, convex-hull equality follows once every input point is shown to lie in the convex hull of the computed output.

The executable definitions are deliberately polymorphic over ordered integral domains: the scan only needs lexicographic ordering and signs of orientation determinants. The geometric specification below is stated for real coordinates, because Isabelle's convex-hull and separating-hyperplane theorems live in real vector spaces.

type-synonym *'a point* = *'a* × *'a*

definition *cross* ::
(*'a*::*linordered-idom*) *point* ⇒ *'a point* ⇒ *'a point* ⇒ *'a*
where
cross p q r =
(*fst q* − *fst p*) ∗ (*snd r* − *snd p*) −
(*snd q* − *snd p*) ∗ (*fst r* − *fst p*)

definition *left-turn* ::

(*'a*::*linordered-idom*) *point* ⇒ *'a point* ⇒ *'a point* ⇒ *bool*
where
left-turn p q r ⟷ *cross p q r* > *0*

definition *collinear* ::
(*'a*::*linordered-idom*) *point* ⇒ *'a point* ⇒ *'a point* ⇒ *bool*
where
collinear p q r ⟷ *cross p q r* = *0*

lemma *cross-same-left* [*simp*]: *cross p p q* = *0*
by (*simp add*: *cross-def*)

lemma *cross-same-right* [*simp*]: *cross p q q* = *0*
by (*simp add*: *cross-def*)

lemma *cross-same-outer* [*simp*]: *cross p q p* = *0*
by (*simp add*: *cross-def*)

lemma *cross-swap-outer*:
cross p q r = − *cross r q p*
by (*simp add*: *cross-def algebra-simps*)

lemma *cross-swap-last*:
cross p r q = − *cross p q r*
by (*simp add*: *cross-def algebra-simps*)

lemma *cross-cycle*:
cross p q r = *cross q r p*
by (*simp add*: *cross-def algebra-simps*)

lemma *two-cross-zero-imp-eq-middle*:
fixes *a p c q* :: *real point*
assumes *cross a p q* = *0*
and *cross p c q* = *0*
and *cross a p c* ≠ *0*
shows *q* = *p*
proof (*cases a*; *cases p*; *cases c*; *cases q*)
fix *ax ay px py cx cy qx qy* :: *real*
assume *pts*: *a* = (*ax*, *ay*) *p* = (*px*, *py*) *c* = (*cx*, *cy*) *q* = (*qx*, *qy*)
define *A* **where** *A* = *ax* − *px*
define *B* **where** *B* = *ay* − *py*
define *C* **where** *C* = *cx* − *px*
define *D* **where** *D* = *cy* − *py*
define *X* **where** *X* = *qx* − *px*
define *Y* **where** *Y* = *qy* − *py*
have *e1*: *A* ∗ *Y* − *B* ∗ *X* = *0*
using *assms*(*1*) *pts*
by (*simp add*: *cross-def A-def B-def X-def Y-def algebra-simps*)
have *e2*: *C* ∗ *Y* − *D* ∗ *X* = *0*

```
    using assms(2) pts
    by (simp add: cross-def C-def D-def X-def Y-def algebra-simps)
  have det-ne: A * D - B * C ≠ 0
    using assms(3) pts
    by (simp add: cross-def A-def B-def C-def D-def algebra-simps)
  have (A * D - B * C) * X = C * (A * Y - B * X) - A * (C * Y - D * X)
    by (simp add: algebra-simps)
  then have X-zero: X = 0
    using e1 e2 det-ne by auto
  have (A * D - B * C) * Y = D * (A * Y - B * X) - B * (C * Y - D * X)
    by (simp add: algebra-simps)
  then have Y-zero: Y = 0
    using e1 e2 det-ne by auto
  show q = p
    using pts X-zero Y-zero by (simp add: X-def Y-def)
qed

lemma cross-pos-trans-coords:
  fixes ax ay bx b-y cx cy dx d-y :: real
  assumes (ax, ay) < (bx, b-y)
    and (bx, b-y) < (cx, cy)
    and (cx, cy) < (dx, d-y)
    and 0 < cross (ax, ay) (bx, b-y) (cx, cy)
    and 0 < cross (bx, b-y) (cx, cy) (dx, d-y)
  shows 0 < cross (ax, ay) (bx, b-y) (dx, d-y)
    and 0 < cross (ax, ay) (cx, cy) (dx, d-y)
proof -
  define ux where ux = bx - ax
  define uy where uy = b-y - ay
  define vx where vx = cx - bx
  define vy where vy = cy - b-y
  define wx where wx = dx - cx
  define wy where wy = d-y - cy
  have ux-nonneg: 0 ≤ ux
    using assms(1) by (auto simp: ux-def less-prod-def')
  have vx-nonneg: 0 ≤ vx
    using assms(2) by (auto simp: vx-def less-prod-def')
  have wx-nonneg: 0 ≤ wx
    using assms(3) by (auto simp: wx-def less-prod-def')
  have cross-uv: 0 < ux * vy - uy * vx
     using assms(4) by (simp add: cross-def ux-def uy-def vx-def vy-def alge-
bra-simps)
  have cross-vw: 0 < vx * wy - vy * wx
     using assms(5) by (simp add: cross-def vx-def vy-def wx-def wy-def alge-
bra-simps)
  have ux-pos: 0 < ux
  proof (rule ccontr)
    assume ¬ 0 < ux
    then have ux-zero: ux = 0
```

```
    using ux-nonneg by simp
  have uy-pos: 0 < uy
    using assms(1) ux-zero by (auto simp: ux-def uy-def less-prod-def')
  have ux * vy - uy * vx ≤ 0
    using ux-zero uy-pos vx-nonneg by (simp add: mult-nonneg-nonneg)
  then show False
    using cross-uv by linarith
qed
have vx-pos: 0 < vx
proof (rule ccontr)
  assume ¬ 0 < vx
  then have vx-zero: vx = 0
    using vx-nonneg by simp
  have vy-pos: 0 < vy
    using assms(2) vx-zero by (auto simp: vx-def vy-def less-prod-def')
  have vx * wy - vy * wx ≤ 0
    using vx-zero vy-pos wx-nonneg by (simp add: mult-nonneg-nonneg)
  then show False
    using cross-vw by linarith
qed
have identity:
  vx * (ux * wy - uy * wx) =
    wx * (ux * vy - uy * vx) + ux * (vx * wy - vy * wx)
  by (simp add: algebra-simps)
have 0 ≤ wx * (ux * vy - uy * vx)
  using wx-nonneg cross-uv by (intro mult-nonneg-nonneg) linarith+
moreover have 0 < ux * (vx * wy - vy * wx)
  using ux-pos cross-vw by (intro mult-pos-pos)
ultimately have 0 < vx * (ux * wy - uy * wx)
  using identity by linarith
then have cross-uw: 0 < ux * wy - uy * wx
  using vx-pos by (simp add: zero-less-mult-iff)
show 0 < cross (ax, ay) (bx, b-y) (dx, d-y)
  using cross-uv cross-uw
  by (simp add: cross-def ux-def uy-def vx-def vy-def wx-def wy-def algebra-simps)
show 0 < cross (ax, ay) (cx, cy) (dx, d-y)
  using cross-uw cross-vw
  by (simp add: cross-def ux-def uy-def vx-def vy-def wx-def wy-def algebra-simps)
qed

lemma cross-pos-trans-left-coords:
  fixes ax ay bx b-y cx cy dx d-y :: real
  assumes (ax, ay) < (bx, b-y)
    and (bx, b-y) < (cx, cy)
    and (cx, cy) < (dx, d-y)
    and 0 < cross (ax, ay) (bx, b-y) (cx, cy)
    and 0 < cross (bx, b-y) (cx, cy) (dx, d-y)
  shows 0 < cross (ax, ay) (bx, b-y) (dx, d-y)
  using assms by (rule cross-pos-trans-coords(1))
```

lemma *cross-pos-trans-left*:
fixes *a b c d* :: *real point*
assumes $a < b$ **and** $b < c$ **and** $c < d$
and $0 <$ *cross a b c* **and** $0 <$ *cross b c d*
shows $0 <$ *cross a b d*
using *assms*
by (*cases a*; *cases b*; *cases c*; *cases d*; *auto intro*: *cross-pos-trans-left-coords*)

lemma *cross-pos-trans-right-coords*:
fixes *ax ay bx b-y cx cy dx d-y* :: *real*
assumes $(ax, ay) < (bx, b\text{-}y)$
and $(bx, b\text{-}y) < (cx, cy)$
and $(cx, cy) < (dx, d\text{-}y)$
and $0 <$ *cross* (ax, ay) $(bx, b\text{-}y)$ (cx, cy)
and $0 <$ *cross* $(bx, b\text{-}y)$ (cx, cy) $(dx, d\text{-}y)$
shows $0 <$ *cross* (ax, ay) (cx, cy) $(dx, d\text{-}y)$
using *assms* **by** (*rule cross-pos-trans-coords*(*2*))

lemma *cross-pos-trans-right*:
fixes *a b c d* :: *real point*
assumes $a < b$ **and** $b < c$ **and** $c < d$
and $0 <$ *cross a b c* **and** $0 <$ *cross b c d*
shows $0 <$ *cross a c d*
using *assms*
by (*cases a*; *cases b*; *cases c*; *cases d*; *auto intro*: *cross-pos-trans-right-coords*)

fun *stack-turns* :: (*'a*::*linordered-idom*) *point list* ⇒ *bool*
where
stack-turns [] = *True*
| *stack-turns* [*p*] = *True*
| *stack-turns* [*p*, *q*] = *True*
| *stack-turns* (*p* # *q* # *r* # *ps*) =
(*left-turn r q p* ∧ *stack-turns* (*q* # *r* # *ps*))

definition *chain-turns* :: (*'a*::*linordered-idom*) *point list* ⇒ *bool*
where
chain-turns ps ⟷ *stack-turns* (*rev ps*)

fun *scan-push* ::
(*'a*::*linordered-idom*) *point list* ⇒ *'a point* ⇒ *'a point list*
where
scan-push [] *p* = [*p*]
| *scan-push* [*q*] *p* = [*p*, *q*]
| *scan-push* (*q* # *r* # *st*) *p* =
(*if cross r q p* ≤ *0 then scan-push* (*r* # *st*) *p else p* # *q* # *r* # *st*)

definition *scan-stack* ::
(*'a*::*linordered-idom*) *point list* ⇒ *'a point list*

where
scan-stack ps = *foldl scan-push* [] *ps*

definition *scan-chain* ::
(*'a*::*linordered-idom*) *point list* ⇒ *'a point list*
where
scan-chain ps = *rev* (*scan-stack ps*)

definition *sorted-unique* ::
(*'a*::*linorder*) *list* ⇒ *'a list*
where
sorted-unique xs = *sorted-list-of-set* (*set xs*)

definition *lower-hull* ::
(*'a*::{*linorder*,*linordered-idom*}) *point list* ⇒ *'a point list*
where
lower-hull ps = *scan-chain* (*sorted-unique ps*)

definition *upper-hull* ::
(*'a*::{*linorder*,*linordered-idom*}) *point list* ⇒ *'a point list*
where
upper-hull ps = *scan-chain* (*rev* (*sorted-unique ps*))

definition *andrew-hull* ::
(*'a*::{*linorder*,*linordered-idom*}) *point list* ⇒ *'a point list*
where
andrew-hull ps =
(*case sorted-unique ps of*
[] ⇒ []
| [*p*] ⇒ [*p*]
| - ⇒ *butlast* (*lower-hull ps*) @ *butlast* (*upper-hull ps*))

lemma *sorted-unique-set* [*simp*]:
set (*sorted-unique xs*) = *set xs*
by (*simp add*: *sorted-unique-def*)

lemma *sorted-unique-distinct* [*simp*]:
distinct (*sorted-unique xs*)
by (*simp add*: *sorted-unique-def*)

lemma *sorted-unique-sorted* [*simp*]:
sorted (*sorted-unique xs*)
by (*simp add*: *sorted-unique-def*)

lemma *sorted-unique-Nil-iff* [*simp*]:
sorted-unique xs = [] ⟷ *xs* = []
by (*metis set-empty sorted-unique-set*)

lemma *sorted-unique-singleton-iff*:

```
  sorted-unique xs = [p] ⟷ set xs = {p}
proof
  assume sorted-unique xs = [p]
  then have set (sorted-unique xs) = {p}
    by simp
  then show set xs = {p}
    by simp
next
  assume set-xs: set xs = {p}
  have set-su: set (sorted-unique xs) = {p}
    using set-xs by simp
  have dist: distinct (sorted-unique xs)
    by simp
  show sorted-unique xs = [p]
  proof (cases sorted-unique xs)
    case Nil
    then show ?thesis using set-su by simp
  next
    case (Cons y ys)
    then have y: y = p
      using set-su by auto
    have ys = []
    proof (rule ccontr)
      assume ys ≠ []
      then obtain z zs where ys: ys = z # zs
        by (cases ys) auto
      with Cons set-su have z = p
        by auto
      with Cons ys y dist show False
        by auto
    qed
    with Cons y show ?thesis by simp
  qed
qed

lemma set-scan-push-subset:
  set (scan-push st p) ⊆ insert p (set st)
  by (induction st p rule: scan-push.induct) auto

lemma set-scan-stack-subset:
  set (scan-stack ps) ⊆ set ps
  unfolding scan-stack-def
proof (induction ps arbitrary: rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc p ps)
  have set (foldl scan-push [] (ps @ [p])) ⊆ insert p (set (foldl scan-push [] ps))
    using set-scan-push-subset by simp
```

```
  also have ... ⊆ set (ps @ [p])
    using snoc.IH by auto
  finally show ?case .
qed

lemma set-scan-chain-subset:
  set (scan-chain ps) ⊆ set ps
  using set-scan-stack-subset by (simp add: scan-chain-def)

lemma lower-hull-subset:
  set (lower-hull ps) ⊆ set ps
  using set-scan-chain-subset[of sorted-unique ps]
  by (auto simp: lower-hull-def)

lemma upper-hull-subset:
  set (upper-hull ps) ⊆ set ps
  using set-scan-chain-subset[of rev (sorted-unique ps)]
  by (auto simp: upper-hull-def)

theorem andrew-hull-subset:
  set (andrew-hull ps) ⊆ set ps
proof (cases sorted-unique ps)
  case Nil
  then have ps = []
    by simp
  then show ?thesis
    by (simp add: andrew-hull-def sorted-unique-def)
next
  case (Cons p qs)
  then show ?thesis
  proof (cases qs)
    case Nil
    with Cons have su: sorted-unique ps = [p]
      by simp
    then have set ps = {p}
      using sorted-unique-singleton-iff by blast
    moreover have andrew-hull ps = [p]
      using su by (simp add: andrew-hull-def)
    ultimately show ?thesis
      by simp
  next
    case (Cons q rs)
    have set (butlast (lower-hull ps)) ⊆ set ps
      using lower-hull-subset[of ps] by (auto dest: in-set-butlastD)
    moreover have set (butlast (upper-hull ps)) ⊆ set ps
      using upper-hull-subset[of ps] by (auto dest: in-set-butlastD)
    ultimately show ?thesis
      using Cons ‹sorted-unique ps = p # qs›
      by (simp add: andrew-hull-def)
```

qed
qed

1.1 Convex-Hull Correctness Interface

The executable algorithm is generic over ordered integral domains. The geometric convex-hull specification is stated here for real coordinates, where Isabelle's convexity library supplies the ambient real vector-space structure on products.

The predicate is the envelope specification proved for the algorithm. The first conjunct says that every returned vertex is an input point. The second conjunct is equality of convex hulls, which includes both coverage of the input by the returned envelope and the absence of any hull area outside the input hull. Thus the predicate is stronger than one-sided soundness.

definition *convex-hull-correct* ::
  real point list ⇒ *real point list* ⇒ *bool*
where
  convex-hull-correct ps hs ⟷
    set hs ⊆ *set ps* ∧ *convex hull set hs* = *convex hull set ps*

definition *convex-hull-irredundant* :: *real point list* ⇒ *bool*
where
  convex-hull-irredundant hs ⟷
    (∀ *p*∈*set hs*. *convex hull* (*set hs* − {*p*}) ≠ *convex hull set hs*)

lemma *strict-support-notin-convex-hull*:
  fixes *p v* :: *real point*
  assumes *strict*: ⋀*q*. *q* ∈ *S* ⟹ *inner v q* < *inner v p*
  shows *p* ∉ *convex hull S*
proof
  assume *p-hull*: *p* ∈ *convex hull S*
  have *convex hull S* ⊆ {*q*. *inner v q* < *inner v p*}
    by (*rule hull-minimal*) (*use strict* **in** ‹*auto simp*: *convex-halfspace-lt*›)
  then show *False*
    using *p-hull* **by** *auto*
qed

lemma *strict-support-hull-delete-ne*:
  fixes *p v* :: *real point*
  assumes *p*: *p* ∈ *S*
    and *strict*: ⋀*q*. *q* ∈ *S* − {*p*} ⟹ *inner v q* < *inner v p*
  shows *convex hull* (*S* − {*p*}) ≠ *convex hull S*
proof
  assume *eq*: *convex hull* (*S* − {*p*}) = *convex hull S*
  have *p* ∉ *convex hull* (*S* − {*p*})
    by (*rule strict-support-notin-convex-hull*[*OF strict*])
  moreover have *p* ∈ *convex hull* (*S* − {*p*})
    using *p eq* **by** (*simp add*: *hull-inc*)

ultimately show *False*
by *contradiction*
qed

theorem *andrew-hull-convex-hull-subset*:
fixes *ps* :: *real point list*
shows *convex hull set* (*andrew-hull ps*) ⊆ *convex hull set ps*
by (*rule hull-mono*) (*rule andrew-hull-subset*)

lemma *convex-hull-eq-from-mutual-inclusion*:
fixes *xs ys* :: ('*a*::*real-vector*) *list*
assumes *set xs* ⊆ *convex hull set ys*
and *set ys* ⊆ *convex hull set xs*
shows *convex hull set xs* = *convex hull set ys*
proof (*rule subset-antisym*)
show *convex hull set xs* ⊆ *convex hull set ys*
by (*rule hull-minimal*) (*use assms* **in** ‹*auto simp*: *convex-convex-hull*›)
show *convex hull set ys* ⊆ *convex hull set xs*
by (*rule hull-minimal*) (*use assms* **in** ‹*auto simp*: *convex-convex-hull*›)
qed

theorem *andrew-hull-convex-hull-eqI*:
fixes *ps* :: *real point list*
assumes *set ps* ⊆ *convex hull set* (*andrew-hull ps*)
shows *convex hull set* (*andrew-hull ps*) = *convex hull set ps*
proof (*rule convex-hull-eq-from-mutual-inclusion*)
show *set* (*andrew-hull ps*) ⊆ *convex hull set ps*
proof
fix *p*
assume *p* ∈ *set* (*andrew-hull ps*)
then have *p* ∈ *set ps*
using *andrew-hull-subset*[*of ps*] **by** *blast*
then show *p* ∈ *convex hull set ps*
by (*rule hull-inc*)
qed
show *set ps* ⊆ *convex hull set* (*andrew-hull ps*)
by (*rule assms*)
qed

theorem *andrew-hull-convex-hull-eq-iff*:
fixes *ps* :: *real point list*
shows *convex hull set* (*andrew-hull ps*) = *convex hull set ps* ⟷
set ps ⊆ *convex hull set* (*andrew-hull ps*)
proof
assume *eq*: *convex hull set* (*andrew-hull ps*) = *convex hull set ps*
show *set ps* ⊆ *convex hull set* (*andrew-hull ps*)
using *eq* **by** (*auto intro*: *hull-inc*)
next
assume *set ps* ⊆ *convex hull set* (*andrew-hull ps*)

```
  then show convex hull set (andrew-hull ps) = convex hull set ps
    by (rule andrew-hull-convex-hull-eqI)
qed

theorem andrew-hull-correctI:
  fixes ps :: real point list
  assumes set ps ⊆ convex hull set (andrew-hull ps)
  shows convex-hull-correct ps (andrew-hull ps)
  using assms andrew-hull-subset[of ps] andrew-hull-convex-hull-eqI[of ps]
  by (simp add: convex-hull-correct-def)

theorem andrew-hull-correct-iff:
  fixes ps :: real point list
  shows convex-hull-correct ps (andrew-hull ps) ⟷
    set ps ⊆ convex hull set (andrew-hull ps)
proof
  assume convex-hull-correct ps (andrew-hull ps)
  then have convex hull set (andrew-hull ps) = convex hull set ps
    by (simp add: convex-hull-correct-def)
  then show set ps ⊆ convex hull set (andrew-hull ps)
    by (auto intro: hull-inc)
next
  assume set ps ⊆ convex hull set (andrew-hull ps)
  then show convex-hull-correct ps (andrew-hull ps)
    by (rule andrew-hull-correctI)
qed

lemma distinct-scan-push:
  assumes distinct st and p ∉ set st
  shows distinct (scan-push st p)
  using assms by (induction st p rule: scan-push.induct) auto

lemma distinct-scan-stack:
  assumes distinct ps
  shows distinct (scan-stack ps)
  using assms unfolding scan-stack-def
proof (induction ps arbitrary: rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc p ps)
  have dps: distinct ps and p-notin: p ∉ set ps
    using snoc.prems by auto
  have distinct (foldl scan-push [] ps)
    using snoc.IH dps by blast
  moreover have p ∉ set (foldl scan-push [] ps)
    using p-notin set-scan-stack-subset[of ps] by (auto simp: scan-stack-def)
  ultimately show ?case
    by (simp add: distinct-scan-push)
```

qed

lemma *distinct-scan-chain*:
assumes *distinct ps*
shows *distinct* (*scan-chain ps*)
using *assms distinct-scan-stack* **by** (*simp add*: *scan-chain-def*)

lemma *distinct-lower-hull*:
distinct (*lower-hull ps*)
by (*simp add*: *lower-hull-def distinct-scan-chain*)

lemma *distinct-upper-hull*:
distinct (*upper-hull ps*)
by (*simp add*: *upper-hull-def distinct-scan-chain*)

lemma *stack-turns-scan-push*:
assumes *stack-turns st*
shows *stack-turns* (*scan-push st p*)
using *assms*
proof (*induction st p rule*: *scan-push.induct*)
case (*3 q r st p*)
then show *?case*
by (*cases st*) (*auto simp*: *left-turn-def*)
qed *auto*

lemma *stack-turns-scan-stack*:
stack-turns (*scan-stack ps*)
unfolding *scan-stack-def*
proof (*induction ps arbitrary*: *rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc p ps*)
then show *?case*
by (*simp add*: *stack-turns-scan-push*)
qed

theorem *chain-turns-scan-chain*:
chain-turns (*scan-chain ps*)
using *stack-turns-scan-stack*[*of ps*] **by** (*simp add*: *chain-turns-def scan-chain-def*)

theorem *lower-hull-turns*:
chain-turns (*lower-hull ps*)
by (*simp add*: *lower-hull-def chain-turns-scan-chain*)

theorem *upper-hull-turns*:
chain-turns (*upper-hull ps*)
by (*simp add*: *upper-hull-def chain-turns-scan-chain*)

lemma *stack-turns-tl*:
assumes *stack-turns xs*
shows *stack-turns* (*tl xs*)
using *assms*
by (*cases xs*; *cases tl xs*; *cases tl* (*tl xs*)) *auto*

lemma *stack-turns-butlast*:
assumes *stack-turns xs*
shows *stack-turns* (*butlast xs*)
using *assms*
by (*induction xs rule*: *stack-turns.induct*) (*auto split*: *list.splits*)

lemma *stack-turns-append-last3*:
assumes *stack-turns* (*xs* @ [*z*, *y*, *x*])
shows *left-turn x y z*
using *assms*
proof (*induction xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a xs*)
have *stack-turns* (*tl* ((*a* # *xs*) @ [*z*, *y*, *x*]))
by (*rule stack-turns-tl*[*OF Cons.prems*])
then have *stack-turns* (*xs* @ [*z*, *y*, *x*])
by *simp*
then show *?case*
by (*rule Cons.IH*)
qed

lemma *chain-turns-tl*:
assumes *chain-turns xs*
shows *chain-turns* (*tl xs*)
proof (*cases xs*)
case *Nil*
then show *?thesis*
by (*simp add*: *chain-turns-def*)
next
case (*Cons x xs'*)
have *stack-turns* (*butlast* (*rev xs'* @ [*x*]))
using *assms Cons* **by** (*intro stack-turns-butlast*) (*simp add*: *chain-turns-def*)
then show *?thesis*
using *Cons* **by** (*simp add*: *chain-turns-def*)
qed

lemma *chain-turns-first*:
assumes *chain-turns* (*x* # *y* # *z* # *xs*)
shows *left-turn x y z*
using *assms stack-turns-append-last3*[*of rev xs z y x*]

```
  by (simp add: chain-turns-def)

lemma sorted-chain-cross-first-two:
  fixes x y z :: real point
  assumes sorted: sorted-wrt (<) (x # y # zs)
    and turns: chain-turns (x # y # zs)
    and z: z ∈ set zs
  shows 0 < cross x y z
  using sorted turns z
proof (induction zs arbitrary: x y z)
  case Nil
  then show ?case
    by simp
next
  case (Cons w ws)
  show ?case
  proof (cases z = w)
    case True
    have left-turn x y w
      using chain-turns-first[OF Cons.prems(2)] .
    then show ?thesis
      using True by (simp add: left-turn-def)
  next
    case False
    then have z-ws: z ∈ set ws
      using Cons.prems(3) by simp
    have sorted-tail: sorted-wrt (<) (y # w # ws)
      using Cons.prems(1) by simp
    have turns-tail: chain-turns (y # w # ws)
      using chain-turns-tl[OF Cons.prems(2)] by simp
    have y-w-z: 0 < cross y w z
      using Cons.IH[OF sorted-tail turns-tail z-ws] .
    have x-y-w: 0 < cross x y w
      using chain-turns-first[OF Cons.prems(2)]
      by (simp add: left-turn-def)
    have x < y and y < w and w < z
      using Cons.prems(1) z-ws by auto
    then show ?thesis
      using cross-pos-trans-left[OF - - - x-y-w y-w-z] by blast
  qed
qed

lemma sorted-chain-cross-nth-increasing:
  fixes xs :: real point list
  assumes sorted: sorted-wrt (<) xs
    and turns: chain-turns xs
    and ij: i < j
    and jk: j < k
    and k-len: k < length xs
```

```
  shows 0 < cross (xs ! i) (xs ! j) (xs ! k)
  using sorted turns ij jk k-len
proof (induction xs arbitrary: i j k)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case (Suc i')
    then obtain j' k' where j: j = Suc j' and k: k = Suc k'
      using Cons.prems(3,4) by (cases j; cases k) auto
    have 0 < cross (xs ! i') (xs ! j') (xs ! k')
    proof (rule Cons.IH)
      show sorted-wrt (<) xs
        using Cons.prems(1) by simp
      show chain-turns xs
        using chain-turns-tl[OF Cons.prems(2)] by simp
      show i' < j'
        using Cons.prems(3) Suc j by simp
      show j' < k'
        using Cons.prems(4) j k by simp
      show k' < length xs
        using Cons.prems(5) k by simp
    qed
    then show ?thesis
      using Suc j k by simp
  next
    case 0
    note i-zero = 0
    then have j-pos: 0 < j
      using Cons.prems(3) by simp
    obtain y ys where xs: xs = y # ys
      using j-pos Cons.prems(4,5) by (cases xs) auto
    obtain j' where j: j = Suc j'
      using j-pos by (cases j) auto
    obtain k' where k: k = Suc k'
      using Cons.prems(4) j by (cases k) auto
    have sorted-tail: sorted-wrt (<) xs
      using Cons.prems(1) by simp
    have turns-tail: chain-turns xs
      using chain-turns-tl[OF Cons.prems(2)] by simp
    show ?thesis
    proof (cases j')
      case 0
      then have k'-pos: 0 < k'
        using Cons.prems(4) j k by simp
      then obtain kk where kk: k' = Suc kk
```

```
    by (cases k') auto
  have kk-len: kk < length ys
    using Cons.prems(5) k kk xs by simp
  have 0 < cross x y (ys ! kk)
   using sorted-chain-cross-first-two[of x y ys ys ! kk] Cons.prems(1,2) xs kk-len
    by simp
  then show ?thesis
    using i-zero 0 j k kk xs by simp
next
  case (Suc jj)
  have j'-lt-k': j' < k'
    using Cons.prems(4) j k by simp
  have k'-len: k' < length xs
    using Cons.prems(5) k by simp
  have y-b-c: 0 < cross (xs ! 0) (xs ! j') (xs ! k')
  proof (rule Cons.IH)
    show sorted-wrt (<) xs
      using sorted-tail .
    show chain-turns xs
      using turns-tail .
    show 0 < j'
      using Suc by simp
    show j' < k'
      using j'-lt-k' .
    show k' < length xs
      using k'-len .
  qed
  have y-b-c': 0 < cross y (xs ! j') (xs ! k')
    using y-b-c xs by simp
  have jj-len: jj < length ys
    using Suc k'-len j'-lt-k' xs by simp
  have x-y-b: 0 < cross x y (ys ! jj)
   using sorted-chain-cross-first-two[of x y ys ys ! jj] Cons.prems(1,2) xs jj-len
    by simp
  have x-y-b': 0 < cross x y (xs ! j')
    using x-y-b Suc xs by simp
  have x-lt-y: x < y
    using Cons.prems(1) xs by simp
  have j'-len: j' < length xs
    using j'-lt-k' k'-len by simp
  have y-lt-b: y < xs ! j'
    using sorted-wrt-nth-less[OF sorted-tail, of 0 j'] Suc j'-len xs by simp
  have b-lt-c: xs ! j' < xs ! k'
    using sorted-wrt-nth-less[OF sorted-tail, of j' k'] j'-lt-k' k'-len .
  have 0 < cross x (xs ! j') (xs ! k')
    by (rule cross-pos-trans-right[OF x-lt-y y-lt-b b-lt-c x-y-b' y-b-c'])
  then show ?thesis
  using i-zero j k by simp
qed
```

```
  qed
qed

lemma uminus-less-point-iff [simp]:
  fixes p q :: real point
  shows − p < − q ⟷ q < p
  by (cases p; cases q) (auto simp: less-prod-def')

lemma cross-uminus [simp]:
  fixes p q r :: real point
  shows cross (− p) (− q) (− r) = cross p q r
  by (cases p; cases q; cases r) (simp add: cross-def algebra-simps)

lemma left-turn-uminus [simp]:
  fixes p q r :: real point
  shows left-turn (− p) (− q) (− r) ⟷ left-turn p q r
  by (simp add: left-turn-def)

lemma stack-turns-map-uminus [simp]:
  fixes xs :: real point list
  shows stack-turns (map (λp. − p) xs) ⟷ stack-turns xs
  by (induction xs rule: stack-turns.induct) auto

lemma chain-turns-map-uminus [simp]:
  fixes xs :: real point list
  shows chain-turns (map (λp. − p) xs) ⟷ chain-turns xs
  by (simp add: chain-turns-def rev-map)

lemma sorted-wrt-less-map-uminus:
  fixes xs :: real point list
  assumes sorted-wrt (>) xs
  shows sorted-wrt (<) (map (λp. − p) xs)
  using assms by (induction xs) auto

lemma sorted-chain-cross-nth-decreasing:
  fixes xs :: real point list
  assumes sorted: sorted-wrt (>) xs
    and turns: chain-turns xs
    and ij: i < j
    and jk: j < k
    and k-len: k < length xs
  shows 0 < cross (xs ! i) (xs ! j) (xs ! k)
proof −
  let ?xs = map (λp. − p) xs
  have 0 < cross (?xs ! i) (?xs ! j) (?xs ! k)
  proof (rule sorted-chain-cross-nth-increasing)
    show sorted-wrt (<) ?xs
      by (rule sorted-wrt-less-map-uminus[OF sorted])
    show chain-turns ?xs
```

```
    using turns by simp
  show i < j
    using ij .
  show j < k
    using jk .
  show k < length ?xs
    using k-len by simp
 qed
 then show ?thesis
  using ij jk k-len by simp
qed

lemma sorted-chain-edge-cross-nonneg-increasing:
 fixes xs :: real point list
 assumes sorted: sorted-wrt (<) xs
  and turns: chain-turns xs
  and edge: Suc i < length xs
  and q: q ∈ set xs
 shows 0 ≤ cross (xs ! i) (xs ! Suc i) q
proof −
 obtain k where k-len: k < length xs and q-eq: q = xs ! k
  using q by (auto simp: in-set-conv-nth)
 show ?thesis
 proof (cases k = i ∨ k = Suc i)
  case True
  then show ?thesis
   using q-eq by auto
 next
  case False
  note not-endpoint = False
  show ?thesis
  proof (cases k < i)
   case True
   have 0 < cross (xs ! k) (xs ! i) (xs ! Suc i)
    by (rule sorted-chain-cross-nth-increasing[OF sorted turns True - edge]) simp
   then show ?thesis
    using q-eq cross-cycle[of xs ! k xs ! i xs ! Suc i] by simp
  next
   case False
   then have Suc i < k
    using not-endpoint by linarith
   have 0 < cross (xs ! i) (xs ! Suc i) (xs ! k)
     by (rule sorted-chain-cross-nth-increasing[OF sorted turns - ‹Suc i < k›
k-len]) simp
   then show ?thesis
    using q-eq by simp
  qed
 qed
qed
```

```
lemma sorted-chain-edge-cross-nonneg-decreasing:
  fixes xs :: real point list
  assumes sorted: sorted-wrt (>) xs
    and turns: chain-turns xs
    and edge: Suc i < length xs
    and q: q ∈ set xs
  shows 0 ≤ cross (xs ! i) (xs ! Suc i) q
proof −
  obtain k where k-len: k < length xs and q-eq: q = xs ! k
    using q by (auto simp: in-set-conv-nth)
  show ?thesis
  proof (cases k = i ∨ k = Suc i)
    case True
    then show ?thesis
      using q-eq by auto
  next
    case False
    note not-endpoint = False
    show ?thesis
    proof (cases k < i)
      case True
      have 0 < cross (xs ! k) (xs ! i) (xs ! Suc i)
       by (rule sorted-chain-cross-nth-decreasing[OF sorted turns True - edge]) simp
      then show ?thesis
        using q-eq cross-cycle[of xs ! k xs ! i xs ! Suc i] by simp
    next
      case False
      then have Suc i < k
        using not-endpoint by linarith
      have 0 < cross (xs ! i) (xs ! Suc i) (xs ! k)
         by (rule sorted-chain-cross-nth-decreasing[OF sorted turns - ‹Suc i < k›
k-len]) simp
      then show ?thesis
        using q-eq by simp
    qed
  qed
qed

lemma scan-push-nonempty [simp]:
  scan-push st p ≠ []
  by (induction st p rule: scan-push.induct) auto

lemma scan-stack-nonempty:
  ps ≠ [] ⟹ scan-stack ps ≠ []
  by (cases ps rule: rev-cases) (simp-all add: scan-stack-def)

lemma scan-chain-nonempty:
  ps ≠ [] ⟹ scan-chain ps ≠ []
```

by (*simp add*: *scan-chain-def scan-stack-nonempty*)

lemma *lower-hull-nonempty*:
ps ≠ [] ⟹ *lower-hull ps* ≠ []
by (*simp add*: *lower-hull-def scan-chain-nonempty*)

lemma *upper-hull-nonempty*:
ps ≠ [] ⟹ *upper-hull ps* ≠ []
by (*simp add*: *upper-hull-def scan-chain-nonempty*)

1.2 Endpoints of the Scans

lemma *hd-scan-push* [*simp*]:
hd (*scan-push st p*) = *p*
by (*induction st p rule*: *scan-push.induct*) *simp-all*

lemma *last-scan-push*:
assumes *st* ≠ []
shows *last* (*scan-push st p*) = *last st*
using *assms* **by** (*induction st p rule*: *scan-push.induct*) *simp-all*

lemma *hd-scan-stack*:
assumes *ps* ≠ []
shows *hd* (*scan-stack ps*) = *last ps*
using *assms*
proof (*induction ps rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc p ps*)
show *?case*
by (*simp add*: *scan-stack-def*)
qed

lemma *last-scan-stack*:
assumes *ps* ≠ []
shows *last* (*scan-stack ps*) = *hd ps*
using *assms*
proof (*induction ps rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc p ps*)
show *?case*
proof (*cases ps*)
case *Nil*
then show *?thesis*
by (*simp add*: *scan-stack-def*)
next

```
    case (Cons q qs)
    let ?st = foldl scan-push [] ps
    have ?st ≠ []
      using Cons scan-stack-nonempty[of ps] by (simp add: scan-stack-def)
    moreover have last ?st = hd ps
    proof −
      have last (scan-stack ps) = hd ps
        using snoc.IH Cons by simp
      then show ?thesis
        by (simp add: scan-stack-def)
    qed
    ultimately show ?thesis
      using Cons by (simp add: scan-stack-def last-scan-push)
  qed
qed

lemma hd-scan-chain:
  assumes ps ≠ []
  shows hd (scan-chain ps) = hd ps
proof −
  have scan-stack ps ≠ []
    using assms scan-stack-nonempty by blast
  then show ?thesis
    using assms last-scan-stack[of ps] by (simp add: scan-chain-def hd-rev)
qed

lemma last-scan-chain:
  assumes ps ≠ []
  shows last (scan-chain ps) = last ps
proof −
  have scan-stack ps ≠ []
    using assms scan-stack-nonempty by blast
  then show ?thesis
    using assms hd-scan-stack[of ps] by (simp add: scan-chain-def last-rev)
qed

lemma hd-lower-hull:
  assumes ps ≠ []
  shows hd (lower-hull ps) = hd (sorted-unique ps)
  using assms by (simp add: lower-hull-def hd-scan-chain)

lemma last-lower-hull:
  assumes ps ≠ []
  shows last (lower-hull ps) = last (sorted-unique ps)
  using assms by (simp add: lower-hull-def last-scan-chain)

lemma hd-upper-hull:
  assumes ps ≠ []
  shows hd (upper-hull ps) = last (sorted-unique ps)
```

```
proof −
  have su-nonempty: sorted-unique ps ≠ []
    using assms by simp
  then show ?thesis
    using assms by (simp add: upper-hull-def hd-scan-chain hd-rev)
qed

lemma last-upper-hull:
  assumes ps ≠ []
  shows last (upper-hull ps) = hd (sorted-unique ps)
proof −
  have su-nonempty: sorted-unique ps ≠ []
    using assms by simp
  then show ?thesis
    using assms by (simp add: upper-hull-def last-scan-chain last-rev)
qed

lemma length-scan-chain-ge2:
  assumes xs ≠ [] and hd xs ≠ last xs
  shows 2 ≤ length (scan-chain xs)
proof −
  have ch-nonempty: scan-chain xs ≠ []
    using assms(1) scan-chain-nonempty by blast
  have endpoints: hd (scan-chain xs) ≠ last (scan-chain xs)
    using assms hd-scan-chain[of xs] last-scan-chain[of xs] by simp
  obtain y ys where ch: scan-chain xs = y # ys
    using ch-nonempty by (cases scan-chain xs) auto
  then show ?thesis
  proof (cases ys)
    case Nil
    then show ?thesis
      using ch endpoints by simp
  next
    case (Cons z zs)
    then show ?thesis
      using ch by simp
  qed
qed

lemma hd-ne-last-sorted-unique-if-card-ge2:
  assumes 2 ≤ card (set ps)
  shows hd (sorted-unique ps) ≠ last (sorted-unique ps)
proof −
  have len: length (sorted-unique ps) = card (set ps)
    by (simp add: sorted-unique-def length-sorted-list-of-set)
  then have len-ge2: 2 ≤ length (sorted-unique ps)
    using assms by simp
  obtain x xs where su-Cons: sorted-unique ps = x # xs
    using len-ge2 by (cases sorted-unique ps) auto
```

```
  then obtain y ys where xs-Cons: xs = y # ys
    using len-ge2 by (cases xs) auto
  have last (sorted-unique ps) ∈ set xs
    by (simp add: su-Cons xs-Cons)
  then show ?thesis
    using sorted-unique-distinct[of ps] by (auto simp: su-Cons)
qed

lemma length-lower-hull-ge2:
  assumes 2 ≤ card (set ps)
  shows 2 ≤ length (lower-hull ps)
proof −
  have ps-nonempty: ps ≠ []
    using assms by auto
  have su-ne: hd (sorted-unique ps) ≠ last (sorted-unique ps)
    using hd-ne-last-sorted-unique-if-card-ge2[OF assms] .
  show ?thesis
    using length-scan-chain-ge2[of sorted-unique ps] ps-nonempty su-ne
    by (simp add: lower-hull-def)
qed

lemma length-upper-hull-ge2:
  assumes 2 ≤ card (set ps)
  shows 2 ≤ length (upper-hull ps)
proof −
  have ps-nonempty: ps ≠ []
    using assms by auto
  have su-nonempty: sorted-unique ps ≠ []
    using ps-nonempty by simp
  have rev-ne: hd (rev (sorted-unique ps)) ≠ last (rev (sorted-unique ps))
    using hd-ne-last-sorted-unique-if-card-ge2[OF assms] su-nonempty
    by (simp add: hd-rev last-rev)
  show ?thesis
    using length-scan-chain-ge2[of rev (sorted-unique ps)] su-nonempty rev-ne
    by (simp add: upper-hull-def)
qed

lemma set-butlast-last:
  assumes xs ≠ []
  shows set xs = set (butlast xs) ∪ {last xs}
proof −
  have xs = butlast xs @ [last xs]
    using assms by simp
  then have set xs = set (butlast xs @ [last xs])
    by auto
  also have ... = set (butlast xs) ∪ {last xs}
    by simp
  finally show ?thesis .
qed
```

```
lemma hd-mem-set-butlast:
  assumes 2 ≤ length xs
  shows hd xs ∈ set (butlast xs)
proof (cases xs)
  case Nil
  then show ?thesis
    using assms by simp
next
  case (Cons x xs')
  then show ?thesis
  proof (cases xs')
    case Nil
    then show ?thesis
      using assms Cons by simp
  next
    case (Cons y ys)
    then show ?thesis
      using ‹xs = x # xs'› by simp
  qed
qed

theorem set-andrew-hull:
  set (andrew-hull ps) = set (lower-hull ps) ∪ set (upper-hull ps)
proof (cases sorted-unique ps)
  case Nil
  then have ps = []
    by simp
  then show ?thesis
    by (simp add: andrew-hull-def lower-hull-def upper-hull-def
        scan-chain-def scan-stack-def sorted-unique-def)
next
  case (Cons p qs)
  then show ?thesis
  proof (cases qs)
    case Nil
    then show ?thesis
      using Cons
      by (simp add: andrew-hull-def lower-hull-def upper-hull-def
          scan-chain-def scan-stack-def sorted-unique-def)
  next
    case (Cons q rs)
    let ?L = lower-hull ps
    let ?U = upper-hull ps
    have ps-nonempty: ps ≠ []
      using ‹sorted-unique ps = p # qs› sorted-unique-Nil-iff[of ps] by force
    have len-su: length (sorted-unique ps) = card (set ps)
      by (simp add: sorted-unique-def length-sorted-list-of-set)
    have 2 ≤ length (sorted-unique ps)
```

```
    using ‹sorted-unique ps = p # qs› Cons by simp
  then have card-ge2: 2 ≤ card (set ps)
    using len-su by simp
  have andrew: andrew-hull ps = butlast ?L @ butlast ?U
    using ‹sorted-unique ps = p # qs› Cons by (simp add: andrew-hull-def)
  have L-nonempty: ?L ≠ []
    using ps-nonempty lower-hull-nonempty by blast
  have U-nonempty: ?U ≠ []
    using ps-nonempty upper-hull-nonempty by blast
  have last-L: last ?L = hd ?U
    using ps-nonempty last-lower-hull[of ps] hd-upper-hull[of ps] by simp
  have last-U: last ?U = hd ?L
    using ps-nonempty last-upper-hull[of ps] hd-lower-hull[of ps] by simp
  have last-L-in-U: last ?L ∈ set (butlast ?U)
    using last-L hd-mem-set-butlast[OF length-upper-hull-ge2[OF card-ge2]] by
simp
  have last-U-in-L: last ?U ∈ set (butlast ?L)
    using last-U hd-mem-set-butlast[OF length-lower-hull-ge2[OF card-ge2]] by
simp
  show ?thesis
   using andrew set-butlast-last[OF L-nonempty] set-butlast-last[OF U-nonempty]
     last-L-in-U last-U-in-L
    by auto
 qed
qed
```

1.3 Support-Function Invariants

```
definition support-value :: real point ⇒ real point ⇒ real
where
  support-value v p = fst v * fst p + snd v * snd p

lemma support-value-eq-inner:
  support-value v p = inner v p
  by (cases v; cases p; simp add: support-value-def)

lemma support-value-edge-normal:
  fixes a b x :: real point
  shows support-value (snd b − snd a, fst a − fst b) x =
    support-value (snd b − snd a, fst a − fst b) a − cross a b x
 by (cases a; cases b; cases x) (simp add: support-value-def cross-def algebra-simps)

lemma support-middle-le-max-increasing:
  fixes r q p v :: real point
  assumes rq: r < q and qp: q < p
    and turn: cross r q p ≤ 0
    and neg: snd v < 0
  shows support-value v q ≤ max (support-value v r) (support-value v p)
proof (cases fst r = fst p)
```

```
  case True
  have frq: fst r = fst q and fqp: fst q = fst p
    using rq qp True by (auto simp: less-prod-def')
  have yrq: snd r ≤ snd q
    using rq frq by (auto simp: less-prod-def')
  have snd v * snd q ≤ snd v * snd r
    using yrq neg by (intro mult-left-mono-neg) auto
  then have support-value v q ≤ support-value v r
    using frq fqp by (simp add: support-value-def)
  then show ?thesis
    by simp
next
  case False
  define A where A = fst q − fst r
  define B where B = fst p − fst q
  define D where D = fst p − fst r
  have xrq: fst r ≤ fst q
    using rq by (auto simp: less-prod-def')
  have xqp: fst q ≤ fst p
    using qp by (auto simp: less-prod-def')
  have A-nonneg: 0 ≤ A
    using xrq by (simp add: A-def)
  have B-nonneg: 0 ≤ B
    using xqp by (simp add: B-def)
  have D-pos: 0 < D
    using xrq xqp False unfolding D-def by linarith
  have D-eq: D = A + B
    by (simp add: A-def B-def D-def)
  have x-combo: D * fst q = B * fst r + A * fst p
    by (simp add: A-def B-def D-def algebra-simps)
  have y-combo: B * snd r + A * snd p ≤ D * snd q
  proof −
    have A * (snd p − snd r) ≤ (snd q − snd r) * D
      using turn by (simp add: cross-def A-def D-def)
    then show ?thesis
      using D-eq by (simp add: algebra-simps)
  qed
  have y-scaled:
    snd v * (D * snd q) ≤ snd v * (B * snd r + A * snd p)
    using y-combo neg by (intro mult-left-mono-neg) auto
  have x-part:
    D * (fst v * fst q) = B * (fst v * fst r) + A * (fst v * fst p)
  proof −
    have D * (fst v * fst q) = fst v * (D * fst q)
      by (simp add: algebra-simps)
    also have ... = fst v * (B * fst r + A * fst p)
      using x-combo by simp
    also have ... = B * (fst v * fst r) + A * (fst v * fst p)
      by (simp add: algebra-simps)
```

```
    finally show ?thesis .
  qed
  have y-part:
    D * (snd v * snd q) ≤ B * (snd v * snd r) + A * (snd v * snd p)
  proof −
    have D * (snd v * snd q) = snd v * (D * snd q)
      by (simp add: algebra-simps)
    also have ... ≤ snd v * (B * snd r + A * snd p)
      using y-scaled .
    also have ... = B * (snd v * snd r) + A * (snd v * snd p)
      by (simp add: algebra-simps)
    finally show ?thesis .
  qed
  have scaled-le:
    D * support-value v q ≤
      B * support-value v r + A * support-value v p
  proof −
    have D * support-value v q =
        D * (fst v * fst q) + D * (snd v * snd q)
      by (simp add: support-value-def algebra-simps)
    also have ... ≤
        (B * (fst v * fst r) + A * (fst v * fst p)) +
        (B * (snd v * snd r) + A * (snd v * snd p))
    proof (rule add-mono)
      show D * (fst v * fst q) ≤
          B * (fst v * fst r) + A * (fst v * fst p)
        using x-part by simp
      show D * (snd v * snd q) ≤
          B * (snd v * snd r) + A * (snd v * snd p)
        using y-part .
    qed
    also have ... = B * support-value v r + A * support-value v p
      by (simp add: support-value-def algebra-simps)
    finally show ?thesis .
  qed
  have r-le: B * support-value v r ≤
      B * max (support-value v r) (support-value v p)
    using B-nonneg by (intro mult-left-mono) simp-all
  have p-le: A * support-value v p ≤
      A * max (support-value v r) (support-value v p)
    using A-nonneg by (intro mult-left-mono) simp-all
  have D * support-value v q ≤
      D * max (support-value v r) (support-value v p)
    using scaled-le r-le p-le D-eq by (simp add: algebra-simps)
  then show ?thesis
    using D-pos by simp
qed

lemma support-middle-le-max-decreasing:
```

```
  fixes r q p v :: real point
  assumes pq: p < q and qr: q < r
    and turn: cross r q p ≤ 0
    and pos: 0 < snd v
  shows support-value v q ≤ max (support-value v r) (support-value v p)
proof (cases fst p = fst r)
  case True
  have fpq: fst p = fst q and fqr: fst q = fst r
    using pq qr True by (auto simp: less-prod-def')
  have yqr: snd q ≤ snd r
    using qr fqr by (auto simp: less-prod-def')
  have snd v * snd q ≤ snd v * snd r
    using yqr pos by (intro mult-left-mono) auto
  then have support-value v q ≤ support-value v r
    using fpq fqr by (simp add: support-value-def)
  then show ?thesis
    by simp
next
  case False
  define A where A = fst q - fst p
  define B where B = fst r - fst q
  define D where D = fst r - fst p
  have xpq: fst p ≤ fst q
    using pq by (auto simp: less-prod-def')
  have xqr: fst q ≤ fst r
    using qr by (auto simp: less-prod-def')
  have A-nonneg: 0 ≤ A
    using xpq by (simp add: A-def)
  have B-nonneg: 0 ≤ B
    using xqr by (simp add: B-def)
  have D-pos: 0 < D
    using xpq xqr False unfolding D-def by linarith
  have D-eq: D = A + B
    by (simp add: A-def B-def D-def)
  have x-combo: D * fst q = B * fst p + A * fst r
    by (simp add: A-def B-def D-def algebra-simps)
  have y-combo: D * snd q ≤ B * snd p + A * snd r
  proof -
    have (snd q - snd p) * D ≤ A * (snd r - snd p)
      using turn by (simp add: cross-def A-def D-def algebra-simps)
    then show ?thesis
      using D-eq by (simp add: algebra-simps)
  qed
  have y-scaled:
    snd v * (D * snd q) ≤ snd v * (B * snd p + A * snd r)
    using y-combo pos by (intro mult-left-mono) auto
  have x-part:
    D * (fst v * fst q) = B * (fst v * fst p) + A * (fst v * fst r)
  proof -
```

```
  have D * (fst v * fst q) = fst v * (D * fst q)
    by (simp add: algebra-simps)
  also have ... = fst v * (B * fst p + A * fst r)
    using x-combo by simp
  also have ... = B * (fst v * fst p) + A * (fst v * fst r)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed
have y-part:
  D * (snd v * snd q) ≤ B * (snd v * snd p) + A * (snd v * snd r)
proof −
  have D * (snd v * snd q) = snd v * (D * snd q)
    by (simp add: algebra-simps)
  also have ... ≤ snd v * (B * snd p + A * snd r)
    using y-scaled .
  also have ... = B * (snd v * snd p) + A * (snd v * snd r)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed
have scaled-le:
  D * support-value v q ≤
    B * support-value v p + A * support-value v r
proof −
  have D * support-value v q =
      D * (fst v * fst q) + D * (snd v * snd q)
    by (simp add: support-value-def algebra-simps)
  also have ... ≤
      (B * (fst v * fst p) + A * (fst v * fst r)) +
      (B * (snd v * snd p) + A * (snd v * snd r))
  proof (rule add-mono)
    show D * (fst v * fst q) ≤
        B * (fst v * fst p) + A * (fst v * fst r)
      using x-part by simp
    show D * (snd v * snd q) ≤
        B * (snd v * snd p) + A * (snd v * snd r)
      using y-part .
  qed
  also have ... = B * support-value v p + A * support-value v r
    by (simp add: support-value-def algebra-simps)
  finally show ?thesis .
qed
have p-le: B * support-value v p ≤
    B * max (support-value v r) (support-value v p)
  using B-nonneg by (intro mult-left-mono) simp-all
have r-le: A * support-value v r ≤
    A * max (support-value v r) (support-value v p)
  using A-nonneg by (intro mult-left-mono) simp-all
have D * support-value v q ≤
    D * max (support-value v r) (support-value v p)
```

```
    using scaled-le p-le r-le D-eq by (simp add: algebra-simps)
  then show ?thesis
    using D-pos by simp
qed

lemma scan-push-strict-sorted-increasing:
  fixes st :: real point list
  assumes sorted: sorted-wrt (<) (rev st)
    and below: ⋀q. q ∈ set st ⟹ q < p
  shows sorted-wrt (<) (rev (scan-push st p))
  using sorted below
  by (induction st p rule: scan-push.induct) (auto simp: sorted-wrt-append)

lemma scan-stack-strict-sorted-increasing:
  fixes xs :: real point list
  assumes sorted-wrt (<) xs
  shows sorted-wrt (<) (rev (scan-stack xs))
  using assms unfolding scan-stack-def
proof (induction xs rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc p xs)
  have sorted-xs: sorted-wrt (<) xs
    using snoc.prems by (simp add: sorted-wrt-append)
  have below: ⋀q. q ∈ set (foldl scan-push [] xs) ⟹ q < p
  proof −
    fix q
    assume q ∈ set (foldl scan-push [] xs)
    then have q ∈ set xs
      using set-scan-stack-subset[of xs] by (auto simp: scan-stack-def)
    then show q < p
      using snoc.prems by (simp add: sorted-wrt-append)
  qed
  show ?case
    using scan-push-strict-sorted-increasing[OF snoc.IH[OF sorted-xs] below] by
simp
qed

lemma scan-push-support-dominates-increasing-step:
  fixes q r p x :: real point and st :: real point list
  assumes tail-dom:
    cross r q p ≤ 0 ⟹
      (⋀x. x ∈ insert p (set (r # st)) ⟹
        ∃y∈set (scan-push (r # st) p). support-value v x ≤ support-value v y)
    and sorted: sorted-wrt (<) (rev (q # r # st))
    and below: ⋀z. z ∈ set (q # r # st) ⟹ z < p
    and neg: snd v < 0
    and x-in: x ∈ insert p (set (q # r # st))
```

```
  shows ∃ y∈set (scan-push (q # r # st) p). support-value v x ≤ support-value v y
proof (cases cross r q p ≤ 0)
  case False
  have x ∈ set (scan-push (q # r # st) p)
    using False x-in by simp
  then show ?thesis
    by (intro bexI[where x = x]) simp-all
next
  case True
  show ?thesis
  proof (cases x = q)
    case False
    then have x-tail: x ∈ insert p (set (r # st))
      using x-in by auto
    obtain y where y-set: y ∈ set (scan-push (r # st) p)
      and y-ge: support-value v x ≤ support-value v y
      using tail-dom[OF True x-tail] by auto
    have scan-push (q # r # st) p = scan-push (r # st) p
      using True by simp
    then show ?thesis
      using y-set y-ge by auto
  next
    case True-x: True
    have rq: r < q
      using sorted by (simp add: sorted-wrt-append)
    have qp: q < p
      using below by simp
    have q-le: support-value v q ≤ max (support-value v r) (support-value v p)
      using support-middle-le-max-increasing[OF rq qp True neg] .
    obtain yr where yr-set: yr ∈ set (scan-push (r # st) p)
      and yr-ge: support-value v r ≤ support-value v yr
      using tail-dom[OF True, of r] by auto
    obtain yp where yp-set: yp ∈ set (scan-push (r # st) p)
      and yp-ge: support-value v p ≤ support-value v yp
      using tail-dom[OF True, of p] by auto
    have support-value v q ≤ support-value v r ∨
       support-value v q ≤ support-value v p
      using q-le by (simp add: max-def split: if-splits)
    then show ?thesis
    proof
      assume support-value v q ≤ support-value v r
      then have support-value v q ≤ support-value v yr
        using yr-ge by (rule order-trans)
      then have support-value v x ≤ support-value v yr
        using True-x by simp
      then show ?thesis
        using True yr-set by (intro bexI[where x = yr]) simp-all
    next
      assume support-value v q ≤ support-value v p
```

```
    then have support-value v q ≤ support-value v yp
      using yp-ge by (rule order-trans)
    then have support-value v x ≤ support-value v yp
      using True-x by simp
    then show ?thesis
      using True yp-set by (intro bexI[where x = yp]) simp-all
  qed
 qed
qed

lemma scan-push-support-dominates-increasing:
 fixes st :: real point list
 shows sorted-wrt (<) (rev st) ⟹
  (⋀q. q ∈ set st ⟹ q < p) ⟹
  snd v < 0 ⟹
  x ∈ insert p (set st) ⟹
  ∃y∈set (scan-push st p). support-value v x ≤ support-value v y
proof (induction st p arbitrary: x rule: scan-push.induct)
 case (1 p)
 then show ?case by auto
next
 case (2 q p)
 then show ?case by auto
next
 case (3 q r st p)
 have sorted-tail: sorted-wrt (<) (rev (r # st))
   using 3.prems(1) by (simp add: sorted-wrt-append)
 have below-tail: ⋀w. w ∈ set (r # st) ⟹ w < p
   using 3.prems(2) by simp
 have neg: snd v < 0
   using 3.prems(3) .
 have tail-dom:
   cross r q p ≤ 0 ⟹
    (⋀z. z ∈ insert p (set (r # st)) ⟹
     ∃y∈set (scan-push (r # st) p). support-value v z ≤ support-value v y)
   using 3.IH[OF - sorted-tail below-tail neg] by blast
 have sorted: sorted-wrt (<) (rev (q # r # st))
   using 3.prems(1) .
 have below: ⋀z. z ∈ set (q # r # st) ⟹ z < p
   using 3.prems(2) .
 have x-in: x ∈ insert p (set (q # r # st))
   using 3.prems(4) .
 show ?case
   by (rule scan-push-support-dominates-increasing-step[
       OF tail-dom sorted below neg x-in])
qed

lemma scan-stack-support-dominates-increasing:
 fixes xs :: real point list
```

```
  assumes sorted: sorted-wrt (<) xs
    and neg: snd v < 0
    and x: x ∈ set xs
  shows ∃ y∈set (scan-stack xs). support-value v x ≤ support-value v y
  using sorted x
proof (induction xs arbitrary: x rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc p xs)
  have sorted-xs: sorted-wrt (<) xs
    using snoc.prems(1) by (simp add: sorted-wrt-append)
  have stack-sorted: sorted-wrt (<) (rev (scan-stack xs))
    using scan-stack-strict-sorted-increasing[OF sorted-xs] .
  have below: ⋀q. q ∈ set (scan-stack xs) ⟹ q < p
  proof −
    fix q
    assume q ∈ set (scan-stack xs)
    then have q ∈ set xs
      using set-scan-stack-subset[of xs] by blast
    then show q < p
      using snoc.prems(1) by (simp add: sorted-wrt-append)
  qed
  have push-dom:
    ⋀z. z ∈ insert p (set (scan-stack xs)) ⟹
      ∃ y∈set (scan-push (scan-stack xs) p). support-value v z ≤ support-value v y
     using scan-push-support-dominates-increasing[OF stack-sorted below neg] by
blast
  show ?case
  proof (cases x = p)
    case True
    then show ?thesis
      using push-dom[of p] by (simp add: scan-stack-def)
  next
    case False
    then have x-in-xs: x ∈ set xs
      using snoc.prems(2) by auto
    obtain z where z-set: z ∈ set (scan-stack xs)
      and z-ge: support-value v x ≤ support-value v z
      using snoc.IH[OF sorted-xs x-in-xs] by blast
    obtain y where y-set: y ∈ set (scan-push (scan-stack xs) p)
      and y-ge: support-value v z ≤ support-value v y
      using push-dom[of z] z-set by auto
    have xy-ge: support-value v x ≤ support-value v y
      by (rule order-trans[OF z-ge y-ge])
    show ?thesis
      using y-set xy-ge by (auto simp: scan-stack-def)
  qed
qed
```

lemma *scan-chain-support-dominates-increasing*:
fixes *xs* :: *real point list*
assumes *sorted-wrt* (<) *xs* **and** *snd v* < *0* **and** *x* ∈ *set xs*
shows ∃ *y*∈*set* (*scan-chain xs*). *support-value v x* ≤ *support-value v y*
using *scan-stack-support-dominates-increasing*[*OF assms*]
by (*simp add*: *scan-chain-def*)

lemma *sorted-wrt-less-sorted-unique* [*simp*]:
sorted-wrt (<) (*sorted-unique* (*xs* :: *'a*::*linorder list*))
by (*simp add*: *strict-sorted-iff*)

lemma *lower-hull-supports-negative*:
assumes *snd v* < *0* **and** *x* ∈ *set ps*
shows ∃ *y*∈*set* (*lower-hull ps*). *support-value v x* ≤ *support-value v y*
using *scan-chain-support-dominates-increasing*[*of sorted-unique ps v x*] *assms*
by (*simp add*: *lower-hull-def*)

lemma *scan-push-strict-sorted-decreasing*:
fixes *st* :: *real point list*
assumes *sorted*: *sorted-wrt* (>) (*rev st*)
and *above*: ⋀*q*. *q* ∈ *set st* ⟹ *p* < *q*
shows *sorted-wrt* (>) (*rev* (*scan-push st p*))
using *sorted above*
by (*induction st p rule*: *scan-push.induct*) (*auto simp*: *sorted-wrt-append*)

lemma *scan-stack-strict-sorted-decreasing*:
fixes *xs* :: *real point list*
assumes *sorted-wrt* (>) *xs*
shows *sorted-wrt* (>) (*rev* (*scan-stack xs*))
using *assms* **unfolding** *scan-stack-def*
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc p xs*)
have *sorted-xs*: *sorted-wrt* (>) *xs*
using *snoc.prems* **by** (*simp add*: *sorted-wrt-append*)
have *above*: ⋀*q*. *q* ∈ *set* (*foldl scan-push* [] *xs*) ⟹ *p* < *q*
proof −
fix *q*
assume *q* ∈ *set* (*foldl scan-push* [] *xs*)
then have *q* ∈ *set xs*
using *set-scan-stack-subset*[*of xs*] **by** (*auto simp*: *scan-stack-def*)
then show *p* < *q*
using *snoc.prems* **by** (*simp add*: *sorted-wrt-append*)
qed
show *?case*
using *scan-push-strict-sorted-decreasing*[*OF snoc.IH*[*OF sorted-xs*] *above*] **by**

simp
qed

lemma *scan-push-support-dominates-decreasing-step*:
fixes *q r p x* :: *real point* **and** *st* :: *real point list*
assumes *tail-dom*:
cross r q p $\leq$ *0* $\Longrightarrow$
($\bigwedge$*x*. *x* $\in$ *insert p* (*set* (*r* # *st*)) $\Longrightarrow$
$\exists$ *y*$\in$*set* (*scan-push* (*r* # *st*) *p*). *support-value v x* $\leq$ *support-value v y*)
and *sorted*: *sorted-wrt* (>) (*rev* (*q* # *r* # *st*))
and *above*: $\bigwedge$*z*. *z* $\in$ *set* (*q* # *r* # *st*) $\Longrightarrow$ *p* < *z*
and *pos*: *0* < *snd v*
and *x-in*: *x* $\in$ *insert p* (*set* (*q* # *r* # *st*))
shows $\exists$ *y*$\in$*set* (*scan-push* (*q* # *r* # *st*) *p*). *support-value v x* $\leq$ *support-value v y*
proof (*cases cross r q p* $\leq$ *0*)
case *False*
have *x* $\in$ *set* (*scan-push* (*q* # *r* # *st*) *p*)
using *False x-in* **by** *simp*
then show *?thesis*
by (*intro bexI*[**where** *x* = *x*]) *simp-all*
next
case *True*
show *?thesis*
proof (*cases x* = *q*)
case *False*
then have *x-tail*: *x* $\in$ *insert p* (*set* (*r* # *st*))
using *x-in* **by** *auto*
obtain *y* **where** *y-set*: *y* $\in$ *set* (*scan-push* (*r* # *st*) *p*)
and *y-ge*: *support-value v x* $\leq$ *support-value v y*
using *tail-dom*[*OF True x-tail*] **by** *auto*
have *scan-push* (*q* # *r* # *st*) *p* = *scan-push* (*r* # *st*) *p*
using *True* **by** *simp*
then show *?thesis*
using *y-set y-ge* **by** *auto*
next
case *True-x*: *True*
have *pq*: *p* < *q*
using *above* **by** *simp*
have *qr*: *q* < *r*
using *sorted* **by** (*simp add*: *sorted-wrt-append*)
have *q-le*: *support-value v q* $\leq$ *max* (*support-value v r*) (*support-value v p*)
using *support-middle-le-max-decreasing*[*OF pq qr True pos*] **.**
obtain *yr* **where** *yr-set*: *yr* $\in$ *set* (*scan-push* (*r* # *st*) *p*)
and *yr-ge*: *support-value v r* $\leq$ *support-value v yr*
using *tail-dom*[*OF True*, *of r*] **by** *auto*
obtain *yp* **where** *yp-set*: *yp* $\in$ *set* (*scan-push* (*r* # *st*) *p*)
and *yp-ge*: *support-value v p* $\leq$ *support-value v yp*
using *tail-dom*[*OF True*, *of p*] **by** *auto*
have *support-value v q* $\leq$ *support-value v r* $\vee$

```
      support-value v q ≤ support-value v p
    using q-le by (simp add: max-def split: if-splits)
  then show ?thesis
  proof
    assume support-value v q ≤ support-value v r
    then have support-value v q ≤ support-value v yr
      using yr-ge by (rule order-trans)
    then have support-value v x ≤ support-value v yr
      using True-x by simp
    then show ?thesis
      using True yr-set by (intro bexI[where x = yr]) simp-all
  next
    assume support-value v q ≤ support-value v p
    then have support-value v q ≤ support-value v yp
      using yp-ge by (rule order-trans)
    then have support-value v x ≤ support-value v yp
      using True-x by simp
    then show ?thesis
      using True yp-set by (intro bexI[where x = yp]) simp-all
  qed
 qed
qed

lemma scan-push-support-dominates-decreasing:
 fixes st :: real point list
 shows sorted-wrt (>) (rev st) ⟹
   (⋀q. q ∈ set st ⟹ p < q) ⟹
   0 < snd v ⟹
   x ∈ insert p (set st) ⟹
   ∃y∈set (scan-push st p). support-value v x ≤ support-value v y
proof (induction st p arbitrary: x rule: scan-push.induct)
 case (1 p)
 then show ?case by auto
next
 case (2 q p)
 then show ?case by auto
next
 case (3 q r st p)
 have sorted-tail: sorted-wrt (>) (rev (r # st))
   using 3.prems(1) by (simp add: sorted-wrt-append)
 have above-tail: ⋀w. w ∈ set (r # st) ⟹ p < w
   using 3.prems(2) by simp
 have pos: 0 < snd v
   using 3.prems(3) .
 have tail-dom:
   cross r q p ≤ 0 ⟹
     (⋀z. z ∈ insert p (set (r # st)) ⟹
       ∃y∈set (scan-push (r # st) p). support-value v z ≤ support-value v y)
   using 3.IH[OF - sorted-tail above-tail pos] by blast
```

```
  have sorted: sorted-wrt (>) (rev (q # r # st))
    using 3.prems(1) .
  have above: ⋀z. z ∈ set (q # r # st) ⟹ p < z
    using 3.prems(2) .
  have x-in: x ∈ insert p (set (q # r # st))
    using 3.prems(4) .
  show ?case
    by (rule scan-push-support-dominates-decreasing-step[
        OF tail-dom sorted above pos x-in])
qed

lemma scan-stack-support-dominates-decreasing:
  fixes xs :: real point list
  assumes sorted: sorted-wrt (>) xs
    and pos: 0 < snd v
    and x: x ∈ set xs
  shows ∃ y∈set (scan-stack xs). support-value v x ≤ support-value v y
  using sorted x
proof (induction xs arbitrary: x rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc p xs)
  have sorted-xs: sorted-wrt (>) xs
    using snoc.prems(1) by (simp add: sorted-wrt-append)
  have stack-sorted: sorted-wrt (>) (rev (scan-stack xs))
    using scan-stack-strict-sorted-decreasing[OF sorted-xs] .
  have above: ⋀q. q ∈ set (scan-stack xs) ⟹ p < q
  proof −
    fix q
    assume q ∈ set (scan-stack xs)
    then have q ∈ set xs
      using set-scan-stack-subset[of xs] by blast
    then show p < q
      using snoc.prems(1) by (simp add: sorted-wrt-append)
  qed
  have push-dom:
    ⋀z. z ∈ insert p (set (scan-stack xs)) ⟹
      ∃ y∈set (scan-push (scan-stack xs) p). support-value v z ≤ support-value v y
     using scan-push-support-dominates-decreasing[OF stack-sorted above pos] by
blast
  show ?case
  proof (cases x = p)
    case True
    then show ?thesis
      using push-dom[of p] by (simp add: scan-stack-def)
  next
    case False
    then have x-in-xs: x ∈ set xs
```

```
    using snoc.prems(2) by auto
  obtain z where z-set: z ∈ set (scan-stack xs)
    and z-ge: support-value v x ≤ support-value v z
    using snoc.IH[OF sorted-xs x-in-xs] by blast
  obtain y where y-set: y ∈ set (scan-push (scan-stack xs) p)
    and y-ge: support-value v z ≤ support-value v y
    using push-dom[of z] z-set by auto
  have xy-ge: support-value v x ≤ support-value v y
    by (rule order-trans[OF z-ge y-ge])
  show ?thesis
    using y-set xy-ge by (auto simp: scan-stack-def)
 qed
qed

lemma scan-chain-support-dominates-decreasing:
 fixes xs :: real point list
 assumes sorted-wrt (>) xs and 0 < snd v and x ∈ set xs
 shows ∃y∈set (scan-chain xs). support-value v x ≤ support-value v y
 using scan-stack-support-dominates-decreasing[OF assms]
 by (simp add: scan-chain-def)

lemma sorted-wrt-greater-rev-sorted-unique [simp]:
 sorted-wrt (>) (rev (sorted-unique (xs :: 'a::linorder list)))
 by (simp add: sorted-wrt-rev strict-sorted-iff)

lemma upper-hull-supports-positive:
 assumes 0 < snd v and x ∈ set ps
 shows ∃y∈set (upper-hull ps). support-value v x ≤ support-value v y
  using scan-chain-support-dominates-decreasing[of rev (sorted-unique ps) v x]
assms
 by (simp add: upper-hull-def)

theorem lower-hull-sorted:
 fixes ps :: real point list
 shows sorted-wrt (<) (lower-hull ps)
 using scan-stack-strict-sorted-increasing[of sorted-unique ps]
 by (simp add: lower-hull-def scan-chain-def)

theorem upper-hull-sorted:
 fixes ps :: real point list
 shows sorted-wrt (>) (upper-hull ps)
 using scan-stack-strict-sorted-decreasing[of rev (sorted-unique ps)]
 by (simp add: upper-hull-def scan-chain-def)

lemma fst-hd-sorted-unique-le:
 fixes x :: real point
 assumes x ∈ set ps
 shows fst (hd (sorted-unique ps)) ≤ fst x
proof −
```

```
  let ?xs = sorted-unique ps
  have xs-ne: ?xs ≠ []
    using assms by auto
  obtain h t where xs: ?xs = h # t
    using xs-ne by (cases ?xs) auto
  have sorted-xs: sorted ?xs
    by simp
  have x-set: x ∈ set ?xs
    using assms by simp
  have h ≤ x
  proof (cases x = h)
    case True
    then show ?thesis by simp
  next
    case False
    then have x ∈ set t
      using x-set xs by auto
    then show ?thesis
      using sorted-xs xs by (simp add: sorted-simps)
  qed
  then show ?thesis
    by (auto simp: xs less-eq-prod-def)
qed

lemma fst-le-last-sorted-unique:
  fixes x :: real point
  assumes x ∈ set ps
  shows fst x ≤ fst (last (sorted-unique ps))
proof −
  let ?xs = sorted-unique ps
  have xs-ne: ?xs ≠ []
    using assms by auto
  have xs-decomp: ?xs = butlast ?xs @ [last ?xs]
    using xs-ne by simp
  have x-set: x ∈ set ?xs
    using assms by simp
  have sorted-decomp: sorted (butlast ?xs @ [last ?xs])
    using xs-decomp by simp
  have x ≤ last ?xs
  proof (cases x = last ?xs)
    case True
    then show ?thesis by simp
  next
    case False
    have x ∈ set (butlast ?xs @ [last ?xs])
      using x-set xs-decomp by metis
    then have x ∈ set (butlast ?xs)
      using False by simp
    then show ?thesis
```

```
    using sorted-decomp by (simp add: sorted-append)
  qed
  then show ?thesis
    by (auto simp: less-eq-prod-def)
qed

lemma hd-sorted-unique-less:
  fixes x :: real point
  assumes x: x ∈ set ps
    and ne: x ≠ hd (sorted-unique ps)
  shows hd (sorted-unique ps) < x
proof −
  let ?xs = sorted-unique ps
  have xs-ne: ?xs ≠ []
    using x by auto
  obtain h t where xs: ?xs = h # t
    using xs-ne by (cases ?xs) auto
  have sorted-xs: sorted ?xs
    by simp
  have x-set: x ∈ set ?xs
    using x by simp
  have h ≤ x
  proof (cases x = h)
    case True
    then show ?thesis by simp
  next
    case False
    then have x ∈ set t
      using x-set xs by auto
    then show ?thesis
      using sorted-xs xs by (simp add: sorted-simps)
  qed
  moreover have h ≠ x
    using ne xs by simp
  ultimately show ?thesis
    using xs by simp
qed

lemma less-last-sorted-unique:
  fixes x :: real point
  assumes x: x ∈ set ps
    and ne: x ≠ last (sorted-unique ps)
  shows x < last (sorted-unique ps)
proof −
  let ?xs = sorted-unique ps
  have xs-ne: ?xs ≠ []
    using x by auto
  have xs-decomp: ?xs = butlast ?xs @ [last ?xs]
    using xs-ne by simp
```

```
  have x-set: x ∈ set ?xs
    using x by simp
  have sorted-decomp: sorted (butlast ?xs @ [last ?xs])
    using xs-decomp by simp
  have x ≤ last ?xs
  proof (cases x = last ?xs)
    case True
    then show ?thesis by simp
  next
    case False
    have x ∈ set (butlast ?xs @ [last ?xs])
      using x-set xs-decomp by metis
    then have x ∈ set (butlast ?xs)
      using False by simp
    then show ?thesis
      using sorted-decomp by (simp add: sorted-append)
  qed
  then show ?thesis
    using ne by simp
qed

lemma lex-min-strict-support:
  fixes p :: real point
  assumes fin: finite S
    and lex: ⋀q. q ∈ S − {p} ⟹ p < q
  shows ∃ v. ∀ q∈S − {p}. inner v q < inner v p
proof −
  let ?T = {q ∈ S − {p}. snd q < snd p}
  let ?R = (λq. (fst q − fst p) / (snd p − snd q)) ‘ ?T
  let ?M = insert 1 ?R
  define e :: real where e = Min ?M / 2
  have finite-T: finite ?T
    using fin by auto
  have finite-M: finite ?M
    using finite-T by auto
  have ratio-pos: 0 < (fst q − fst p) / (snd p − snd q) if q: q ∈ ?T for q
  proof −
    have p-lt-q: p < q
      using lex q by auto
    have fst-lt: fst p < fst q
      using p-lt-q q
      by (cases p; cases q) (auto simp: less-prod-def')
    have snd-pos: 0 < snd p − snd q
      using q by simp
    show ?thesis
      using fst-lt snd-pos by simp
  qed
  have M-pos: ∀ r∈?M. 0 < r
    using ratio-pos by auto
```

```
have min-pos: 0 < Min ?M
  using finite-M M-pos by (simp add: Min-gr-iff)
have e-pos: 0 < e
  using min-pos by (simp add: e-def)
have e-lt-ratio: e < (fst q − fst p) / (snd p − snd q) if q-T: q ∈ ?T for q
proof −
  have q-R: (fst q − fst p) / (snd p − snd q) ∈ ?M
    using finite-T q-T by auto
  have min-le: Min ?M ≤ (fst q − fst p) / (snd p − snd q)
    using Min-le[OF finite-M q-R] .
  have half-lt-min: Min ?M / 2 < Min ?M
    using min-pos by linarith
  have e-eq: e = Min ?M / 2
    by (simp add: e-def)
  show ?thesis
    using min-le half-lt-min e-eq by linarith
qed
have strict: inner (−1, − e) q < inner (−1, − e) p if q: q ∈ S − {p} for q
proof −
  have p-lt-q: p < q
    using lex[OF q] .
  have key: e ∗ (snd p − snd q) < fst q − fst p
  proof (cases snd q < snd p)
    case True
    then have q-T: q ∈ ?T
      using q by simp
    have snd-pos: 0 < snd p − snd q
      using True by simp
    show ?thesis
      using e-lt-ratio[OF q-T] snd-pos by (simp add: pos-less-divide-eq)
  next
    case False
    show ?thesis
    proof (cases fst p < fst q)
      case True
      have lhs-nonpos: e ∗ (snd p − snd q) ≤ 0
        using e-pos False by (intro mult-nonneg-nonpos) linarith+
      then show ?thesis
        using True by linarith
    next
      case False
      have fst-eq: fst p = fst q
        using p-lt-q False
        by (cases p; cases q) (auto simp: less-prod-def')
      have snd-lt: snd p < snd q
        using p-lt-q fst-eq
        by (cases p; cases q) (auto simp: less-prod-def')
      have lhs-neg: e ∗ (snd p − snd q) < 0
        using e-pos snd-lt by (intro mult-pos-neg) linarith+
```

```
      then show ?thesis
        using fst-eq by linarith
    qed
  qed
  note key = this
  have key': e * snd p - e * snd q < fst q - fst p
  proof -
    have e * (snd p - snd q) = e * snd p - e * snd q
      by (simp add: algebra-simps)
    then show ?thesis
      using key by linarith
  qed
  have q-sv: support-value (-1, - e) q = - fst q - e * snd q
    by (simp add: support-value-def)
  have p-sv: support-value (-1, - e) p = - fst p - e * snd p
    by (simp add: support-value-def)
  have support-value (-1, - e) q < support-value (-1, - e) p
    using key' q-sv p-sv by linarith
  then show ?thesis
    by (simp add: support-value-eq-inner)
qed
show ?thesis
  using strict by blast
qed

lemma lex-max-strict-support:
  fixes p :: real point
  assumes fin: finite S
    and lex: ⋀q. q ∈ S - {p} ⟹ q < p
  shows ∃ v. ∀ q∈S - {p}. inner v q < inner v p
proof -
  have neg-lex: - p < r if r: r ∈ (λx. - x) ` S - {- p} for r
  proof -
    obtain q where q: q ∈ S and r-eq: r = - q
      using r by auto
    have q-ne: q ≠ p
      using r r-eq by auto
    have q < p
      using lex[of q] q q-ne by auto
    then show ?thesis
      using r-eq by simp
  qed
  obtain v where v: ∀ r∈(λx. - x) ` S - {- p}. inner v r < inner v (- p)
    using lex-min-strict-support[of (λx. - x) ` S - p] fin neg-lex by auto
  have ∀ q∈S - {p}. inner (- v) q < inner (- v) p
  proof
    fix q
    assume q: q ∈ S - {p}
    then have - q ∈ (λx. - x) ` S - {- p}
```

```
    by auto
  then have inner v (− q) < inner v (− p)
    using v by blast
  then show inner (− v) q < inner (− v) p
    by (cases v; cases p; cases q) simp
 qed
 then show ?thesis
  by blast
qed

lemma lower-hull-edge-cross-nonneg-input-if-fst-less:
 fixes ps :: real point list
 defines L ≡ lower-hull ps
 assumes edge: Suc i < length L
  and fst-less: fst (L ! i) < fst (L ! Suc i)
  and x: x ∈ set ps
 shows 0 ≤ cross (L ! i) (L ! Suc i) x
proof −
 let ?a = L ! i
 let ?b = L ! Suc i
 let ?v = (snd ?b − snd ?a, fst ?a − fst ?b)
 have snd-v: snd ?v < 0
  using fst-less by simp
 obtain y where y: y ∈ set L
  and x-le-y: support-value ?v x ≤ support-value ?v y
  using lower-hull-supports-negative[OF snd-v x] L-def by blast
 have edge-lower: Suc i < length (lower-hull ps)
  using edge L-def by simp
 have y-lower: y ∈ set (lower-hull ps)
  using y L-def by simp
 have y-side: 0 ≤ cross ?a ?b y
  using sorted-chain-edge-cross-nonneg-increasing[
      OF lower-hull-sorted lower-hull-turns edge-lower y-lower] L-def
  by simp
 have y-eq: support-value ?v y = support-value ?v ?a − cross ?a ?b y
  by (rule support-value-edge-normal)
 have support-value ?v y ≤ support-value ?v ?a
  using y-side y-eq by linarith
 then have x-le-a: support-value ?v x ≤ support-value ?v ?a
  using x-le-y by linarith
 have x-eq: support-value ?v x = support-value ?v ?a − cross ?a ?b x
  by (rule support-value-edge-normal)
 show ?thesis
  using x-le-a x-eq by linarith
qed

lemma upper-hull-edge-cross-nonneg-input-if-fst-greater:
 fixes ps :: real point list
 defines U ≡ upper-hull ps
```

```
  assumes edge: Suc i < length U
    and fst-greater: fst (U ! Suc i) < fst (U ! i)
    and x: x ∈ set ps
  shows 0 ≤ cross (U ! i) (U ! Suc i) x
proof −
  let ?a = U ! i
  let ?b = U ! Suc i
  let ?v = (snd ?b − snd ?a, fst ?a − fst ?b)
  have snd-v: 0 < snd ?v
    using fst-greater by simp
  obtain y where y: y ∈ set U
    and x-le-y: support-value ?v x ≤ support-value ?v y
    using upper-hull-supports-positive[OF snd-v x] U-def by blast
  have edge-upper: Suc i < length (upper-hull ps)
    using edge U-def by simp
  have y-upper: y ∈ set (upper-hull ps)
    using y U-def by simp
  have y-side: 0 ≤ cross ?a ?b y
    using sorted-chain-edge-cross-nonneg-decreasing[
        OF upper-hull-sorted upper-hull-turns edge-upper y-upper] U-def
    by simp
  have y-eq: support-value ?v y = support-value ?v ?a − cross ?a ?b y
    by (rule support-value-edge-normal)
  have support-value ?v y ≤ support-value ?v ?a
    using y-side y-eq by linarith
  then have x-le-a: support-value ?v x ≤ support-value ?v ?a
    using x-le-y by linarith
  have x-eq: support-value ?v x = support-value ?v ?a − cross ?a ?b x
    by (rule support-value-edge-normal)
  show ?thesis
    using x-le-a x-eq by linarith
qed

lemma lower-hull-vertical-edge-last:
  fixes ps :: real point list
  defines L ≡ lower-hull ps
  assumes edge: Suc i < length L
    and same-x: fst (L ! i) = fst (L ! Suc i)
  shows Suc i = length L − 1
proof (rule ccontr)
  assume not-last: Suc i ≠ length L − 1
  have next-len: Suc (Suc i) < length L
    using edge not-last by linarith
  have sorted-L: sorted-wrt (<) L
    using lower-hull-sorted L-def by simp
  have turns-L: chain-turns L
    using lower-hull-turns L-def by simp
  have a-lt-b: L ! i < L ! Suc i
    using sorted-wrt-nth-less[OF sorted-L, of i Suc i] edge by simp
```

```
  have b-lt-c: L ! Suc i < L ! Suc (Suc i)
    using sorted-wrt-nth-less[OF sorted-L, of Suc i Suc (Suc i)] next-len by simp
  have snd-lt: snd (L ! i) < snd (L ! Suc i)
    using a-lt-b same-x
    by (cases L ! i; cases L ! Suc i) (auto simp: less-prod-def')
  have fst-le: fst (L ! Suc i) ≤ fst (L ! Suc (Suc i))
    using b-lt-c
    by (cases L ! Suc i; cases L ! Suc (Suc i))
      (auto simp: less-prod-def' less-eq-prod-def)
  have prod-nonneg:
    0 ≤ (snd (L ! Suc i) − snd (L ! i)) *
        (fst (L ! Suc (Suc i)) − fst (L ! i))
    using snd-lt fst-le same-x by (intro mult-nonneg-nonneg) linarith+
  have cross-nonpos: cross (L ! i) (L ! Suc i) (L ! Suc (Suc i)) ≤ 0
    using same-x prod-nonneg
    by (cases L ! i; cases L ! Suc i; cases L ! Suc (Suc i))
      (simp add: cross-def algebra-simps)
  have 0 < cross (L ! i) (L ! Suc i) (L ! Suc (Suc i))
     by (rule sorted-chain-cross-nth-increasing[OF sorted-L turns-L - - next-len])
simp-all
  then show False
    using cross-nonpos by linarith
qed

lemma upper-hull-vertical-edge-last:
  fixes ps :: real point list
  defines U ≡ upper-hull ps
  assumes edge: Suc i < length U
    and same-x: fst (U ! i) = fst (U ! Suc i)
  shows Suc i = length U − 1
proof (rule ccontr)
  assume not-last: Suc i ≠ length U − 1
  have next-len: Suc (Suc i) < length U
    using edge not-last by linarith
  have sorted-U: sorted-wrt (>) U
    using upper-hull-sorted U-def by simp
  have turns-U: chain-turns U
    using upper-hull-turns U-def by simp
  have b-lt-a: U ! Suc i < U ! i
    using sorted-wrt-nth-less[OF sorted-U, of i Suc i] edge by simp
  have c-lt-b: U ! Suc (Suc i) < U ! Suc i
    using sorted-wrt-nth-less[OF sorted-U, of Suc i Suc (Suc i)] next-len by simp
  have snd-lt: snd (U ! Suc i) < snd (U ! i)
    using b-lt-a same-x
    by (cases U ! i; cases U ! Suc i) (auto simp: less-prod-def')
  have fst-le: fst (U ! Suc (Suc i)) ≤ fst (U ! Suc i)
    using c-lt-b
    by (cases U ! Suc (Suc i); cases U ! Suc i)
      (auto simp: less-prod-def' less-eq-prod-def)
```

```
  have prod-nonneg:
    0 ≤ (snd (U ! Suc i) − snd (U ! i)) ∗
        (fst (U ! Suc (Suc i)) − fst (U ! i))
    using snd-lt fst-le same-x by (intro mult-nonpos-nonpos) linarith+
  have cross-nonpos: cross (U ! i) (U ! Suc i) (U ! Suc (Suc i)) ≤ 0
    using same-x prod-nonneg
    by (cases U ! i; cases U ! Suc i; cases U ! Suc (Suc i))
      (simp add: cross-def algebra-simps)
  have 0 < cross (U ! i) (U ! Suc i) (U ! Suc (Suc i))
    by (rule sorted-chain-cross-nth-decreasing[OF sorted-U turns-U - - next-len])
simp-all
  then show False
    using cross-nonpos by linarith
qed

lemma lower-hull-edge-cross-nonneg-input:
  fixes ps :: real point list
  defines L ≡ lower-hull ps
  assumes edge: Suc i < length L
    and x: x ∈ set ps
  shows 0 ≤ cross (L ! i) (L ! Suc i) x
proof (cases fst (L ! i) < fst (L ! Suc i))
  case True
  then show ?thesis
    using lower-hull-edge-cross-nonneg-input-if-fst-less[where ps=ps and i=i and
x=x]
      edge x L-def
    by simp
next
  case False
  have sorted-L: sorted-wrt (<) L
    using lower-hull-sorted L-def by simp
  have a-lt-b: L ! i < L ! Suc i
    using sorted-wrt-nth-less[OF sorted-L, of i Suc i] edge by simp
  have same-x: fst (L ! i) = fst (L ! Suc i)
    using a-lt-b False
    by (cases L ! i; cases L ! Suc i) (auto simp: less-prod-def')
  have snd-lt: snd (L ! i) < snd (L ! Suc i)
    using a-lt-b same-x
    by (cases L ! i; cases L ! Suc i) (auto simp: less-prod-def')
  have last-edge: Suc i = length L − 1
    using lower-hull-vertical-edge-last[where ps=ps and i=i] edge same-x L-def
by simp
  have ps-ne: ps ≠ []
    using x by auto
  have b-last: L ! Suc i = last L
  proof −
    have L-ne: L ≠ []
      using edge by auto
```

```
    have last L = L ! (length L − 1)
      using L-ne by (rule last-conv-nth)
    then show ?thesis
      using last-edge by simp
  qed
  have b-su: L ! Suc i = last (sorted-unique ps)
    using b-last last-lower-hull[OF ps-ne] L-def by simp
  have fst-x-le: fst x ≤ fst (L ! Suc i)
    using fst-le-last-sorted-unique[OF x] b-su by simp
  have prod-nonpos:
    (snd (L ! Suc i) − snd (L ! i)) * (fst x − fst (L ! i)) ≤ 0
    using snd-lt fst-x-le same-x by (intro mult-nonneg-nonpos) linarith+
  show ?thesis
    using same-x prod-nonpos
    by (cases L ! i; cases L ! Suc i; cases x)
      (simp add: cross-def algebra-simps)
qed

lemma upper-hull-edge-cross-nonneg-input:
  fixes ps :: real point list
  defines U ≡ upper-hull ps
  assumes edge: Suc i < length U
    and x: x ∈ set ps
  shows 0 ≤ cross (U ! i) (U ! Suc i) x
proof (cases fst (U ! Suc i) < fst (U ! i))
  case True
  then show ?thesis
    using upper-hull-edge-cross-nonneg-input-if-fst-greater[where ps=ps and i=i
and x=x]
      edge x U-def
    by simp
next
  case False
  have sorted-U: sorted-wrt (>) U
    using upper-hull-sorted U-def by simp
  have b-lt-a: U ! Suc i < U ! i
    using sorted-wrt-nth-less[OF sorted-U, of i Suc i] edge by simp
  have same-x: fst (U ! i) = fst (U ! Suc i)
    using b-lt-a False
    by (cases U ! i; cases U ! Suc i) (auto simp: less-prod-def')
  have snd-lt: snd (U ! Suc i) < snd (U ! i)
    using b-lt-a same-x
    by (cases U ! i; cases U ! Suc i) (auto simp: less-prod-def')
  have last-edge: Suc i = length U − 1
    using upper-hull-vertical-edge-last[where ps=ps and i=i] edge same-x U-def
by simp
  have ps-ne: ps ≠ []
    using x by auto
  have b-last: U ! Suc i = last U
```

```
  proof −
    have U-ne: U ≠ []
      using edge by auto
    have last U = U ! (length U − 1)
      using U-ne by (rule last-conv-nth)
    then show ?thesis
      using last-edge by simp
  qed
  have b-su: U ! Suc i = hd (sorted-unique ps)
    using b-last last-upper-hull[OF ps-ne] U-def by simp
  have fst-le-x: fst (U ! Suc i) ≤ fst x
    using fst-hd-sorted-unique-le[OF x] b-su by simp
  have prod-nonpos:
    (snd (U ! Suc i) − snd (U ! i)) * (fst x − fst (U ! i)) ≤ 0
    using snd-lt fst-le-x same-x by (intro mult-nonpos-nonneg) linarith+
  show ?thesis
    using same-x prod-nonpos
    by (cases U ! i; cases U ! Suc i; cases x)
      (simp add: cross-def algebra-simps)
qed

lemma strict-support-from-two-edges:
  fixes a p c :: real point
  assumes left: ⋀q. q ∈ S ⟹ 0 ≤ cross a p q
    and right: ⋀q. q ∈ S ⟹ 0 ≤ cross p c q
    and noncol: cross a p c ≠ 0
  shows ∃ v. ∀ q∈S − {p}. inner v q < inner v p
proof −
  let ?v1 = (snd p − snd a, fst a − fst p)
  let ?v2 = (snd c − snd p, fst p − fst c)
  let ?v = ?v1 + ?v2
  have strict: inner ?v q < inner ?v p if q: q ∈ S − {p} for q
  proof −
    have q-S: q ∈ S and q-ne: q ≠ p
      using q by auto
    have c1-nonneg: 0 ≤ cross a p q
      by (rule left[OF q-S])
    have c2-nonneg: 0 ≤ cross p c q
      by (rule right[OF q-S])
    have not-both-zero: cross a p q ≠ 0 ∨ cross p c q ≠ 0
    proof (rule ccontr)
      assume ¬ (cross a p q ≠ 0 ∨ cross p c q ≠ 0)
      then have z1: cross a p q = 0 and z2: cross p c q = 0
        by auto
      have q = p
        by (rule two-cross-zero-imp-eq-middle[OF z1 z2 noncol])
      then show False
        using q-ne by simp
    qed
```

```
    have sum-pos: 0 < cross a p q + cross p c q
      using c1-nonneg c2-nonneg not-both-zero by linarith
    have q1: support-value ?v1 q = support-value ?v1 a − cross a p q
      by (rule support-value-edge-normal)
    have p1-eq: support-value ?v1 p = support-value ?v1 a − cross a p p
      by (rule support-value-edge-normal)
    have p1: support-value ?v1 p = support-value ?v1 a
      using p1-eq by simp
    have v1-diff: support-value ?v1 q = support-value ?v1 p − cross a p q
      using q1 p1 by simp
    have v2-diff: support-value ?v2 q = support-value ?v2 p − cross p c q
      by (rule support-value-edge-normal)
    have sum-q: support-value ?v q = support-value ?v1 q + support-value ?v2 q
     by (cases ?v1; cases ?v2; cases q) (simp add: support-value-def algebra-simps)
    have sum-p: support-value ?v p = support-value ?v1 p + support-value ?v2 p
     by (cases ?v1; cases ?v2; cases p) (simp add: support-value-def algebra-simps)
    show ?thesis
     using v1-diff v2-diff sum-q sum-p sum-pos by (simp add: support-value-eq-inner)
  qed
  show ?thesis
    using strict by blast
qed

lemma lower-hull-interior-strict-support:
  fixes ps :: real point list
  defines L ≡ lower-hull ps
  assumes i-pos: 0 < i
    and i-len: Suc i < length L
  shows ∃ v. ∀ q∈set ps − {L ! i}. inner v q < inner v (L ! i)
proof −
  let ?a = L ! (i − 1)
  let ?p = L ! i
  let ?c = L ! Suc i
  have edge-left: Suc (i − 1) < length L
    using i-pos i-len by simp
  have Suc-pred-i: Suc (i − 1) = i
    using i-pos by simp
  have turn: 0 < cross ?a ?p ?c
    using sorted-chain-cross-nth-increasing[
        OF lower-hull-sorted lower-hull-turns, of i − 1 i Suc i] i-pos i-len L-def
    by simp
  show ?thesis
  proof (rule strict-support-from-two-edges[where S=set ps and a=?a and p=?p
and c=?c])
    fix q
    assume q: q ∈ set ps
    show 0 ≤ cross ?a ?p q
      using lower-hull-edge-cross-nonneg-input[where ps=ps and i=i − 1 and
x=q]
```

```
      edge-left q L-def Suc-pred-i
    by simp
  show 0 ≤ cross ?p ?c q
    using lower-hull-edge-cross-nonneg-input[where ps=ps and i=i and x=q]
i-len q L-def
    by simp
 next
  show cross ?a ?p ?c ≠ 0
    using turn by linarith
 qed
qed

lemma upper-hull-interior-strict-support:
 fixes ps :: real point list
 defines U ≡ upper-hull ps
 assumes i-pos: 0 < i
  and i-len: Suc i < length U
 shows ∃ v. ∀ q∈set ps − {U ! i}. inner v q < inner v (U ! i)
proof −
 let ?a = U ! (i − 1)
 let ?p = U ! i
 let ?c = U ! Suc i
 have edge-left: Suc (i − 1) < length U
  using i-pos i-len by simp
 have Suc-pred-i: Suc (i − 1) = i
  using i-pos by simp
 have turn: 0 < cross ?a ?p ?c
  using sorted-chain-cross-nth-decreasing[
      OF upper-hull-sorted upper-hull-turns, of i − 1 i Suc i] i-pos i-len U-def
  by simp
 show ?thesis
 proof (rule strict-support-from-two-edges[where S=set ps and a=?a and p=?p
and c=?c])
  fix q
  assume q: q ∈ set ps
  show 0 ≤ cross ?a ?p q
    using upper-hull-edge-cross-nonneg-input[where ps=ps and i=i − 1 and
x=q]
      edge-left q U-def Suc-pred-i
    by simp
  show 0 ≤ cross ?p ?c q
    using upper-hull-edge-cross-nonneg-input[where ps=ps and i=i and x=q]
i-len q U-def
    by simp
 next
  show cross ?a ?p ?c ≠ 0
    using turn by linarith
 qed
qed
```

lemma *lower-hull-interior-strict-support-if-fst-less*:
fixes *ps* :: *real point list*
defines $L \equiv$ *lower-hull ps*
assumes *i-pos*: $0 < i$
and *i-len*: $Suc\ i < length\ L$
and *fst-left*: $fst\ (L\ !\ (i - 1)) < fst\ (L\ !\ i)$
and *fst-right*: $fst\ (L\ !\ i) < fst\ (L\ !\ Suc\ i)$
shows $\exists v.\ \forall q \in set\ ps - \{L\ !\ i\}.\ inner\ v\ q < inner\ v\ (L\ !\ i)$
proof −
let $?a = L\ !\ (i - 1)$
let $?p = L\ !\ i$
let $?c = L\ !\ Suc\ i$
let $?v1 = (snd\ ?p - snd\ ?a,\ fst\ ?a - fst\ ?p)$
let $?v2 = (snd\ ?c - snd\ ?p,\ fst\ ?p - fst\ ?c)$
let $?v = ?v1 + ?v2$
have *edge-left*: $Suc\ (i - 1) < length\ L$
using *i-pos i-len* **by** *simp*
have *Suc-pred-i*: $Suc\ (i - 1) = i$
using *i-pos* **by** *simp*
have *turn*: $0 < cross\ ?a\ ?p\ ?c$
using *sorted-chain-cross-nth-increasing*[
OF lower-hull-sorted lower-hull-turns, *of* $i - 1$ *i Suc i*] *i-pos i-len L-def*
by *simp*
have *strict*: $inner\ ?v\ q < inner\ ?v\ ?p$ **if** *q*: $q \in set\ ps - \{?p\}$ **for** *q*
proof −
have *q-ps*: $q \in set\ ps$ **and** *q-ne*: $q \neq ?p$
using *q* **by** *auto*
have *c1-nonneg*: $0 \leq cross\ ?a\ ?p\ q$
using *lower-hull-edge-cross-nonneg-input-if-fst-less*[
where *ps=ps* **and** *i=i* − *1* **and** *x=q*] *edge-left fst-left q-ps L-def*
using *Suc-pred-i* **by** *simp*
have *c2-nonneg*: $0 \leq cross\ ?p\ ?c\ q$
using *lower-hull-edge-cross-nonneg-input-if-fst-less*[
where *ps=ps* **and** *i=i* **and** *x=q*] *i-len fst-right q-ps L-def*
by *simp*
have *not-both-zero*: $cross\ ?a\ ?p\ q \neq 0 \vee cross\ ?p\ ?c\ q \neq 0$
proof (*rule ccontr*)
assume $\neg\ (cross\ ?a\ ?p\ q \neq 0 \vee cross\ ?p\ ?c\ q \neq 0)$
then have *z1*: $cross\ ?a\ ?p\ q = 0$ **and** *z2*: $cross\ ?p\ ?c\ q = 0$
by *auto*
have *nz*: $cross\ ?a\ ?p\ ?c \neq 0$
using *turn* **by** *linarith*
have $q = ?p$
by (*rule two-cross-zero-imp-eq-middle*[*OF z1 z2 nz*])
then show *False*
using *q-ne* **by** *simp*
qed
have *sum-pos*: $0 < cross\ ?a\ ?p\ q + cross\ ?p\ ?c\ q$

```
    using c1-nonneg c2-nonneg not-both-zero by linarith
  have q1: support-value ?v1 q = support-value ?v1 ?a − cross ?a ?p q
    by (rule support-value-edge-normal)
  have p1-eq: support-value ?v1 ?p = support-value ?v1 ?a − cross ?a ?p ?p
    by (rule support-value-edge-normal)
  have p1: support-value ?v1 ?p = support-value ?v1 ?a
    using p1-eq by simp
  have v1-diff: support-value ?v1 q = support-value ?v1 ?p − cross ?a ?p q
    using q1 p1 by simp
  have v2-diff: support-value ?v2 q = support-value ?v2 ?p − cross ?p ?c q
    by (rule support-value-edge-normal)
  have sum-q: support-value ?v q = support-value ?v1 q + support-value ?v2 q
   by (cases ?v1; cases ?v2; cases q) (simp add: support-value-def algebra-simps)
  have sum-p: support-value ?v ?p = support-value ?v1 ?p + support-value ?v2
?p
   by (cases ?v1; cases ?v2; cases ?p) (simp add: support-value-def algebra-simps)
  show ?thesis
   using v1-diff v2-diff sum-q sum-p sum-pos by (simp add: support-value-eq-inner)
 qed
 show ?thesis
  using strict by blast
qed

lemma hd-sorted-unique-mem-andrew-hull:
 assumes ps ≠ []
 shows hd (sorted-unique ps) ∈ set (andrew-hull ps)
proof −
 have lower-ne: lower-hull ps ≠ []
  using assms lower-hull-nonempty by blast
 then have hd (lower-hull ps) ∈ set (lower-hull ps)
  by (cases lower-hull ps) auto
 then show ?thesis
  using hd-lower-hull[OF assms] set-andrew-hull[of ps] by auto
qed

lemma last-sorted-unique-mem-andrew-hull:
 assumes ps ≠ []
 shows last (sorted-unique ps) ∈ set (andrew-hull ps)
proof −
 have lower-ne: lower-hull ps ≠ []
  using assms lower-hull-nonempty by blast
 then have last (lower-hull ps) ∈ set (lower-hull ps)
  by (cases lower-hull ps) auto
 then show ?thesis
  using last-lower-hull[OF assms] set-andrew-hull[of ps] by auto
qed

lemma andrew-hull-supports-horizontal:
 assumes x: x ∈ set ps and horizontal: snd v = 0
```

shows $\exists y{\in}set\ (andrew\text{-}hull\ ps).\ support\text{-}value\ v\ x \leq support\text{-}value\ v\ y$
proof (*cases* $0 \leq fst\ v$)
case *True*
let *?y* = *last* (*sorted-unique ps*)
have *ps-ne*: $ps \neq []$
using *x* **by** *auto*
have *y-set*: *?y* ∈ *set* (*andrew-hull ps*)
using *last-sorted-unique-mem-andrew-hull*[*OF ps-ne*] **.**
have *fst-le*: *fst x* ≤ *fst ?y*
using *fst-le-last-sorted-unique*[*OF x*] **.**
have *support-value v x* ≤ *support-value v ?y*
using *True fst-le horizontal*
by (*simp add*: *support-value-def mult-left-mono*)
then show *?thesis*
using *y-set* **by** *blast*
next
case *False*
let *?y* = *hd* (*sorted-unique ps*)
have *ps-ne*: $ps \neq []$
using *x* **by** *auto*
have *y-set*: *?y* ∈ *set* (*andrew-hull ps*)
using *hd-sorted-unique-mem-andrew-hull*[*OF ps-ne*] **.**
have *fst-le*: *fst ?y* ≤ *fst x*
using *fst-hd-sorted-unique-le*[*OF x*] **.**
have *fst-neg*: *fst v* < *0*
using *False* **by** *simp*
have *support-value v x* ≤ *support-value v ?y*
using *fst-le fst-neg horizontal*
by (*simp add*: *support-value-def mult-left-mono-neg*)
then show *?thesis*
using *y-set* **by** *blast*
qed

lemma *andrew-hull-supports*:
assumes *x*: *x* ∈ *set ps*
shows $\exists y{\in}set\ (andrew\text{-}hull\ ps).\ support\text{-}value\ v\ x \leq support\text{-}value\ v\ y$
proof (*cases snd v* < *0*)
case *True*
obtain *y* **where** *y* ∈ *set* (*lower-hull ps*)
and *support-value v x* ≤ *support-value v y*
using *lower-hull-supports-negative*[*OF True x*] **by** *blast*
then show *?thesis*
using *set-andrew-hull*[*of ps*] **by** *auto*
next
case *False*
then show *?thesis*
proof (*cases* $0 < snd\ v$)
case *True*
obtain *y* **where** *y* ∈ *set* (*upper-hull ps*)

```
    and support-value v x ≤ support-value v y
    using upper-hull-supports-positive[OF True x] by blast
  then show ?thesis
    using set-andrew-hull[of ps] by auto
 next
  case False
  then have snd v = 0
    using ‹¬ snd v < 0› by simp
  then show ?thesis
    using andrew-hull-supports-horizontal[OF x] by blast
 qed
qed

declare support-value-eq-inner [simp]

theorem andrew-hull-covers-input:
 fixes ps :: real point list
 shows set ps ⊆ convex hull set (andrew-hull ps)
proof
 fix x
 assume x: x ∈ set ps
 let ?S = convex hull set (andrew-hull ps)
 show x ∈ ?S
 proof (rule ccontr)
  assume x-not: x ∉ ?S
  have closed-S: closed ?S
    using finite-imp-compact-convex-hull[of set (andrew-hull ps)]
    by (simp add: compact-imp-closed)
  obtain a b where ax-lt: inner a x < b
    and S-gt: ⋀y. y ∈ ?S ⟹ b < inner a y
   using separating-hyperplane-closed-point[OF convex-convex-hull closed-S x-not]
    by auto
  obtain y where y-set: y ∈ set (andrew-hull ps)
    and support: support-value (− a) x ≤ support-value (− a) y
    using andrew-hull-supports[OF x, of − a] by auto
  have y-in-S: y ∈ ?S
    using y-set by (simp add: hull-inc)
  have inner a y ≤ inner a x
    using support by simp
  then show False
    using ax-lt S-gt[OF y-in-S] by linarith
 qed
qed

theorem andrew-hull-correct:
 fixes ps :: real point list
 shows convex-hull-correct ps (andrew-hull ps)
 using andrew-hull-covers-input[of ps]
 by (rule andrew-hull-correctI)
```

```
lemma andrew-hull-point-strict-support:
  fixes ps :: real point list
  assumes p: p ∈ set (andrew-hull ps)
  shows ∃ v. ∀ q∈set (andrew-hull ps) − {p}. inner v q < inner v p
proof −
  let ?S = set (andrew-hull ps)
  let ?lo = hd (sorted-unique ps)
  let ?hi = last (sorted-unique ps)
  have p-ps: p ∈ set ps
    using p andrew-hull-subset[of ps] by auto
  then have ps-ne: ps ≠ []
    by auto
  show ?thesis
  proof (cases p = ?lo)
    case True
    have ∃ v. ∀ q∈?S − {?lo}. inner v q < inner v ?lo
    proof (rule lex-min-strict-support)
      show finite ?S
        by simp
      fix q
      assume q: q ∈ ?S − {?lo}
      then have q-ps: q ∈ set ps
        using andrew-hull-subset[of ps] by auto
      have q-ne: q ≠ ?lo
        using q by auto
      show ?lo < q
        by (rule hd-sorted-unique-less[OF q-ps q-ne])
    qed
    then show ?thesis
      using True by simp
  next
    case p-ne-lo: False
    show ?thesis
    proof (cases p = ?hi)
      case True
      have ∃ v. ∀ q∈?S − {?hi}. inner v q < inner v ?hi
      proof (rule lex-max-strict-support)
        show finite ?S
          by simp
        fix q
        assume q: q ∈ ?S − {?hi}
        then have q-ps: q ∈ set ps
          using andrew-hull-subset[of ps] by auto
        have q-ne: q ≠ ?hi
          using q by auto
        show q < ?hi
          by (rule less-last-sorted-unique[OF q-ps q-ne])
      qed
```

```
    then show ?thesis
      using True by simp
  next
    case p-ne-hi: False
    have p-union: p ∈ set (lower-hull ps) ∨ p ∈ set (upper-hull ps)
      using p set-andrew-hull[of ps] by auto
    show ?thesis
    proof (cases p ∈ set (lower-hull ps))
      case True
      let ?L = lower-hull ps
      obtain i where i-len: i < length ?L and p-i: p = ?L ! i
        using True by (auto simp: in-set-conv-nth)
      have i-pos: 0 < i
      proof (rule ccontr)
        assume ¬ 0 < i
        then have i0: i = 0
          by simp
        have p = hd ?L
          using p-i i-len i0 by (cases ?L) auto
        also have ... = ?lo
          using hd-lower-hull[OF ps-ne] .
        finally show False
          using p-ne-lo by simp
      qed
      have i-suc: Suc i < length ?L
      proof (rule ccontr)
        assume not-suc: ¬ Suc i < length ?L
        have i-last: i = length ?L − 1
          using i-len not-suc by linarith
        have L-ne: ?L ≠ []
          using i-len by auto
        have last ?L = ?L ! (length ?L − 1)
          using L-ne by (rule last-conv-nth)
        then have p = last ?L
          using p-i i-last by simp
        also have ... = ?hi
          using last-lower-hull[OF ps-ne] .
        finally show False
          using p-ne-hi by simp
      qed
      obtain v where strict-ps: ∀ q∈set ps − {p}. inner v q < inner v p
       using lower-hull-interior-strict-support[where ps=ps and i=i] i-pos i-suc
p-i
        by auto
      have ∀ q∈?S − {p}. inner v q < inner v p
        using strict-ps andrew-hull-subset[of ps] by auto
      then show ?thesis
        by blast
    next
```

```
      case False
      then have p-upper: p ∈ set (upper-hull ps)
        using p-union by auto
      let ?U = upper-hull ps
      obtain i where i-len: i < length ?U and p-i: p = ?U ! i
        using p-upper by (auto simp: in-set-conv-nth)
      have i-pos: 0 < i
      proof (rule ccontr)
        assume ¬ 0 < i
        then have i0: i = 0
          by simp
        have p = hd ?U
          using p-i i-len i0 by (cases ?U) auto
        also have ... = ?hi
          using hd-upper-hull[OF ps-ne] .
        finally show False
          using p-ne-hi by simp
      qed
      have i-suc: Suc i < length ?U
      proof (rule ccontr)
        assume not-suc: ¬ Suc i < length ?U
        have i-last: i = length ?U − 1
          using i-len not-suc by linarith
        have U-ne: ?U ≠ []
          using i-len by auto
        have last ?U = ?U ! (length ?U − 1)
          using U-ne by (rule last-conv-nth)
        then have p = last ?U
          using p-i i-last by simp
        also have ... = ?lo
          using last-upper-hull[OF ps-ne] .
        finally show False
          using p-ne-lo by simp
      qed
      obtain v where strict-ps: ∀ q∈set ps − {p}. inner v q < inner v p
       using upper-hull-interior-strict-support[where ps=ps and i=i] i-pos i-suc
p-i
        by auto
      have ∀ q∈?S − {p}. inner v q < inner v p
        using strict-ps andrew-hull-subset[of ps] by auto
      then show ?thesis
        by blast
    qed
   qed
  qed
qed

theorem andrew-hull-delete-changes-convex-hull:
  fixes ps :: real point list
```

```
  assumes p: p ∈ set (andrew-hull ps)
  shows convex hull (set (andrew-hull ps) − {p}) ≠ convex hull set (andrew-hull
ps)
proof −
  obtain v where strict: ∀ q∈set (andrew-hull ps) − {p}. inner v q < inner v p
    using andrew-hull-point-strict-support[OF p] by blast
  show ?thesis
    by (rule strict-support-hull-delete-ne[where p=p and S=set (andrew-hull ps)
and v=v])
      (use p strict in auto)
qed

theorem andrew-hull-irredundant:
  fixes ps :: real point list
  shows convex-hull-irredundant (andrew-hull ps)
  unfolding convex-hull-irredundant-def
  using andrew-hull-delete-changes-convex-hull[of - ps] by blast

lemma length-scan-push-le:
  length (scan-push st p) ≤ Suc (length st)
  by (induction st p rule: scan-push.induct) auto

lemma length-scan-stack-le:
  length (scan-stack ps) ≤ length ps
  unfolding scan-stack-def
proof (induction ps arbitrary: rule: rev-induct)
  case Nil
  then show ?case by simp
next
  case (snoc p ps)
  have length (foldl scan-push [] (ps @ [p])) ≤ Suc (length (foldl scan-push [] ps))
    using length-scan-push-le by simp
  also have ... ≤ Suc (length ps)
    using snoc.IH by simp
  finally show ?case by simp
qed

lemma length-scan-chain-le:
  length (scan-chain ps) ≤ length ps
  by (simp add: scan-chain-def length-scan-stack-le)

lemma length-lower-hull-le:
  length (lower-hull ps) ≤ card (set ps)
proof −
  have length (lower-hull ps) ≤ length (sorted-unique ps)
    using length-scan-chain-le[of sorted-unique ps]
    by (simp add: lower-hull-def)
  also have ... = card (set ps)
    by (simp add: sorted-unique-def length-sorted-list-of-set)
```

```
  finally show ?thesis .
qed

lemma length-upper-hull-le:
  length (upper-hull ps) ≤ card (set ps)
proof −
  have length (upper-hull ps) ≤ length (rev (sorted-unique ps))
    using length-scan-chain-le[of rev (sorted-unique ps)]
    by (simp add: upper-hull-def)
  also have ... = card (set ps)
    by (simp add: sorted-unique-def length-sorted-list-of-set)
  finally show ?thesis .
qed

theorem length-andrew-hull-le-twice-card:
  length (andrew-hull ps) ≤ 2 * card (set ps)
proof (cases sorted-unique ps)
  case Nil
  then have ps = []
    by simp
  then show ?thesis
    by (simp add: andrew-hull-def sorted-unique-def)
next
  case (Cons p qs)
  then show ?thesis
  proof (cases qs)
    case Nil
    with Cons have su: sorted-unique ps = [p]
      by simp
    then have set ps = {p}
      using sorted-unique-singleton-iff by blast
    moreover have andrew-hull ps = [p]
      using su by (simp add: andrew-hull-def)
    ultimately show ?thesis
      by simp
  next
    case (Cons q rs)
    have length (andrew-hull ps) ≤ length (lower-hull ps) + length (upper-hull ps)
      using Cons ‹sorted-unique ps = p # qs›
      by (simp add: andrew-hull-def)
    also have ... ≤ card (set ps) + card (set ps)
      using length-lower-hull-le[of ps] length-upper-hull-le[of ps] by simp
    finally show ?thesis by simp
  qed
qed
```

For inputs with at least two distinct points, the implementation returns exactly the standard Andrew concatenation of lower and upper scans. This theorem is often the most convenient way to unfold the top-level algorithm

without exposing the special empty and singleton cases.

theorem *andrew-hull-ge2*:
  assumes *card* (*set ps*) ≥ *2*
  shows *andrew-hull ps* = *butlast* (*lower-hull ps*) @ *butlast* (*upper-hull ps*)
proof −
  have *sorted-unique ps* ≠ [] **and** ⋀*p*. *sorted-unique ps* ≠ [*p*]
    using *assms* **by** (*auto simp*: *sorted-unique-singleton-iff card-Suc-eq*)
  then show *?thesis*
    by (*cases sorted-unique ps*) (*auto simp*: *andrew-hull-def split*: *list.splits*)
qed

theorem *andrew-hull-ordered-chains*:
  fixes *ps* :: *real point list*
  assumes *2* ≤ *card* (*set ps*)
  shows *andrew-hull ps* = *butlast* (*lower-hull ps*) @ *butlast* (*upper-hull ps*)
    and *sorted-wrt* (<) (*lower-hull ps*)
    and *sorted-wrt* (>) (*upper-hull ps*)
    and *chain-turns* (*lower-hull ps*)
    and *chain-turns* (*upper-hull ps*)
proof −
  show *andrew-hull ps* = *butlast* (*lower-hull ps*) @ *butlast* (*upper-hull ps*)
    using *andrew-hull-ge2*[*OF assms*] **.**
  show *sorted-wrt* (<) (*lower-hull ps*)
    by (*rule lower-hull-sorted*)
  show *sorted-wrt* (>) (*upper-hull ps*)
    by (*rule upper-hull-sorted*)
  show *chain-turns* (*lower-hull ps*)
    by (*rule lower-hull-turns*)
  show *chain-turns* (*upper-hull ps*)
    by (*rule upper-hull-turns*)
qed

theorem *distinct-andrew-hull-iff*:
  fixes *ps* :: *real point list*
  shows *distinct* (*andrew-hull ps*) ⟷
    card (*set ps*) < *2* ∨
    set (*butlast* (*lower-hull ps*)) ∩ *set* (*butlast* (*upper-hull ps*)) = {}
proof (*cases card* (*set ps*) < *2*)
  case *True*
  have *distinct* (*andrew-hull ps*)
  proof (*cases sorted-unique ps*)
    case *Nil*
    then have *ps* = []
      by *simp*
    then show *?thesis*
      by (*simp add*: *andrew-hull-def sorted-unique-def*)
  next
    case (*Cons p qs*)
    then show *?thesis*

```
    proof (cases qs)
      case Nil
      then show ?thesis
        using Cons by (simp add: andrew-hull-def)
    next
      case (Cons q rs)
      have len-su: length (sorted-unique ps) = card (set ps)
        by (simp add: sorted-unique-def length-sorted-list-of-set)
      have 2 ≤ card (set ps)
        using len-su ‹sorted-unique ps = p # qs› Cons by simp
      then show ?thesis
        using True by simp
    qed
  qed
  then show ?thesis
    using True by simp
next
  case False
  then have ge2: 2 ≤ card (set ps)
    by simp
  have dL: distinct (butlast (lower-hull ps))
    by (rule distinct-butlast[OF distinct-lower-hull])
  have dU: distinct (butlast (upper-hull ps))
    by (rule distinct-butlast[OF distinct-upper-hull])
  have distinct (andrew-hull ps) ⟷
      set (butlast (lower-hull ps)) ∩ set (butlast (upper-hull ps)) = {}
    using andrew-hull-ge2[OF ge2] dL dU
    by (simp add: distinct-append)
  then show ?thesis
    using False by simp
qed

theorem distinct-andrew-hull-if-trimmed-chains-disjoint:
  fixes ps :: real point list
  assumes set (butlast (lower-hull ps)) ∩ set (butlast (upper-hull ps)) = {}
  shows distinct (andrew-hull ps)
  using assms distinct-andrew-hull-iff[of ps] by auto
```

1.4 Top-Level Correctness Statement

The final theorem collects the specification in one place. For real-coordinate inputs, the returned list consists only of input points, covers every input point by its convex hull, has exactly the same convex hull as the input, and is irredundant: deleting any returned point changes the convex hull of the returned set. The preceding sortedness and left-turn theorems describe the order and shape of the lower and upper scans; they are supporting invariants, not a substitute for this envelope and irredundancy specification.

The irredundancy conjunct is the semantic minimality property for the

returned vertex set. It does not merely follow from convex-hull equality; it is proved separately by exposing every returned point with a strict supporting direction.

theorem *andrew-hull-correctness*:
  fixes *ps* :: *real point list*
  shows *set* (*andrew-hull ps*) ⊆ *set ps*
    and *set ps* ⊆ *convex hull set* (*andrew-hull ps*)
    and *convex hull set* (*andrew-hull ps*) = *convex hull set ps*
    and *convex-hull-correct ps* (*andrew-hull ps*)
    and *convex-hull-irredundant* (*andrew-hull ps*)
proof −
  show *set* (*andrew-hull ps*) ⊆ *set ps*
    by (*rule andrew-hull-subset*)
  show *covers*: *set ps* ⊆ *convex hull set* (*andrew-hull ps*)
    by (*rule andrew-hull-covers-input*)
  show *convex hull set* (*andrew-hull ps*) = *convex hull set ps*
    using *covers* **by** (*rule andrew-hull-convex-hull-eqI*)
  show *convex-hull-correct ps* (*andrew-hull ps*)
    by (*rule andrew-hull-correct*)
  show *convex-hull-irredundant* (*andrew-hull ps*)
    by (*rule andrew-hull-irredundant*)
qed

end

2 Examples

theory *Andrew-Monotone-Chain-Examples*
  imports
    Andrew-Monotone-Chain
    HOL−Analysis.Convex-Euclidean-Space
begin

The first examples use integer coordinates to exercise the polymorphic executable code path: the scan, sorting, and orientation tests work over any ordered integral domain. The geometric convex-hull specification is then instantiated in the following subsection over real coordinates, where Isabelle's convexity library provides the ambient vector-space structure.

abbreviation *square-points* :: (*int* × *int*) *list*
where
  square-points ≡
    [(*0*, *0*), (*1*, *0*), (*1*, *1*), (*0*, *1*), (*0*, *0*), (*1*, *1*)]

lemma *square-hull*:
  andrew-hull square-points = [(*0*, *0*), (*1*, *0*), (*1*, *1*), (*0*, *1*)]
  by *eval*

lemma *square-lower-hull*:

lower-hull square-points = [(*0*, *0*), (*1*, *0*), (*1*, *1*)]
by *eval*

lemma *square-upper-hull*:
upper-hull square-points = [(*1*, *1*), (*0*, *1*), (*0*, *0*)]
by *eval*

abbreviation *collinear-points* :: (*int* × *int*) *list*
where
collinear-points ≡ [(*2*, *0*), (*0*, *0*), (*1*, *0*), (*3*, *0*), (*1*, *0*)]

lemma *collinear-hull*:
andrew-hull collinear-points = [(*0*, *0*), (*3*, *0*)]
by *eval*

abbreviation *diamond-points* :: (*int* × *int*) *list*
where
diamond-points ≡
[(*0*, *0*), (*1*, *1*), (*2*, *0*), (*1*, −*1*), (*1*, *0*), (*0*, *0*), (*2*, *0*)]

lemma *diamond-hull*:
andrew-hull diamond-points = [(*0*, *0*), (*1*, −*1*), (*2*, *0*), (*1*, *1*)]
by *eval*

lemma *diamond-hull-subset*:
set (*andrew-hull diamond-points*) ⊆ *set diamond-points*
by (*rule andrew-hull-subset*)

lemma *diamond-hull-turns-lower*:
chain-turns (*lower-hull diamond-points*)
by (*rule lower-hull-turns*)

lemma *diamond-hull-turns-upper*:
chain-turns (*upper-hull diamond-points*)
by (*rule upper-hull-turns*)

2.1 Convex-hull Examples over the Reals

definition *square-real* :: (*real* × *real*) *list*
where
square-real =
[(*0*, *0*), (*1*, *0*), (*1*, *1*), (*0*, *1*), (*0*, *0*), (*1*, *1*)]

definition *square-real-vertices* :: (*real* × *real*) *list*
where
square-real-vertices = [(*0*, *0*), (*1*, *0*), (*1*, *1*), (*0*, *1*)]

lemma *square-real-hull*:
andrew-hull square-real = *square-real-vertices*

by *eval*

lemma *square-real-correct*:
convex-hull-correct square-real (*andrew-hull square-real*)
by (*rule andrew-hull-correctness*(*4*))

lemma *square-real-irredundant*:
convex-hull-irredundant (*andrew-hull square-real*)
by (*rule andrew-hull-correctness*(*5*))

lemma *square-real-convex-hull*:
convex hull set (*andrew-hull square-real*) = *convex hull set square-real*
by (*rule andrew-hull-correctness*(*3*))

definition *collinear-real* :: (*real* × *real*) *list*
where
collinear-real = [(*2*, *0*), (*0*, *0*), (*1*, *0*), (*3*, *0*), (*1*, *0*)]

definition *collinear-real-vertices* :: (*real* × *real*) *list*
where
collinear-real-vertices = [(*0*, *0*), (*3*, *0*)]

lemma *collinear-real-hull*:
andrew-hull collinear-real = *collinear-real-vertices*
by *eval*

lemma *one-zero-in-collinear-real-hull*:
(*1*::*real*, *0*) ∈ *convex hull set collinear-real-vertices*
proof −
let *?a* = (*0*::*real*, *0*::*real*)
let *?b* = (*3*::*real*, *0*::*real*)
have (*1*::*real*, *0*::*real*) ∈ *closed-segment ?a ?b*
by (*simp add*: *in-segment scaleR-prod-def*, *rule exI*[**where** *x* = *1* / *3*], *simp*)
also have ... = *convex hull* {*?a*, *?b*}
by (*simp add*: *segment-convex-hull*)
also have ... = *convex hull set collinear-real-vertices*
by (*simp add*: *collinear-real-vertices-def*)
finally show *?thesis* **.**
qed

lemma *two-zero-in-collinear-real-hull*:
(*2*::*real*, *0*) ∈ *convex hull set collinear-real-vertices*
proof −
let *?a* = (*0*::*real*, *0*::*real*)
let *?b* = (*3*::*real*, *0*::*real*)
have (*2*::*real*, *0*::*real*) ∈ *closed-segment ?a ?b*
by (*simp add*: *in-segment scaleR-prod-def*, *rule exI*[**where** *x* = *2* / *3*], *simp*)
also have ... = *convex hull* {*?a*, *?b*}
by (*simp add*: *segment-convex-hull*)

```
  also have ... = convex hull set collinear-real-vertices
    by (simp add: collinear-real-vertices-def)
  finally show ?thesis .
qed

lemma collinear-real-correct:
  convex-hull-correct collinear-real (andrew-hull collinear-real)
  by (rule andrew-hull-correctness(4))

lemma collinear-real-irredundant:
  convex-hull-irredundant (andrew-hull collinear-real)
  by (rule andrew-hull-correctness(5))

lemma collinear-real-convex-hull:
  convex hull set (andrew-hull collinear-real) = convex hull set collinear-real
  by (rule andrew-hull-correctness(3))

definition diamond-real :: (real × real) list
where
  diamond-real =
    [(0, 0), (1, 1), (2, 0), (1, −1), (1, 0), (0, 0), (2, 0)]

definition diamond-real-vertices :: (real × real) list
where
  diamond-real-vertices = [(0, 0), (1, −1), (2, 0), (1, 1)]

lemma diamond-real-hull:
  andrew-hull diamond-real = diamond-real-vertices
  by eval

lemma diamond-center-in-computed-hull:
  (1::real, 0) ∈ convex hull set diamond-real-vertices
proof −
  let ?a = (0::real, 0::real)
  let ?c = (2::real, 0::real)
  have (1::real, 0::real) ∈ closed-segment ?a ?c
    by (simp add: in-segment scaleR-prod-def, rule exI[where x = 1 / 2], simp)
  also have ... = convex hull {?a, ?c}
    by (simp add: segment-convex-hull)
  also have ... ⊆ convex hull set diamond-real-vertices
    by (rule hull-mono) (auto simp: diamond-real-vertices-def)
  finally show ?thesis .
qed

lemma diamond-real-correct:
  convex-hull-correct diamond-real (andrew-hull diamond-real)
  by (rule andrew-hull-correctness(4))

lemma diamond-real-irredundant:
```

convex-hull-irredundant (*andrew-hull diamond-real*)
by (*rule andrew-hull-correctness*(*5*))

lemma *diamond-real-convex-hull*:
convex hull set (*andrew-hull diamond-real*) = *convex hull set diamond-real*
by (*rule andrew-hull-correctness*(*3*))

value *andrew-hull square-points*
value *andrew-hull collinear-points*
value *andrew-hull diamond-points*

end

References

[1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979. DOI: 10.1016/0020-0190(79)90072-3.