

# Affine Arithmetic

Fabian Immler

March 17, 2025

## Abstract

We give a formalization of affine forms [1, 2] as abstract representations of zonotopes. We provide affine operations as well as overapproximations of some non-affine operations like multiplication and division. Expressions involving those operations can automatically be turned into (executable) functions approximating the original expression in affine arithmetic.

Moreover we give a verified implementation of a functional algorithm to compute the intersection of a zonotope with a hyperplane, as described in the paper [3].

## Contents

0.1	<i>sum-list</i>	5
0.2	Radiant and Degree	5
<b>1</b>	<b>Euclidean Space: Executability</b>	<b>6</b>
1.1	Ordered representation of Basis and Rounding of Components	7
1.2	Instantiations	7
1.3	Representation as list	10
1.4	Bounded Linear Functions	22
1.5	bounded linear functions	22
<b>2</b>	<b>Affine Form</b>	<b>30</b>
2.1	Auxiliary developments	30
2.2	Partial Deviations	32
2.3	Affine Forms	33
2.4	Evaluation, Range, Joint Range	33
2.5	Domain	36
2.6	Least Fresh Index	36
2.7	Total Deviation	38
2.8	Binary Pointwise Operations	38
2.9	Addition	38
2.10	Total Deviation	39

2.11	Unary Operations	39
2.12	Pointwise Scaling of Partial Deviations	40
2.13	Partial Deviations Scale Pointwise	40
2.14	Pointwise Unary Minus	41
2.15	Constant	41
2.16	Inner Product	42
2.17	Inner Product Pair	42
2.18	Update	42
2.19	Inf/Sup	43
2.20	Minkowski Sum	44
2.21	Splitting	50
2.22	From List of Generators	52
2.22.1	(reverse) ordered coefficients as list	62
2.23	2d zonotopes	67
2.24	Intervals	68
<b>3</b>	<b>Operations on Expressions</b>	<b>73</b>
3.1	Approximating Expression*s*	73
3.2	Syntax	73
3.3	Derived symbols	74
3.4	Constant Folding	75
3.5	Free Variables	77
3.6	Derivatives	82
3.7	Definition of Approximating Function using Affine Arithmetic	94
<b>4</b>	<b>Straight Line Programs</b>	<b>123</b>
4.1	Definition	123
4.2	Reification as straight line program (with common subexpression elimination)	123
4.3	better code equations for construction of large programs	140
<b>5</b>	<b>Approximation with Affine Forms</b>	<b>145</b>
5.1	Approximate Operations	148
5.1.1	set of generated endpoints	148
5.1.2	Approximate total deviation	149
5.1.3	truncate partial deviations	150
5.1.4	truncation with error bound	152
5.1.5	general affine operation	154
5.1.6	Inf/Sup	157
5.2	Min Range approximation	158
5.2.1	Addition	160
5.2.2	Scaling	161
5.2.3	Multiplication	162
5.2.4	Inverse	166

5.3	Reduction (Summarization of Coefficients) . . . . .	174
5.4	Splitting with heuristics . . . . .	182
5.5	Approximate Min Range - Kind Of Trigonometric Functions .	185
5.6	Power, TODO: compare with Min-range approximation?! . .	200
5.7	Generic operations on Affine Forms in Euclidean Space . . . .	224
<b>6</b>	<b>Counterclockwise</b>	<b>228</b>
6.1	Auxiliary Lemmas . . . . .	228
6.2	Sort Elements of a List . . . . .	229
6.3	Abstract CCW Systems . . . . .	233
<b>7</b>	<b>CCW Vector Space</b>	<b>235</b>
<b>8</b>	<b>CCW for Nonaligned Points in the Plane</b>	<b>239</b>
8.1	Determinant . . . . .	240
8.2	Strict CCW Predicate . . . . .	245
8.3	Collinearity . . . . .	246
8.4	Polygonal chains . . . . .	250
8.5	Dirvec: Inverse of Polychain . . . . .	254
8.6	Polychain of Sorted ( <i>polychain-of, ccw'.sortedP</i> ) . . . . .	254
<b>9</b>	<b>CCW for Arbitrary Points in the Plane</b>	<b>261</b>
9.1	Interpretation of Knuth's axioms in the plane . . . . .	261
9.2	Order prover setup . . . . .	266
9.3	Contradictions . . . . .	266
<b>10</b>	<b>Intersection</b>	<b>280</b>
10.1	Polygons and <i>ccw, Counterclockwise-2D-Arbitrary.lex, psi, coll</i>	280
10.2	Orient all entries . . . . .	284
10.3	Lowest Vertex . . . . .	285
10.4	Collinear Generators . . . . .	287
10.5	Independent Generators . . . . .	290
10.6	Independent Oriented Generators . . . . .	298
10.7	Half Segments . . . . .	301
10.8	Mirror . . . . .	310
10.9	Full Segments . . . . .	312
10.10	Continuous Generalization . . . . .	320
10.11	Intersection of Vertical Line with Segment . . . . .	321
10.12	Bounds on Vertical Intersection with Oriented List of Segments	324
10.13	Bounds on Vertical Intersection with General List of Segments	328
10.14	Approximation from Orthogonal Directions . . . . .	335
10.15	"Completeness" of Intersection . . . . .	337

<b>11 Implementation</b>	<b>340</b>
11.1 Reverse Sorted, Distinct Association Lists . . . . .	340
11.2 Degree . . . . .	341
11.3 Auxiliary Definitions . . . . .	342
11.4 Pointwise Addition . . . . .	343
11.5 prod of pdevs . . . . .	344
11.6 Set of Coefficients . . . . .	344
11.7 Domain . . . . .	344
11.8 Application . . . . .	345
11.9 Total Deviation . . . . .	345
11.10 Minkowski Sum . . . . .	346
11.11 Unary Operations . . . . .	348
11.12 Filter . . . . .	348
11.13 Constant . . . . .	349
11.14 Update . . . . .	350
11.15 Approximate Total Deviation . . . . .	351
11.16 Equality . . . . .	351
11.17 From List of Generators . . . . .	351
<b>12 Optimizations for Code Integer</b>	<b>352</b>
<b>13 Optimizations for Code Float</b>	<b>353</b>
<b>14 Target Language debug messages</b>	<b>355</b>
14.1 Printing . . . . .	355
14.2 Write to File . . . . .	355
14.3 Show for Floats . . . . .	355
14.4 Convert Float to Decimal number . . . . .	356
14.4.1 Version that should be easy to prove correct, but slow!	356
14.5 Trusted, but faster version . . . . .	357
14.6 gnuplot output . . . . .	359
14.6.1 vector output of 2D zonotope . . . . .	359
<b>15 Dyadic Rational Representation of Real</b>	<b>360</b>
<b>16 Examples</b>	<b>362</b>
<b>17 Examples on Proving Inequalities</b>	<b>363</b>
<b>18 Examples: Intersection of Zonotopes with Hyperplanes</b>	<b>366</b>
18.1 Example . . . . .	366
<b>theory</b> <i>Affine-Arithmetic-Auxiliarities</i>	
<b>imports</b> <i>HOL-Analysis.Multivariate-Analysis</i>	
<b>begin</b>	

## 0.1 *sum-list*

**lemma** *sum-list-nth-eqI*:

**fixes** *xs ys::'a::monoid-add list*

**shows**

$length\ xs = length\ ys \implies (\bigwedge x\ y. (x, y) \in set\ (zip\ xs\ ys) \implies x = y) \implies$   
 $sum-list\ xs = sum-list\ ys$

**by** (*induct xs ys rule: list-induct2*) *auto*

**lemma** *fst-sum-list*:  $fst\ (sum-list\ xs) = sum-list\ (map\ fst\ xs)$

**by** (*induct xs*) *auto*

**lemma** *snd-sum-list*:  $snd\ (sum-list\ xs) = sum-list\ (map\ snd\ xs)$

**by** (*induct xs*) *auto*

**lemma** *take-greater-eqI*:  $take\ c\ xs = take\ c\ ys \implies c \geq a \implies take\ a\ xs = take\ a\ ys$

**proof** (*induct xs arbitrary: a c ys*)

**case** (*Cons x xs*) **note** *ICons = Cons*

**thus** *?case*

**proof** (*cases a*)

**case** (*Suc b*)

**thus** *?thesis using Cons(2,3)*

**proof** (*cases ys*)

**case** (*Cons z zs*)

**from** *ICons* **obtain** *d* **where** *c: c = Suc d*

**by** (*auto simp: Cons Suc dest!: Suc-le-D*)

**show** *?thesis*

**using** *ICons(2,3)*

**by** (*auto simp: Suc Cons c intro: ICons(1)*)

**qed** *simp*

**qed** *simp*

**qed** (*metis le-0-eq take-eq-Nil*)

**lemma** *take-max-eqD*:

$take\ (max\ a\ b)\ xs = take\ (max\ a\ b)\ ys \implies take\ a\ xs = take\ a\ ys \wedge take\ b\ xs = take\ b\ ys$

**by** (*metis max.cobounded1 max.cobounded2 take-greater-eqI*)

**lemma** *take-Suc-eq*:  $take\ (Suc\ n)\ xs = (if\ n < length\ xs\ then\ take\ n\ xs\ @\ [xs\ !\ n]\ else\ xs)$

**by** (*auto simp: take-Suc-conv-app-nth*)

## 0.2 *Radiant and Degree*

**definition** *rad-of*  $w = w * pi / 180$

**definition** *deg-of*  $w = 180 * w / pi$

**lemma** *rad-of-inverse[simp]*:  $deg-of\ (rad-of\ w) = w$

```

and deg-of-inverse[simp]: rad-of (deg-of w) = w
by (auto simp: deg-of-def rad-of-def)

lemma deg-of-monoI:  $x \leq y \implies \text{deg-of } x \leq \text{deg-of } y$ 
by (auto simp: deg-of-def intro!: divide-right-mono)

lemma rad-of-monoI:  $x \leq y \implies \text{rad-of } x \leq \text{rad-of } y$ 
by (auto simp: rad-of-def)

lemma deg-of-strict-monoI:  $x < y \implies \text{deg-of } x < \text{deg-of } y$ 
by (auto simp: deg-of-def intro!: divide-strict-right-mono)

lemma rad-of-strict-monoI:  $x < y \implies \text{rad-of } x < \text{rad-of } y$ 
by (auto simp: rad-of-def)

lemma deg-of-mono[simp]:  $\text{deg-of } x \leq \text{deg-of } y \iff x \leq y$ 
using rad-of-monoI
by (fastforce intro!: deg-of-monoI)

lemma rad-of-mono[simp]:  $\text{rad-of } x \leq \text{rad-of } y \iff x \leq y$ 
using rad-of-monoI
by (fastforce intro!: deg-of-monoI)

lemma deg-of-strict-mono[simp]:  $\text{deg-of } x < \text{deg-of } y \iff x < y$ 
using rad-of-strict-monoI
by (fastforce intro!: deg-of-strict-monoI)

lemma rad-of-strict-mono[simp]:  $\text{rad-of } x < \text{rad-of } y \iff x < y$ 
using rad-of-strict-monoI
by (fastforce intro!: deg-of-strict-monoI)

lemma rad-of-lt-iff:  $\text{rad-of } d < r \iff d < \text{deg-of } r$ 
and rad-of-gt-iff:  $\text{rad-of } d > r \iff d > \text{deg-of } r$ 
and rad-of-le-iff:  $\text{rad-of } d \leq r \iff d \leq \text{deg-of } r$ 
and rad-of-ge-iff:  $\text{rad-of } d \geq r \iff d \geq \text{deg-of } r$ 
using rad-of-strict-mono[of d deg-of r] rad-of-mono[of d deg-of r]
by auto

end

```

## 1 Euclidean Space: Executability

```

theory Executable-Euclidean-Space
imports
  HOL-Analysis.Multivariate-Analysis
  List-Index.List-Index
  HOL-Library.Float
  HOL-Library.Code-Cardinality
  Affine-Arithmetic-Auxiliarities

```

begin

## 1.1 Ordered representation of Basis and Rounding of Components

**class** *executable-euclidean-space* = *ordered-euclidean-space* +  
**fixes** *Basis-list* *eucl-down* *eucl-truncate-down* *eucl-truncate-up*  
**assumes** *eucl-down-def*:  
     $eucl-down\ p\ b = (\sum i \in Basis. round-down\ p\ (b \cdot i) *_R i)$   
**assumes** *eucl-truncate-down-def*:  
     $eucl-truncate-down\ q\ b = (\sum i \in Basis. truncate-down\ q\ (b \cdot i) *_R i)$   
**assumes** *eucl-truncate-up-def*:  
     $eucl-truncate-up\ q\ b = (\sum i \in Basis. truncate-up\ q\ (b \cdot i) *_R i)$   
**assumes** *Basis-list[simp]*: *set Basis-list* = *Basis*  
**assumes** *distinct-Basis-list[simp]*: *distinct Basis-list*  
begin

**lemma** *length-Basis-list*:  
     $length\ Basis-list = card\ Basis$   
    **by** (*metis Basis-list distinct-Basis-list distinct-card*)

end

**lemma** *eucl-truncate-down-zero[simp]*:  $eucl-truncate-down\ p\ 0 = 0$   
    **by** (*auto simp: eucl-truncate-down-def truncate-down-zero*)

**lemma** *eucl-truncate-up-zero[simp]*:  $eucl-truncate-up\ p\ 0 = 0$   
    **by** (*auto simp: eucl-truncate-up-def*)

## 1.2 Instantiations

**instantiation** *real::executable-euclidean-space*  
begin

**definition** *Basis-list-real* :: *real list* **where**  
     $Basis-list-real = [1]$

**definition** *eucl-down prec b* = *round-down prec b*

**definition** *eucl-truncate-down prec b* = *truncate-down prec b*

**definition** *eucl-truncate-up prec b* = *truncate-up prec b*

**instance proof qed** (*auto simp: Basis-list-real-def eucl-down-real-def eucl-truncate-down-real-def eucl-truncate-up-real-def*)

end

**instantiation** *prod::(executable-euclidean-space, executable-euclidean-space)*  
     $executable-euclidean-space$   
begin

**definition** *Basis-list-prod* :: ('a × 'b) list **where**

*Basis-list-prod* =  
 zip *Basis-list* (replicate (length (*Basis-list*::'a list)) 0) @  
 zip (replicate (length (*Basis-list*::'b list)) 0) *Basis-list*

**definition** *eucl-down* p a = (*eucl-down* p (fst a), *eucl-down* p (snd a))

**definition** *eucl-truncate-down* p a = (*eucl-truncate-down* p (fst a), *eucl-truncate-down* p (snd a))

**definition** *eucl-truncate-up* p a = (*eucl-truncate-up* p (fst a), *eucl-truncate-up* p (snd a))

**instance**

**proof**

**show** set *Basis-list* = (*Basis*::('a\*'b) set)  
**by** (auto simp: *Basis-list-prod-def* *Basis-prod-def* elim!: *in-set-zipE*)  
 (auto simp: *Basis-list[symmetric]* *in-set-zip* *in-set-conv-nth* simp del: *Basis-list*)  
**show** distinct (*Basis-list*::('a\*'b)list)  
**using** *distinct-Basis-list[where 'a='a]* *distinct-Basis-list[where 'a='b]*  
**by** (auto simp: *Basis-list-prod-def* *Basis-list* intro: *distinct-zipI1* *distinct-zipI2*  
 elim!: *in-set-zipE*)

**qed**

(auto simp: *eucl-down-prod-def* *eucl-truncate-down-prod-def* *eucl-truncate-up-prod-def*  
*sum-Basis-prod-eq* *inner-add-left* *inner-sum-left* *inner-Basis* *eucl-down-def*  
*eucl-truncate-down-def* *eucl-truncate-up-def*  
 intro!: *euclidean-eqI[where 'a='a\*'b]*)

**end**

**lemma** *eucl-truncate-down-Basis[simp]*:

$i \in \text{Basis} \implies \text{eucl-truncate-down } e \cdot i = \text{truncate-down } e (x \cdot i)$   
**by** (*simp* add: *eucl-truncate-down-def*)

**lemma** *eucl-truncate-down-correct*:

$\text{dist } (x::'a::\text{executable-euclidean-space}) (\text{eucl-down } e \cdot x) \in$   
 $\{0.. \text{sqrt } (\text{DIM } ('a)) * 2 \text{ powr of-int } (- e)\}$

**proof** –

**have**  $\text{dist } x (\text{eucl-down } e \cdot x) = \text{sqrt } (\sum_{i \in \text{Basis}} (\text{dist } (x \cdot i) (\text{eucl-down } e \cdot x \cdot i))^2)$

**unfolding** *euclidean-dist-l2[where 'a='a]* *L2-set-def* ..

**also have**  $\dots \leq \text{sqrt } (\sum_{i \in (\text{Basis}::'a \text{ set})} ((2 \text{ powr of-int } (- e))^2))$

**by** (*intro* *real-sqrt-le-mono* *sum-mono* *power-mono*)

(auto simp: *dist-real-def* *eucl-down-def* *abs-round-down-le*)

**finally show** *?thesis*

**by** (*simp* add: *real-sqrt-mult*)

**qed**

**lemma** *eucl-down*:  $\text{eucl-down } e (x::'a::\text{executable-euclidean-space}) \leq x$

**by** (auto simp add: *eucl-le[where 'a='a]* *round-down eucl-down-def*)



**lemma** *eucl-truncate-down*: *eucl-truncate-down*  $e$  ( $x::'a::\text{executable-euclidean-space}$ )  
 $\leq x$   
**by** (*auto simp add: eucl-le*[**where**  $'a='a$ ] *truncate-down*)

**lemma** *eucl-truncate-down-le*:  
 $x \leq y \implies \text{eucl-truncate-down } w \ x \leq (y::'a::\text{executable-euclidean-space})$   
**using** *eucl-truncate-down*  
**by** (*rule order.trans*)

**lemma** *eucl-truncate-up-Basis*[*simp*]:  $i \in \text{Basis} \implies \text{eucl-truncate-up } e \ x \cdot i =$   
 $\text{truncate-up } e \ (x \cdot i)$   
**by** (*simp add: eucl-truncate-up-def truncate-up-def*)

**lemma** *eucl-truncate-up*:  $x \leq \text{eucl-truncate-up } e \ (x::'a::\text{executable-euclidean-space})$   
**by** (*auto simp add: eucl-le*[**where**  $'a='a$ ] *round-up truncate-up-def*)

**lemma** *eucl-truncate-up-le*:  $x \leq y \implies x \leq \text{eucl-truncate-up } e \ (y::'a::\text{executable-euclidean-space})$   
**using** - *eucl-truncate-up*  
**by** (*rule order.trans*)

**lemma** *eucl-truncate-down-mono*:  
**fixes**  $x::'a::\text{executable-euclidean-space}$   
**shows**  $x \leq y \implies \text{eucl-truncate-down } p \ x \leq \text{eucl-truncate-down } p \ y$   
**by** (*auto simp: eucl-le*[**where**  $'a='a$ ] *intro!*: *truncate-down-mono*)

**lemma** *eucl-truncate-up-mono*:  
**fixes**  $x::'a::\text{executable-euclidean-space}$   
**shows**  $x \leq y \implies \text{eucl-truncate-up } p \ x \leq \text{eucl-truncate-up } p \ y$   
**by** (*auto simp: eucl-le*[**where**  $'a='a$ ] *intro!*: *truncate-up-mono*)

**lemma** *infnorm*[*code*]:  
**fixes**  $x::'a::\text{executable-euclidean-space}$   
**shows**  $\text{infnorm } x = \text{fold } \max \ (\text{map } (\lambda i. \text{abs } (x \cdot i)) \ \text{Basis-list}) \ 0$   
**by** (*auto simp: Max.set-eq-fold*[*symmetric*] *infnorm-Max*[*symmetric*] *infnorm-pos-le*  
*intro!*: *max.absorb2*[*symmetric*])

**declare** *Inf-real-def*[*code del*]  
**declare** *Sup-real-def*[*code del*]  
**declare** *Inf-prod-def*[*code del*]  
**declare** *Sup-prod-def*[*code del*]  
**declare** [[*code abort: Inf::real set  $\Rightarrow$  real*]]  
**declare** [[*code abort: Sup::real set  $\Rightarrow$  real*]]  
**declare** [[*code abort: Inf::('a::Inf \* 'b::Inf) set  $\Rightarrow$  'a \* 'b*]]  
**declare** [[*code abort: Sup::('a::Sup \* 'b::Sup) set  $\Rightarrow$  'a \* 'b*]]

**lemma** *nth-Basis-list-in-Basis*[*simp*]:  
 $n < \text{length } (\text{Basis-list}::'a::\text{executable-euclidean-space list}) \implies \text{Basis-list } ! \ n \in$   
 $(\text{Basis}::'a \ \text{set})$

by (metis Basis-list nth-mem)

### 1.3 Representation as list

**lemma** *nth-eq-iff-index*:

$distinct\ xs \implies n < length\ xs \implies xs\ !\ n = i \longleftrightarrow n = index\ xs\ i$   
using *index-nth-id* by *fastforce*

**lemma** *in-Basis-index-Basis-list*:  $i \in Basis \implies i = Basis-list\ !\ index\ Basis-list\ i$   
by *simp*

**lemmas** [*simp*] = *length-Basis-list*

**lemma** *sum-Basis-sum-nth-Basis-list*:

$(\sum_{i \in Basis} f\ i) = (\sum_{i < DIM('a::executable-euclidean-space)} f\ ((Basis-list::'a\ list)\ !\ i))$

**apply** (*rule sum.reindex-cong[OF - - refl]*)

**apply** (*auto intro!: inj-on-nth*)

by (*metis Basis-list image-iff in-Basis-index-Basis-list index-less-size-conv length-Basis-list lessThan-iff*)

**definition** *eucl-of-list*  $xs = (\sum (x, i) \leftarrow zip\ xs\ Basis-list. x *_{\mathbb{R}} i)$

**lemma** *eucl-of-list-nth*:

**assumes**  $length\ xs = DIM('a)$

**shows**  $eucl-of-list\ xs = (\sum_{i < DIM('a::executable-euclidean-space)} (xs\ !\ i) *_{\mathbb{R}} ((Basis-list::'a\ list)\ !\ i))$

by (*auto simp: eucl-of-list-def sum-list-sum-nth length-Basis-list assms atLeast0LessThan*)

**lemma** *eucl-of-list-inner*:

**fixes**  $i::'a::executable-euclidean-space$

**assumes**  $i \in Basis$

**assumes**  $l: length\ xs = DIM('a)$

**shows**  $eucl-of-list\ xs \cdot i = xs\ !\ (index\ Basis-list\ i)$

by (*simp add: eucl-of-list-nth[OF l] inner-sum-left assms inner-Basis nth-eq-iff-index sum.delta if-distrib cong: if-cong*)

**lemma** *inner-eucl-of-list*:

**fixes**  $i::'a::executable-euclidean-space$

**assumes**  $i \in Basis$

**assumes**  $l: length\ xs = DIM('a)$

**shows**  $i \cdot eucl-of-list\ xs = xs\ !\ (index\ Basis-list\ i)$

using *eucl-of-list-inner[OF assms]* by (*auto simp: inner-commute*)

**definition** *list-of-eucl*  $x = map\ ((\cdot)\ x)\ Basis-list$

**lemma** *index-Basis-list-nth[*simp*]*:

$i < DIM('a::executable-euclidean-space) \implies index\ Basis-list\ ((Basis-list::'a\ list)$

! i) = i  
 by (simp add: index-nth-id)

**lemma** list-of-eucl-eucl-of-list[simp]:  
 length xs = DIM('a::executable-euclidean-space)  $\implies$  list-of-eucl (eucl-of-list xs::'a)  
 = xs  
 by (auto simp: list-of-eucl-def eucl-of-list-inner intro!: nth-equalityI)

**lemma** eucl-of-list-list-of-eucl[simp]:  
 eucl-of-list (list-of-eucl x) = x  
 by (auto simp: list-of-eucl-def eucl-of-list-inner intro!: euclidean-eqI[where 'a='a])

**lemma** length-list-of-eucl[simp]: length (list-of-eucl (x::'a::executable-euclidean-space))  
 = DIM('a)  
 by (auto simp: list-of-eucl-def)

**lemma** list-of-eucl-nth[simp]:  $n < DIM('a::executable-euclidean-space) \implies$  list-of-eucl  
 $x ! n = x \cdot (Basis-list ! n::'a)$   
 by (auto simp: list-of-eucl-def)

**lemma** nth-ge-len:  $n \geq \text{length } xs \implies xs ! n = [] ! (n - \text{length } xs)$   
 by (induction xs arbitrary: n) auto

**lemma** list-of-eucl-nth-if: list-of-eucl  $x ! n =$  (if  $n < DIM('a::executable-euclidean-space)$   
 then  $x \cdot (Basis-list ! n::'a)$  else  $[] ! (n - DIM('a))$ )  
 apply (auto simp: list-of-eucl-def )  
 apply (subst nth-ge-len)  
 apply auto  
 done

**lemma** list-of-eucl-eq-iff:  
 list-of-eucl (x::'a::executable-euclidean-space) = list-of-eucl (y::'b::executable-euclidean-space)  
 $\longleftrightarrow$   
 (DIM('a) = DIM('b)  $\wedge$  ( $\forall i < DIM('b). x \cdot Basis-list ! i = y \cdot Basis-list ! i$ ))  
 by (auto simp: list-eq-iff-nth-eq)

**lemma** eucl-le-Basis-list-iff:  
 (x::'a::executable-euclidean-space)  $\leq y \longleftrightarrow$   
 ( $\forall i < DIM('a). x \cdot Basis-list ! i \leq y \cdot Basis-list ! i$ )  
 apply (auto simp: eucl-le[where 'a='a])  
 subgoal for i  
 subgoal by (auto dest!: spec[where x=index Basis-list i])  
 done  
 done

**lemma** eucl-of-list-inj: length xs = DIM('a::executable-euclidean-space)  $\implies$  length  
 ys = DIM('a)  $\implies$   
 (eucl-of-list xs::'a) = eucl-of-list (ys)  $\implies xs = ys$

**apply** (*auto intro!*: *nth-equalityI simp: euclidean-eq-iff* [**where** 'a='a] *eucl-of-list-inner*)  
**using** *nth-Basis-list-in-Basis* [**where** 'a='a]  
**by** *fastforce*

**lemma** *eucl-of-list-map-plus*[*simp*]:  
**assumes** [*simp*]: *length xs = DIM('a::executable-euclidean-space)*  
**shows** (*eucl-of-list (map (λx. f x + g x) xs)::'a*) =  
*eucl-of-list (map f xs) + eucl-of-list (map g xs)*  
**by** (*auto intro!*: *euclidean-eqI* [**where** 'a='a] *simp: algebra-simps eucl-of-list-inner*)

**lemma** *eucl-of-list-map-uminus*[*simp*]:  
**assumes** [*simp*]: *length xs = DIM('a::executable-euclidean-space)*  
**shows** (*eucl-of-list (map (λx. - f x) xs)::'a*) = - *eucl-of-list (map f xs)*  
**by** (*auto intro!*: *euclidean-eqI* [**where** 'a='a] *simp: algebra-simps eucl-of-list-inner*)

**lemma** *eucl-of-list-map-mult-left*[*simp*]:  
**assumes** [*simp*]: *length xs = DIM('a::executable-euclidean-space)*  
**shows** (*eucl-of-list (map (λx. r \* f x) xs)::'a*) = *r \*<sub>R</sub> eucl-of-list (map f xs)*  
**by** (*auto intro!*: *euclidean-eqI* [**where** 'a='a] *simp: algebra-simps eucl-of-list-inner*)

**lemma** *eucl-of-list-map-mult-right*[*simp*]:  
**assumes** [*simp*]: *length xs = DIM('a::executable-euclidean-space)*  
**shows** (*eucl-of-list (map (λx. f x \* r) xs)::'a*) = *r \*<sub>R</sub> eucl-of-list (map f xs)*  
**by** (*auto intro!*: *euclidean-eqI* [**where** 'a='a] *simp: algebra-simps eucl-of-list-inner*)

**lemma** *eucl-of-list-map-divide-right*[*simp*]:  
**assumes** [*simp*]: *length xs = DIM('a::executable-euclidean-space)*  
**shows** (*eucl-of-list (map (λx. f x / r) xs)::'a*) = *eucl-of-list (map f xs) /<sub>R</sub> r*  
**by** (*auto intro!*: *euclidean-eqI* [**where** 'a='a] *simp: algebra-simps eucl-of-list-inner divide-simps*)

**lemma** *eucl-of-list-map-const*[*simp*]:  
**assumes** [*simp*]: *length xs = DIM('a::executable-euclidean-space)*  
**shows** (*eucl-of-list (map (λx. c) xs)::'a*) = *c \*<sub>R</sub> One*  
**by** (*auto intro!*: *euclidean-eqI* [**where** 'a='a] *simp: algebra-simps eucl-of-list-inner*)

**lemma** *replicate-eq-list-of-eucl-zero*: *replicate DIM('a::executable-euclidean-space)*  
*0 = list-of-eucl (0::'a)*  
**by** (*auto intro!*: *nth-equalityI*)

**lemma** *eucl-of-list-append-zeroes*[*simp*]: *eucl-of-list (xs @ replicate n 0) = eucl-of-list*  
*xs*  
**unfolding** *eucl-of-list-def*  
**apply** (*auto simp: sum-list-sum-nth*)  
**apply** (*rule sum.mono-neutral-cong-right*)  
**by** (*auto simp: nth-append*)

**lemma** *Basis-prodD*:  
**assumes** (*i, j*) ∈ *Basis*

**shows**  $i \in \text{Basis} \wedge j = 0 \vee i = 0 \wedge j \in \text{Basis}$   
**using** *assms*  
**by** (*auto simp: Basis-prod-def*)

**lemma** *eucl-of-list-take-DIM*[*simp*]:  
**assumes**  $d = \text{DIM}('b::\text{executable-euclidean-space})$   
**shows**  $(\text{eucl-of-list } (\text{take } d \text{ } xs)::'b) = (\text{eucl-of-list } xs)$   
**by** (*auto simp: eucl-of-list-inner eucl-of-list-def fst-sum-list sum-list-sum-nth assms dest!: Basis-prodD*)

**lemma** *eucl-of-list-eqI*:  
**assumes**  $\text{take } \text{DIM}('a) (xs @ \text{replicate } (\text{DIM}('a) - \text{length } xs) 0) =$   
 $\text{take } \text{DIM}('a) (ys @ \text{replicate } (\text{DIM}('a) - \text{length } ys) 0)$   
**shows**  $\text{eucl-of-list } xs = (\text{eucl-of-list } ys::'a::\text{executable-euclidean-space})$   
**proof** –  
**have**  $(\text{eucl-of-list } xs::'a) = \text{eucl-of-list } (\text{take } \text{DIM}('a) (xs @ \text{replicate } (\text{DIM}('a)$   
 $- \text{length } xs) 0))$   
**by** *simp*  
**also note** *assms*  
**also have**  $\text{eucl-of-list } (\text{take } \text{DIM}('a) (ys @ \text{replicate } (\text{DIM}('a) - \text{length } ys) 0))$   
 $= (\text{eucl-of-list } ys::'a)$   
**by** *simp*  
**finally show** *?thesis* .  
**qed**

**lemma** *eucl-of-list-replicate-zero*[*simp*]:  $\text{eucl-of-list } (\text{replicate } E 0) = 0$   
**proof** –  
**have**  $\text{eucl-of-list } (\text{replicate } E 0) = (\text{eucl-of-list } (\text{replicate } E 0 @ \text{replicate } (\text{DIM}('a)$   
 $- E) 0)::'a)$   
**by** *simp*  
**also have**  $\dots = \text{eucl-of-list } (\text{replicate } \text{DIM}('a) 0)$   
**apply** (*rule eucl-of-list-eqI*)  
**by** (*auto simp: min-def nth-append intro!: nth-equalityI*)  
**also have**  $\dots = 0$   
**by** (*simp add: replicate-eq-list-of-eucl-zero*)  
**finally show** *?thesis* **by** *simp*  
**qed**

**lemma** *eucl-of-list-Nil*[*simp*]:  $\text{eucl-of-list } [] = 0$   
**using** *eucl-of-list-replicate-zero*[*of 0*] **by** *simp*

**lemma** *fst-eucl-of-list-prod*:  
**shows**  $\text{fst } (\text{eucl-of-list } xs::'b::\text{executable-euclidean-space} \times -) = (\text{eucl-of-list } (\text{take}$   
 $\text{DIM}('b) \text{ } xs)::'b)$   
**apply** (*auto simp: eucl-of-list-inner eucl-of-list-def fst-sum-list dest!: Basis-prodD*)  
**apply** (*simp add: sum-list-sum-nth*)  
**apply** (*rule sum.mono-neutral-cong-right*)  
**subgoal by** *simp*

**subgoal by** *auto*  
**subgoal by** (*auto simp: Basis-list-prod-def nth-append*)  
**subgoal by** (*auto simp: Basis-list-prod-def nth-append*)  
**done**

**lemma** *index-zip-replicate1*[*simp*]:  $\text{index } (\text{zip } (\text{replicate } d \ a) \ bs) \ (a, \ b) = \text{index } bs \ b$

**if**  $d = \text{length } bs$   
**using** *that*  
**by** (*induction bs arbitrary: d*) *auto*

**lemma** *index-zip-replicate2*[*simp*]:  $\text{index } (\text{zip } as \ (\text{replicate } d \ b)) \ (a, \ b) = \text{index } as \ a$

**if**  $d = \text{length } as$   
**using** *that*  
**by** (*induction as arbitrary: d*) *auto*

**lemma** *index-Basis-list-prod*[*simp*]:

**fixes**  $a::'a::\text{executable-euclidean-space}$  **and**  $b::'b::\text{executable-euclidean-space}$   
**shows**  $a \in \text{Basis} \implies \text{index } \text{Basis-list } (a, \ 0::'b) = \text{index } \text{Basis-list } a$   
 $b \in \text{Basis} \implies \text{index } \text{Basis-list } (0::'a, \ b) = \text{DIM}('a) + \text{index } \text{Basis-list } b$   
**by** (*auto simp: Basis-list-prod-def index-append*)  
*in-set-zip zip-replicate index-map-inj dest: spec*[**where**  $x = \text{index } \text{Basis-list } a$ ]

**lemma** *eucl-of-list-eq-takeI*:

**assumes**  $(\text{eucl-of-list } (\text{take } \text{DIM}('a::\text{executable-euclidean-space}) \ xs)::'a) = x$   
**shows**  $\text{eucl-of-list } xs = x$   
**using** *eucl-of-list-take-DIM*[*OF refl, of xs, where 'b='a*] *assms*  
**by** *auto*

**lemma** *eucl-of-list-inner-le*:

**fixes**  $i::'a::\text{executable-euclidean-space}$   
**assumes**  $i \in \text{Basis}$   
**assumes**  $l: \text{length } xs \geq \text{DIM}('a)$   
**shows**  $\text{eucl-of-list } xs \cdot i = xs \ ! \ (\text{index } \text{Basis-list } i)$

**proof** –

**have**  $(\text{eucl-of-list } xs::'a) = \text{eucl-of-list } (\text{take } \text{DIM}('a) \ (xs \ @ \ (\text{replicate } (\text{DIM}('a) - \text{length } xs) \ 0)))$   
**by** (*rule eucl-of-list-eq-takeI simp*)  
**also have**  $\dots \cdot i = xs \ ! \ (\text{index } \text{Basis-list } i)$   
**using** *assms*  
**by** (*subst eucl-of-list-inner*) *auto*  
**finally show** *?thesis* .

**qed**

**lemma** *eucl-of-list-prod-if*:

**assumes**  $\text{length } xs = \text{DIM}('a::\text{executable-euclidean-space}) + \text{DIM}('b::\text{executable-euclidean-space})$   
**shows**  $\text{eucl-of-list } xs =$   
 $(\text{eucl-of-list } (\text{take } \text{DIM}('a) \ xs)::'a, \ \text{eucl-of-list } (\text{drop } \text{DIM}('a) \ xs)::'b)$

```

apply (rule euclidean-eqI)
using assms
apply (auto simp: eucl-of-list-inner dest!: Basis-prodD)
apply (subst eucl-of-list-inner-le)
apply (auto simp: Basis-list-prod-def index-append in-set-zip)
done

lemma snd-eucl-of-list-prod:
shows snd (eucl-of-list xs::'b::executable-euclidean-space × 'c::executable-euclidean-space)
=
  (eucl-of-list (drop DIM('b) xs)::'c)
proof (cases length xs ≤ DIM('b))
case True
then show ?thesis
by (auto simp: eucl-of-list-inner eucl-of-list-def snd-sum-list dest!: Basis-prodD)
  (simp add: sum-list-sum-nth Basis-list-prod-def nth-append)
next
case False
have xs = take DIM('b) xs @ drop DIM('b) xs by simp
also have eucl-of-list ... = (eucl-of-list (... @ replicate (length xs - DIM('c))
0)::'b × 'c)
by simp
finally have eucl-of-list xs = (eucl-of-list (xs @ replicate (DIM('b) + DIM('c)
- length xs) 0)::'b × 'c)
by simp
also have ... = eucl-of-list (take (DIM ('b × 'c)) (xs @ replicate (DIM('b) +
DIM('c) - length xs) 0))
by simp
finally have *: (eucl-of-list xs::'b × 'c) = eucl-of-list (take DIM('b × 'c) (xs @
replicate (DIM('b) + DIM('c) - length xs) 0))
by simp
show ?thesis
apply (subst *)
apply (subst eucl-of-list-prod-if)
subgoal by simp
subgoal
apply simp
apply (subst (2) eucl-of-list-take-DIM[OF refl, symmetric])
apply (subst (2) eucl-of-list-take-DIM[OF refl, symmetric])
apply (rule arg-cong[where f=eucl-of-list])
by (auto intro!: nth-equalityI simp: nth-append min-def split: if-splits)
done
qed

lemma eucl-of-list-prod:
shows eucl-of-list xs = (eucl-of-list (take DIM('b) xs)::'b::executable-euclidean-space,
eucl-of-list (drop DIM('b) xs)::'c::executable-euclidean-space)
using snd-eucl-of-list-prod[of xs, where 'b'=b and 'c'=c]

```

using *fst-eucl-of-list-prod*[*of xs, where 'b='b and 'a='c*]  
 by (*auto simp del: snd-eucl-of-list-prod fst-eucl-of-list-prod simp add: prod-eq-iff*)

**lemma** *eucl-of-list-real*[*simp*]: *eucl-of-list* [x] = (x::real)  
 by (*auto simp: eucl-of-list-def Basis-list-real-def*)

**lemma** *eucl-of-list-append*[*simp*]:  
 assumes *length xs = DIM('i::executable-euclidean-space)*  
 assumes *length ys = DIM('j::executable-euclidean-space)*  
 shows *eucl-of-list (xs @ ys) = (eucl-of-list xs::'i, eucl-of-list ys::'j)*  
 using *assms*  
 by (*auto simp: eucl-of-list-prod*)

**lemma** *list-allI*: ( $\bigwedge x. x \in \text{set } xs \implies P x$ )  $\implies \text{list-all } P \text{ } xs$   
 by (*auto simp: list-all-iff*)

**lemma**  
*concat-map-nthI*:  
 assumes  $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } (f x) \implies P y$   
 assumes  $j < \text{length } (\text{concat } (\text{map } f \text{ } xs))$   
 shows  $P (\text{concat } (\text{map } f \text{ } xs) ! j)$   
**proof** –  
 have *list-all P (concat (map f xs))*  
 by (*rule list-allI (auto simp: assms)*)  
 then show *?thesis*  
 by (*auto simp: list-all-length assms*)  
**qed**

**lemma** *map-nth-append1*:  
 assumes *length xs = d*  
 shows *map (!) (xs @ ys) [0..*d*] = xs*  
 using *assms*  
 by (*auto simp: nth-append intro!: nth-equalityI*)

**lemma** *map-nth-append2*:  
 assumes *length ys = d*  
 shows *map (!) (xs @ ys) [length xs..*length xs + d*] = ys*  
 using *assms*  
 by (*auto simp: intro!: nth-equalityI*)

**lemma** *length-map2* [*simp*]: *length (map2 f xs ys) = min (length xs) (length ys)*  
 by *simp*

**lemma** *map2-nth* [*simp*]: *map2 f xs ys ! n = f (xs ! n) (ys ! n)*  
 if  $n < \text{length } xs$   $n < \text{length } ys$   
 using *that by simp*

**lemma** *list-of-eucl-add*: *list-of-eucl (x + y) = map2 (+) (list-of-eucl x) (list-of-eucl y)*



by (auto intro!: nth-equalityI simp: inner-simps)

**lemma** *list-of-eucl-inj*:

*list-of-eucl z = list-of-eucl y  $\implies$  y = z*

by (metis eucl-of-list-list-of-eucl)

**lemma** *length-Basis-list-pos[simp]*: *length Basis-list > 0*

by (metis length-pos-if-in-set Basis-list SOME-Basis)

**lemma** *Basis-list-nth-nonzero*:

*i < length (Basis-list::'a::executable-euclidean-space list)  $\implies$  (Basis-list::'a list) ! i  $\neq$  0*

by (auto dest!: nth-mem simp: nonzero-Basis)

**lemma** *nth-Basis-list-prod*:

*i < DIM('a) + DIM('b)  $\implies$  (Basis-list::('a::executable-euclidean-space  $\times$  'b::executable-euclidean-space) list) ! i =*

*(if i < DIM('a) then (Basis-list ! i, 0) else (0, Basis-list ! (i - DIM('a))))*

by (auto simp: Basis-list-nth-nonzero prod-eq-iff Basis-list-prod-def nth-append not-less)

**lemma** *eucl-of-list-if*:

**assumes** [simp]: *length xs = DIM('a::executable-euclidean-space) distinct xs*

**shows** *eucl-of-list (map ( $\lambda xa. \text{if } xa = x \text{ then } 1 \text{ else } 0$ ) (xs::nat list)) =*

*(if x  $\in$  set xs then Basis-list ! index xs x else 0::'a)*

by (rule euclidean-eqI) (auto simp: eucl-of-list-inner inner-Basis index-nth-id)

**lemma** *take-append-take-minus-idem*: *take n XS @ map (! XS) [n..*length XS*] = XS*

by (auto intro!: nth-equalityI simp: nth-append min-def)

**lemma** *sum-list-Basis-list[simp]*: *sum-list (map f Basis-list) = ( $\sum b \in \text{Basis}. f b$ )*

by (subst sum-list-distinct-conv-sum-set) (auto simp: Basis-list distinct-Basis-list)

**lemma** *hd-Basis-list[simp]*: *hd Basis-list  $\in$  Basis*

**unfolding** *Basis-list[symmetric]*

by (rule hd-in-set) (auto simp: set-empty[symmetric])

**definition** *inner-lv-rel a b = sum-list (map2 (\*) a b)*

**lemma** *eucl-of-list-inner-eq*: *(eucl-of-list xs::'a)  $\cdot$  eucl-of-list ys = inner-lv-rel xs ys*

**if** *length xs = DIM('a::executable-euclidean-space) length ys = DIM('a)*

**using** *that*

by (subst euclidean-inner[abs-def], subst sum-list-Basis-list[symmetric])

(auto simp: eucl-of-list-inner sum-list-sum-nth index-nth-id inner-lv-rel-def)

**lemma** *euclidean-vec-componentwise*:

```

  (∑ (xa: 'a::euclidean-space ^'b::finite)∈Basis. f xa) = (∑ a∈Basis. (∑ b::'b∈UNIV.
  f (axis b a)))
  apply (auto simp: Basis-vec-def)
  apply (subst sum.swap)
  apply (subst sum.Union-disjoint)
  apply auto
  apply (simp add: axis-eq-axis nonzero-Basis)
  apply (simp add: axis-eq-axis nonzero-Basis)
  apply (subst sum.reindex)
  apply (auto intro!: injI)
  subgoal
  apply (auto simp: set-eq-iff)
  by (metis (full-types) all-not-in-conv inner-axis-axis inner-eq-zero-iff nonempty-Basis
  nonzero-Basis)
  apply (rule sum.cong[OF refl])
  apply (auto)
  apply (rule sum.reindex-cong[OF - - refl])
  apply (auto intro!: inj-onI)
  using axis-eq-axis by blast

```

**lemma** *vec-nth-inner-scaleR-craziness*:

```

  f (x $ i · j) *R j = (∑ xa∈UNIV. f (x $ xa · j) *R axis xa j) $ i
  by vector (auto simp: axis-def if-distrib scaleR-vec-def sum.delta' cong: if-cong)

```

**instantiation** *vec* :: (*{executable-euclidean-space}*, *enum*) *executable-euclidean-space*  
**begin**

**definition** *Basis-list-vec* :: ('a, 'b) *vec list* **where**

```

  Basis-list-vec = concat (map (λn. map (axis n) Basis-list) enum-class.enum)

```

**definition** *eucl-down-vec* :: int ⇒ ('a, 'b) *vec* ⇒ ('a, 'b) *vec* **where**

```

  eucl-down-vec p x = (χ i. eucl-down p (x $ i))

```

**definition** *eucl-truncate-down-vec* :: nat ⇒ ('a, 'b) *vec* ⇒ ('a, 'b) *vec* **where**

```

  eucl-truncate-down-vec p x = (χ i. eucl-truncate-down p (x $ i))

```

**definition** *eucl-truncate-up-vec* :: nat ⇒ ('a, 'b) *vec* ⇒ ('a, 'b) *vec* **where**

```

  eucl-truncate-up-vec p x = (χ i. eucl-truncate-up p (x $ i))

```

**instance**

**proof**

```

  show *: set (Basis-list::('a, 'b) vec list) = Basis

```

```

  unfolding Basis-list-vec-def Basis-vec-def

```

```

  apply (auto simp: Basis-list-vec-def vec-eq-iff distinct-map Basis-vec-def
  intro!: distinct-concat inj-onI split: if-splits)

```

```

  apply (auto simp: Basis-list-vec-def vec-eq-iff distinct-map enum-distinct
  UNIV-enum[symmetric])

```

```

  intro!: distinct-concat inj-onI split: if-splits)

```

```

  done

```

**have**  $\text{length} (\text{Basis-list}::('a, 'b) \text{ vec list}) = \text{CARD}('b) * \text{DIM}('a)$   
**by** (*auto simp: Basis-list-vec-def length-concat o-def enum-distinct*  
*sum-list-distinct-conv-sum-set UNIV-enum[symmetric]*)  
**then show**  $\text{distinct} (\text{Basis-list}::('a, 'b) \text{ vec list})$   
**using \* by** (*auto intro!: card-distinct*)  
**qed** (*simp-all only: vector-cart[symmetric] vec-eq-iff*  
*eucl-down-vec-def eucl-down-def*  
*eucl-truncate-down-vec-def eucl-truncate-down-def*  
*eucl-truncate-up-vec-def eucl-truncate-up-def,*  
*auto simp: euclidean-vec-componentwise inner-axis Basis-list-vec-def*  
*vec-nth-inner-scaleR-craziness*  
*intro!: sum.cong[OF refl]*)  
**end**

**lemma** *concat-same-lengths-nth:*  
**assumes**  $\bigwedge xs. xs \in \text{set } XS \implies \text{length } xs = N$   
**assumes**  $i < \text{length } XS * N \quad N > 0$   
**shows**  $\text{concat } XS ! i = XS ! (i \text{ div } N) ! (i \text{ mod } N)$   
**using** *assms* **by** (*induction XS arbitrary: i*)  
*(auto simp: nth-append nth-Cons div-eq-0-iff le-div-geq le-mod-geq split: nat.splits)*

**lemma** *concat-map-map-index:*  
**shows**  $\text{concat} (\text{map } (\lambda n. \text{map } (f \ n) \ xs) \ ys) =$   
 $\text{map } (\lambda i. f \ (ys ! (i \text{ div } \text{length } xs)) \ (xs ! (i \text{ mod } \text{length } xs))) [0..<\text{length } xs * \text{length } ys]$   
**apply** (*auto intro!: nth-equalityI simp: length-concat o-def sum-list-sum-nth*)  
**apply** (*subst concat-same-lengths-nth*)  
**apply** *auto*  
**apply** (*subst nth-map-upt*)  
**apply** (*auto simp: ac-simps*)  
**apply** (*subst nth-map*)  
**apply** (*metis div-eq-0-iff div-mult2-eq mult.commute mult-0 not-less0*)  
**apply** (*subst nth-map*)  
**subgoal for**  $i$   
**using** *gr-implies-not-zero* **by** *fastforce*  
**subgoal by** *simp*  
**done**

**lemma**  
*sum-list-zip-map:*  
**assumes**  $\text{distinct } xs$   
**shows**  $(\sum (x, y) \leftarrow \text{zip } xs \ (\text{map } g \ xs). f \ x \ y) = (\sum x \in \text{set } xs. f \ x \ (g \ x))$   
**by** (*force simp add: sum-list-distinct-conv-sum-set assms distinct-zipI1 split-beta'*  
*in-set-zip in-set-conv-nth inj-on-convol-ident*  
*intro!: sum.reindex-cong[where  $l = \lambda x. (x, g \ x)$ ]*)

**lemma**  
*sum-list-zip-map-of:*

**assumes** *distinct bs*  
**assumes** *length xs = length bs*  
**shows**  $(\sum (x, y) \leftarrow \text{zip } xs \text{ } bs. f x y) = (\sum x \in \text{set } bs. f (\text{the } (\text{map-of } (\text{zip } bs \text{ } xs) \text{ } x)) x)$   
**proof** –  
**have**  $(\sum (x, y) \leftarrow \text{zip } xs \text{ } bs. f x y) = (\sum (y, x) \leftarrow \text{zip } bs \text{ } xs. f x y)$   
**by** (*subst zip-commute*) (*auto simp: o-def split-beta'*)  
**also have**  $\dots = (\sum (x, y) \leftarrow \text{zip } bs \text{ } (\text{map } (\text{the } o \text{ map-of } (\text{zip } bs \text{ } xs)) \text{ } bs). f y x)$   
**proof** (*rule arg-cong, rule map-cong*)  
**have**  $xs = (\text{map } (\text{the } o \text{ map-of } (\text{zip } bs \text{ } xs)) \text{ } bs)$   
**using** *assms*  
**by** (*auto intro!: nth-equalityI simp: map-nth map-of-zip-nth*)  
**then show**  $\text{zip } bs \text{ } xs = \text{zip } bs \text{ } (\text{map } (\text{the } o \text{ map-of } (\text{zip } bs \text{ } xs)) \text{ } bs)$   
**by** *simp*  
**qed** *auto*  
**also have**  $\dots = (\sum x \in \text{set } bs. f (\text{the } (\text{map-of } (\text{zip } bs \text{ } xs) \text{ } x)) x)$   
**using** *assms(1)*  
**by** (*subst sum-list-zip-map*) (*auto simp: o-def*)  
**finally show** *?thesis* .  
**qed**

**lemma** *vec-nth-matrix*:  
 $\text{vec-nth } (\text{vec-nth } (\text{matrix } y) \text{ } i) \text{ } j = \text{vec-nth } (y \text{ } (\text{axis } j \text{ } 1)) \text{ } i$   
**unfolding** *matrix-def* **by** *simp*

**lemma** *matrix-eqI*:  
**assumes**  $\bigwedge x. x \in \text{Basis} \implies A *v x = B *v x$   
**shows**  $(A::\text{real}^n \times \text{real}^n) = B$   
**apply** *vector*  
**using** *assms*  
**apply** (*auto simp: Basis-vec-def*)  
**by** (*metis cart-eq-inner-axis matrix-vector-mul-component*)

**lemma** *matrix-columnI*:  
**assumes**  $\bigwedge i. \text{column } i \text{ } A = \text{column } i \text{ } B$   
**shows**  $(A::\text{real}^n \times \text{real}^n) = B$   
**using** *assms*  
**apply** *vector*  
**apply** (*auto simp: column-def*)  
**apply** *vector*  
**by** (*metis iso-tuple-UNIV-I vec-lambda-inject*)

**lemma**  
*vec-nth-Basis*:  
**fixes**  $x::\text{real}^n$   
**shows**  $x \in \text{Basis} \implies \text{vec-nth } x \text{ } i = (\text{if } x = \text{axis } i \text{ } 1 \text{ then } 1 \text{ else } 0)$   
**apply** (*auto simp: Basis-vec-def*)

```

by (metis cart-eq-inner-axis inner-axis-axis)

lemma vec-nth-eucl-of-list-eq: length M = CARD('n)  $\implies$ 
  vec-nth (eucl-of-list M::real'n::enum) i = M ! index Basis-list (axis i (1::real))
  apply (auto simp: eucl-of-list-def)
  apply (subst sum-list-zip-map-of)
  apply (auto intro!: distinct-zipI2 simp: split-beta')
  apply (subst sum.cong[OF refl])
  apply (subst vec-nth-Basis)
  apply (force simp: set-zip)
  apply (rule refl)
  apply (auto simp: if-distrib sum.delta cong: if-cong)
  subgoal
    apply (cases map-of (zip Basis-list M) (axis i 1::real'n::enum))
    subgoal premises prems
      proof -
        have fst ' set (zip Basis-list M) = (Basis::(real'n::enum) set) using prems
          by (auto simp: in-set-zip)
        then show ?thesis
          using prems
          by (subst (asm) map-of-eq-None-iff) simp
      qed
    subgoal for a
      apply (auto simp: in-set-zip)
      subgoal premises prems for n
        by (metis DIM-cart DIM-real index-Basis-list-nth mult.right-neutral prems(2)
prems(3))
      done
    done
  done
done

lemma index-Basis-list-axis1: index Basis-list (axis i (1::real)) = index enum-class.enum
i
  apply (auto simp: Basis-list-vec-def Basis-list-real-def )
  apply (subst index-map-inj)
  by (auto intro!: injI simp: axis-eq-axis)

lemma vec-nth-eq-list-of-eucl1:
  (vec-nth (M::real'n::enum) i) = list-of-eucl M ! (index enum-class.enum i)
  apply (subst eucl-of-list-list-of-eucl[of M, symmetric])
  apply (subst vec-nth-eucl-of-list-eq)
  unfolding index-Basis-list-axis1
  by auto

lemma enum-3[simp]: (enum-class.enum::3 list) = [0, 1, 2]
  by code-simp+

lemma three-eq-zero: (3::3) = 0 by simp

```

**lemma** *forall-3'*:  $(\forall i::3. P\ i) \longleftrightarrow P\ 0 \wedge P\ 1 \wedge P\ 2$   
**using** *forall-3 three-eq-zero* **by** *auto*

**lemma** *euclidean-eq-list-of-euclI*:  $x = y$  **if** *list-of-eucl*  $x =$  *list-of-eucl*  $y$   
**using** *that*  
**by** (*metis eucl-of-list-list-of-eucl*)

**lemma** *axis-one-neq-zero[simp]*: *axis*  $xa$  ( $1::'a::zero-neq-one$ )  $\neq 0$   
**by** (*auto simp: axis-def vec-eq-iff*)

**lemma** *eucl-of-list-vec-nth3[simp]*:  
*(eucl-of-list*  $[g, h, i]::real^3$ )  $\$ 0 = g$   
*(eucl-of-list*  $[g, h, i]::real^3$ )  $\$ 1 = h$   
*(eucl-of-list*  $[g, h, i]::real^3$ )  $\$ 2 = i$   
*(eucl-of-list*  $[g, h, i]::real^3$ )  $\$ 3 = g$   
**by** (*auto simp: cart-eq-inner-axis eucl-of-list-inner vec-nth-eq-list-of-eucl1 in-dex-Basis-list-axis1*)

**type-synonym**  $R3 = real*real*real$

**lemma** *Basis-list-R3*: *Basis-list*  $= [(1,0,0), (0, 1, 0), (0, 0, 1)::R3]$   
**by** (*auto simp: Basis-list-prod-def Basis-list-real-def zero-prod-def*)

**lemma** *Basis-list-vec3*: *Basis-list*  $= [axis\ 0\ 1::real^3, axis\ 1\ 1, axis\ 2\ 1]$   
**by** (*auto simp: Basis-list-vec-def Basis-list-real-def*)

**lemma** *eucl-of-list3[simp]*: *eucl-of-list*  $[a, b, c] = (a, b, c)$   
**by** (*auto simp: eucl-of-list-inner Basis-list-vec-def zero-prod-def Basis-prod-def Basis-list-vec3 Basis-list-R3 intro!: euclidean-eqI[where 'a=R3]*)

## 1.4 Bounded Linear Functions

### 1.5 bounded linear functions

**locale** *blinfun-syntax*

**begin**

**no-notation** *vec-nth* (**infixl**  $\langle \$ \rangle$  90)

**notation** *blinfun-apply* (**infixl**  $\langle \$ \rangle$  999)

**end**

**lemma** *bounded-linear-via-derivative*:

**fixes**  $f::'a::real-normed-vector \Rightarrow 'b::euclidean-space \Rightarrow_L 'c::real-normed-vector$

— **TODO**: generalize?

**assumes**  $\bigwedge i. ((\lambda x. blinfun-apply\ (f\ x)\ i)\ has-derivative\ (\lambda x. f'\ y\ x\ i))\ (at\ y)$

**shows** *bounded-linear*  $(f'\ y\ x)$

**proof** —

**interpret** *linear*  $f'\ y\ x$

**proof** (*unfold-locales, goal-cases*)

```

  case (1 v w)
  from has-derivative-unique[OF assms[of v + w, unfolded blinfun.bilinear-simps]
    has-derivative-add[OF assms[of v] assms[of w]], THEN fun-cong, of x]
  show ?case .
next
  case (2 r v)
  from has-derivative-unique[OF assms[of r *R v, unfolded blinfun.bilinear-simps]
    has-derivative-scaleR-right[OF assms[of v], of r], THEN fun-cong, of x]
  show ?case .
qed
let ?bnd =  $\sum_{i \in \text{Basis.}} \text{norm } (f' y x i)$ 
{
  fix v
  have  $f' y x v = (\sum_{i \in \text{Basis.}} (v \cdot i) *_{\text{R}} f' y x i)$ 
  by (subst euclidean-representation[symmetric]) (simp add: sum scaleR)
  also have  $\text{norm } \dots \leq \text{norm } v * ?bnd$ 
  by (auto intro!: order.trans[OF norm-sum] sum-mono mult-right-mono
    simp: sum-distrib-left Basis-le-norm)
  finally have  $\text{norm } (f' y x v) \leq \text{norm } v * ?bnd$  .
}
then show ?thesis by unfold-locales auto
qed

definition blinfun-scaleR::('a::real-normed-vector  $\Rightarrow_L$  real)  $\Rightarrow$  'b::real-normed-vector
 $\Rightarrow$  ('a  $\Rightarrow_L$  'b)
  where blinfun-scaleR a b = blinfun-scaleR-left b oL a

lemma blinfun-scaleR-transfer[transfer-rule]:
  rel-fun (pcr-blinfun (=) (=)) (rel-fun (=) (pcr-blinfun (=) (=)))
  ( $\lambda a b c. a c *_{\text{R}} b$ ) blinfun-scaleR
  by (auto simp: blinfun-scaleR-def rel-fun-def pcr-blinfun-def cr-blinfun-def OO-def)

lemma blinfun-scaleR-rep-eq[simp]:
  blinfun-scaleR a b c = a c *R b
  by (simp add: blinfun-scaleR-def)

lemma bounded-linear-blinfun-scaleR: bounded-linear (blinfun-scaleR a)
  unfolding blinfun-scaleR-def[abs-def]
  by (auto intro!: bounded-linear-intros)

lemma blinfun-scaleR-has-derivative[derivative-intros]:
  assumes (f has-derivative f') (at x within s)
  shows (( $\lambda x. \text{blinfun-scaleR } a (f x)$ ) has-derivative ( $\lambda x. \text{blinfun-scaleR } a (f' x)$ ))
  (at x within s)
  using bounded-linear-blinfun-scaleR assms
  by (rule bounded-linear.has-derivative)

lemma blinfun-componentwise:
  fixes f::'a::real-normed-vector  $\Rightarrow$  'b::euclidean-space  $\Rightarrow_L$  'c::real-normed-vector

```

**shows**  $f = (\lambda x. \sum_{i \in \text{Basis}} \text{blinfun-scaleR } (\text{blinfun-inner-left } i) (f x i))$   
**by** (*auto intro!*: *blinfun-eqI*  
*simp*: *blinfun.sum-left euclidean-representation blinfun.scaleR-right[symmetric]*  
*blinfun.sum-right[symmetric]*)

**lemma**

*blinfun-has-derivative-componentwiseI*:  
**fixes**  $f :: 'a :: \text{real-normed-vector} \Rightarrow 'b :: \text{euclidean-space} \Rightarrow_L 'c :: \text{real-normed-vector}$   
**assumes**  $\bigwedge i. i \in \text{Basis} \implies ((\lambda x. f x i) \text{ has-derivative } \text{blinfun-apply } (f' i)) (at x)$   
**shows**  $(f \text{ has-derivative } (\lambda x. \sum_{i \in \text{Basis}} \text{blinfun-scaleR } (\text{blinfun-inner-left } i) (f' i x))) (at x)$   
**by** (*subst blinfun-componentwise*) (*force intro: derivative-eq-intros assms simp: blinfun.bilinear-simps*)

**lemma**

*has-derivative-BlinfunI*:  
**fixes**  $f :: 'a :: \text{real-normed-vector} \Rightarrow 'b :: \text{euclidean-space} \Rightarrow_L 'c :: \text{real-normed-vector}$   
**assumes**  $\bigwedge i. ((\lambda x. f x i) \text{ has-derivative } (\lambda x. f' y x i)) (at y)$   
**shows**  $(f \text{ has-derivative } (\lambda x. \text{Blinfun } (f' y x))) (at y)$   
**proof** –  
**have** 1:  $f = (\lambda x. \sum_{i \in \text{Basis}} \text{blinfun-scaleR } (\text{blinfun-inner-left } i) (f x i))$   
**by** (*rule blinfun-componentwise*)  
**moreover have** 2:  $(\dots \text{ has-derivative } (\lambda x. \sum_{i \in \text{Basis}} \text{blinfun-scaleR } (\text{blinfun-inner-left } i) (f' y x i))) (at y)$   
**by** (*force intro: assms derivative-eq-intros*)  
**moreover**  
**interpret**  $f'$ : *bounded-linear f' y x for x*  
**by** (*rule bounded-linear-via-derivative*) (*rule assms*)  
**have** 3:  $(\sum_{i \in \text{Basis}} \text{blinfun-scaleR } (\text{blinfun-inner-left } i) (f' y x i)) i = f' y x i$   
**for**  $x i$   
**by** (*auto simp: if-distrib if-distribR blinfun.bilinear-simps*  
*f'.scaleR[symmetric] f'.sum[symmetric] euclidean-representation*  
*intro!: blinfun-euclidean-eqI*)  
**have** 4:  $\text{blinfun-apply } (\text{Blinfun } (f' y x)) = f' y x$  **for**  $x$   
**apply** (*subst bounded-linear-Blinfun-apply*)  
**subgoal by** *unfold-locales*  
**subgoal by** *simp*  
**done**  
**show** *?thesis*  
**apply** (*subst 1*)  
**apply** (*rule 2[THEN has-derivative-eq-rhs]*)  
**apply** (*rule ext*)  
**apply** (*rule blinfun-eqI*)  
**apply** (*subst 3*)  
**apply** (*subst 4*)  
**apply** (*rule refl*)  
**done**  
**qed**



**lemma**

*has-derivative-Blinfun:*

**assumes** (*f has-derivative f'*) *F*

**shows** (*f has-derivative Blinfun f'*) *F*

**using** *assms*

**by** (*subst bounded-linear-Blinfun-apply*) *auto*

**lift-definition** *flip-blinfun::*

(*'a::real-normed-vector*  $\Rightarrow_L$  *'b::real-normed-vector*  $\Rightarrow_L$  *'c::real-normed-vector*)  $\Rightarrow$

*'b*  $\Rightarrow_L$  *'a*  $\Rightarrow_L$  *'c* **is**

$\lambda f x y. f y x$

**using** *bounded-bilinear.bounded-linear-left bounded-bilinear.bounded-linear-right bounded-bilinear.flip*

**by** *auto*

**lemma** *flip-blinfun-apply[simp]: flip-blinfun f a b = f b a*

**by** *transfer simp*

**lemma** *le-norm-blinfun:*

**shows** *norm (blinfun-apply f x) / norm x*  $\leq$  *norm f*

**by** *transfer (rule le-onorm)*

**lemma** *norm-flip-blinfun[simp]: norm (flip-blinfun x) = norm x (is ?l = ?r)*

**proof** (*rule antisym*)

**from** *order-trans[OF norm-blinfun, OF mult-right-mono, OF norm-blinfun, OF norm-ge-zero, of x]*

**show** *?l*  $\leq$  *?r*

**by** (*auto intro!: norm-blinfun-bound simp: ac-simps*)

**have** *norm (x a b)  $\leq$  norm (flip-blinfun x) \* norm a \* norm b* **for** *a b*

**proof** –

**have** *norm (x a b) / norm a  $\leq$  norm (flip-blinfun x b)*

**by** (*rule order-trans[OF le-norm-blinfun]*) *auto*

**also have**  $\dots \leq$  *norm (flip-blinfun x) \* norm b*

**by** (*rule norm-blinfun*)

**finally show** *?thesis*

**by** (*auto simp add: divide-simps blinfun.bilinear-simps algebra-simps split: if-split-asm*)

**qed**

**then show** *?r*  $\leq$  *?l*

**by** (*auto intro!: norm-blinfun-bound*)

**qed**

**lemma** *bounded-linear-flip-blinfun[bounded-linear]: bounded-linear flip-blinfun*

**by** *unfold-locales (auto simp: blinfun.bilinear-simps intro!: blinfun-eqI exI[where x=1])*

**lemma** *dist-swap2-swap2[simp]: dist (flip-blinfun f) (flip-blinfun g) = dist f g*

**by** (*metis (no-types) bounded-linear-flip-blinfun dist-blinfun-def linear-simps(2) norm-flip-blinfun*)

**context includes** *blinfun.lifting* **begin**

**lift-definition** *blinfun-of-vmatrix*::( $\text{real}^m \text{ } ^n$ )  $\Rightarrow$  ( $(\text{real}^{m::\text{finite}}) \Rightarrow_L (\text{real}^{n::\text{finite}})$ )

**is**

*matrix-vector-mult*:: ( $(\text{real}, 'm) \text{ vec}, 'n) \text{ vec} \Rightarrow ((\text{real}, 'm) \text{ vec} \Rightarrow (\text{real}, 'n) \text{ vec})$

**unfolding** *linear-linear*

**by** (*rule matrix-vector-mul-linear*)

**lemma** *matrix-blinfun-of-vmatrix[simp]*:  $\text{matrix} (\text{blinfun-of-vmatrix } M) = M$

**apply** *vector*

**apply** (*auto simp: matrix-def*)

**apply** *transfer*

**by** (*metis cart-eq-inner-axis matrix-vector-mul-component*)

**end**

**lemma** *blinfun-apply-componentwise*:

$B = (\sum_{i \in \text{Basis}} \text{blinfun-scaleR} (\text{blinfun-inner-left } i) (\text{blinfun-apply } B \ i))$

**using** *blinfun-componentwise[of  $\lambda x. B$ , unfolded fun-eq-iff]*

**by** *blast*

**lemma** *blinfun-apply-eq-sum*:

**assumes** [*simp*]:  $\text{length } v = \text{CARD}('n)$

**shows**  $\text{blinfun-apply} (B::(\text{real}^{n::\text{enum}}) \Rightarrow_L (\text{real}^{m::\text{enum}})) (\text{eucl-of-list } v) =$

$(\sum_{i < \text{CARD}('m)}. \sum_{j < \text{CARD}('n)}. ((B (\text{Basis-list } ! j) \cdot \text{Basis-list } ! i) * v ! j) *_R (\text{Basis-list } ! i))$

**apply** (*subst blinfun-apply-componentwise[of B]*)

**apply** (*auto intro!: euclidean-eqI[where 'a=(real,'m) vec]*)

*simp: blinfun.bilinear-simps eucl-of-list-inner inner-sum-left inner-Basis if-distrib*

*sum-Basis-sum-nth-Basis-list nth-eq-iff-index if-distribR*

*cong: if-cong*)

**apply** (*subst sum.swap*)

**by** (*auto simp: sum.delta algebra-simps*)

**lemma** *in-square-lemma[intro, simp]*:  $x * C + y < D * C$  **if**  $x < D$   $y < C$  **for**

$x::\text{nat}$

**proof** –

**have**  $x * C + y < (D - 1) * C + C$

**apply** (*rule add-le-less-mono*)

**apply** (*rule mult-right-mono*)

**using** *that*

**by** *auto*

**also have**  $\dots \leq D * C$

**using** *that*

**by** (*auto simp: algebra-simps*)

**finally show** *?thesis* .

**qed**

**lemma** *less-square-imp-div-less*[*intro, simp*]:  $i < E * D \implies i \text{ div } E < D$  **for**  $i::\text{nat}$   
**by** (*metis div-eq-0-iff div-mult2-eq gr-implies-not0 mult-not-zero*)

**lemma** *in-square-lemma'*[*intro, simp*]:  $i < L \implies n < N \implies i * N + n < N * L$   
**for**  $i n::\text{nat}$   
**by** (*metis in-square-lemma mult.commute*)

**lemma**  
*distinct-nth-eq-iff*:  
 $\text{distinct } xs \implies x < \text{length } xs \implies y < \text{length } xs \implies xs ! x = xs ! y \longleftrightarrow x = y$   
**by** (*drule inj-on-nth[where I={..)*

**lemma** *index-Basis-list-axis2*:  
 $\text{index Basis-list (axis (j::'j)::\text{enum}) (axis (i::'i)::\text{enum}) (1::\text{real}))} =$   
 $(\text{index enum-class.enum } j) * \text{CARD}('i) + \text{index enum-class.enum } i$   
**apply** (*auto simp: Basis-list-vec-def Basis-list-real-def o-def*)  
**apply** (*subst concat-map-map-index*)  
**unfolding** *card-UNIV-length-enum[symmetric]*  
**subgoal**  
**proof** –  
**have** *index-less-cardi*:  $\text{index enum-class.enum } k < \text{CARD}('i)$  **for**  $k::'i$   
**by** (*rule index-less*) (*auto simp: enum-UNIV card-UNIV-length-enum*)  
**have** *index-less-cardj*:  $\text{index enum-class.enum } k < \text{CARD}('j)$  **for**  $k::'j$   
**by** (*rule index-less*) (*auto simp: enum-UNIV card-UNIV-length-enum*)  
**have** \*:  $\text{axis } j (\text{axis } i 1) =$   
 $(\lambda i. \text{axis (enum-class.enum ! (i \text{ div } \text{CARD}('i)))}$   
 $\quad (\text{axis (enum-class.enum ! (i \text{ mod } \text{CARD}('i))) 1}))$   
 $((\text{index enum-class.enum } j) * \text{CARD}('i) + \text{index enum-class.enum } i)$   
**by** (*auto simp: index-less-cardi enum-UNIV*)  
**note** *less=in-square-lemma[OF index-less-cardj index-less-cardi, of j i]*  
**show** *?thesis*  
**apply** (*subst \**)  
**apply** (*subst index-map-inj-on[where S={..)  
**subgoal**  
**apply** (*auto intro!: inj-onI simp: axis-eq-axis*)  
**apply** (*subst (asm) distinct-nth-eq-iff*)  
**apply** (*auto simp: enum-distinct card-UNIV-length-enum*)  
**subgoal for**  $x y$   
**using** *gr-implies-not0* **by** *fastforce*  
**subgoal for**  $x y$   
**using** *gr-implies-not0* **by** *fastforce*  
**subgoal for**  $x y$   
**apply** (*drule inj-onD[OF inj-on-nth[OF enum-distinct[where 'a='j],*  
**where**  $I = \{.., *rotated*])  
**apply** (*auto simp: card-UNIV-length-enum mult.commute*)  
**subgoal**  
**by** (*metis mod-mult-div-eq*)  
**done**$*

```

    done
  subgoal using less by auto
  subgoal by (auto simp: card-UNIV-length-enum ac-simps)
  subgoal apply (subst index-upt)
    subgoal using less by auto
    subgoal using less by (auto simp: ac-simps)
    subgoal using less by auto
  done
done
qed
done

lemma
  vec-nth-Basis2:
  fixes  $x::\text{real}^n \text{ } ^m$ 
  shows  $x \in \text{Basis} \implies \text{vec-nth} (\text{vec-nth } x \ i) \ j = ((\text{if } x = \text{axis } i \ (\text{axis } j \ 1) \ \text{then } 1 \ \text{else } 0))$ 
  by (auto simp: Basis-vec-def axis-def)

lemma vec-nth-eucl-of-list-eq2:  $\text{length } M = \text{CARD}(^n) * \text{CARD}(^m) \implies$ 
   $\text{vec-nth} (\text{vec-nth} (\text{eucl-of-list } M::\text{real}^n::\text{enum}^m::\text{enum}) \ i) \ j = M \ ! \ \text{index } \text{Basis-list} \ (\text{axis } i \ (\text{axis } j \ 1::\text{real}))$ 
  apply (auto simp: eucl-of-list-def)
  apply (subst sum-list-zip-map-of)
  apply (auto intro!: distinct-zipI2 simp: split-beta')
  apply (subst sum.cong[OF refl])
  apply (subst vec-nth-Basis2)
  apply (force simp: set-zip)
  apply (rule refl)
  apply (auto simp: if-distrib sum.delta cong: if-cong)
  subgoal
    apply (cases map-of (zip Basis-list M) (axis i (axis j 1):: $\text{real}^n::\text{enum}^m::\text{enum}$ ))
    subgoal premises prems
      proof -
        have fst ' set (zip Basis-list M) = (Basis::( $\text{real}^n::\text{enum}^m::\text{enum}$ ) set)
      using prems
        by (auto simp: in-set-zip)
      then show ?thesis
        using prems
        by (subst (asm) map-of-eq-None-iff) auto
    qed
  subgoal for a
    apply (auto simp: in-set-zip)
    subgoal premises prems for n
      proof -
        have  $n < \text{card} (\text{Basis}::(\text{real}^n::\text{enum}^m::\text{enum}) \ \text{set})$ 
        by (simp add: prems(4))
        then show ?thesis
        by (metis index-Basis-list-nth prems(2))
      qed
    qed
  qed

```

qed  
done  
done  
done

**lemma** *vec-nth-eq-list-of-eucl2*:

*vec-nth (vec-nth (M::real<sup>'n</sup>::enum<sup>'m</sup>::enum) i) j =*  
*list-of-eucl M ! (index enum-class.enum i \* CARD('n) + index enum-class.enum*  
*j)*  
**apply** (*subst eucl-of-list-list-of-eucl[of M, symmetric]*)  
**apply** (*subst vec-nth-eucl-of-list-eq2*)  
**unfolding** *index-Basis-list-axis2*  
**by** *auto*

**theorem**

*eucl-of-list-matrix-vector-mult-eq-sum-nth-Basis-list:*  
**assumes** *length M = CARD('n) \* CARD('m)*  
**assumes** *length v = CARD('n)*  
**shows** (*eucl-of-list M::real<sup>'n</sup>::enum<sup>'m</sup>::enum) \*v eucl-of-list v =*  
*( $\sum_{i < \text{CARD}('m)}$*   
*( $\sum_{j < \text{CARD}('n)}$ . M ! (i \* CARD('n) + j) \* v ! j) \*<sub>R</sub> Basis-list ! i)*  
**apply** (*vector matrix-vector-mult-def*)  
**apply** *auto*  
**apply** (*subst vec-nth-eucl-of-list-eq2*)  
**apply** (*auto simp: assms*)  
**apply** (*subst vec-nth-eucl-of-list-eq*)  
**apply** (*auto simp: assms index-Basis-list-axis2 index-Basis-list-axis1 vec-nth-Basis*  
*sum.delta*  
*nth-eq-iff-index*  
*if-distrib cong: if-cong*)  
**subgoal for** *i*  
**apply** (*rule sum.reindex-cong[where l=nth enum-class.enum]*)  
**apply** (*auto simp: enum-distinct card-UNIV-length-enum distinct-nth-eq-iff*  
*intro!: inj-onI*)  
**apply** (*rule image-eqI[OF ]*)  
**apply** (*rule nth-index[symmetric]*)  
**apply** (*auto simp: enum-UNIV*)  
**by** (*auto simp: algebra-simps enum-UNIV enum-distinct index-nth-id*)  
**subgoal for** *i*  
**using** *index-less[of i enum-class.enum CARD('n)]*  
**by** (*auto simp: enum-UNIV card-UNIV-length-enum*)  
**done**

**lemma** *index-enum-less[intro, simp]: index enum-class.enum (i::'n::enum) < CARD('n)*  
**by** (*auto intro!: index-less simp: enum-UNIV card-UNIV-length-enum*)

**lemmas** [*intro, simp*] = *enum-distinct*

**lemmas** [*simp*] = *card-UNIV-length-enum[symmetric] enum-UNIV*

```

lemma sum-index-enum-eq:
   $(\sum (k::'n::\text{enum}) \in \text{UNIV}. f (\text{index enum-class.enum } k)) = (\sum i < \text{CARD}('n). f i)$ 
  by (rule sum.reindex-cong[where  $l = \text{nth enum-class.enum}$ ])
  (force intro!: inj-onI simp: distinct-nth-eq-iff index-nth-id)+

```

**end**

## 2 Affine Form

**theory** *Affine-Form*

**imports**

*HOL-Analysis.Multivariate-Analysis*  
*HOL-Combinatorics.List-Permutation*  
*Affine-Arithmetic-Auxiliarities*  
*Executable-Euclidean-Space*

**begin**

### 2.1 Auxiliary developments

**lemma** *sum-list-mono*:

**fixes**  $xs\ ys::'a::\text{ordered-ab-group-add list}$

**shows**

$\text{length } xs = \text{length } ys \implies (\bigwedge x\ y. (x, y) \in \text{set } (\text{zip } xs\ ys) \implies x \leq y) \implies$   
 $\text{sum-list } xs \leq \text{sum-list } ys$

**by** (*induct xs ys rule: list-induct2*) (*auto simp: algebra-simps intro: add-mono*)

**lemma**

**fixes**  $xs::'a::\text{ordered-comm-monoid-add list}$

**shows** *sum-list-nonneg*:  $(\bigwedge x. x \in \text{set } xs \implies x \geq 0) \implies \text{sum-list } xs \geq 0$

**by** (*induct xs*) (*auto intro!: add-nonneg-nonneg*)

**lemma** *map-filter*:

$\text{map } f (\text{filter } (\lambda x. P (f x)) xs) = \text{filter } P (\text{map } f xs)$

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

**lemma**

*map-of-zip-upto2-length-eq-nth*:

**assumes** *distinct B*

**assumes**  $i < \text{length } B$

**shows**  $(\text{map-of } (\text{zip } B [0..<\text{length } B]) (B ! i)) = \text{Some } i$

**proof** –

**have**  $\text{length } [0..<\text{length } B] = \text{length } B$

**by** *simp*

**from** *map-of-zip-is-Some*[*OF this, of i*] *assms*

**have**  $\text{map-of } (\text{zip } B [0..<\text{length } B]) (B ! i) = \text{Some } i$

**using** *assms* **by** (*auto simp: in-set-zip*)

**thus** *?thesis* **by** *simp*

**qed**

**lemma** *distinct-map-fst-snd-eqD*:

$distinct (map\ fst\ xs) \implies (i, a) \in set\ xs \implies (i, b) \in set\ xs \implies a = b$   
**by** (*metis (lifting) map-of-is-SomeI option.inject*)

**lemma** *length-filter-snd-zip*:

$length\ ys = length\ xs \implies length\ (filter\ (p \circ snd)\ (zip\ ys\ xs)) = length\ (filter\ p\ xs)$   
**by** (*induct ys xs rule: list-induct2*) (*auto*)

**lemma** *filter-snd-nth*:

$length\ ys = length\ xs \implies n < length\ (filter\ p\ xs) \implies$   
 $snd\ (filter\ (p \circ snd)\ (zip\ ys\ xs)\ !\ n) = filter\ p\ xs\ !\ n$   
**by** (*induct ys xs arbitrary: n rule: list-induct2*) (*auto simp: o-def nth-Cons split: nat.split*)

**lemma** *distinct-map-snd-fst-eqD*:

$distinct (map\ snd\ xs) \implies (i, a) \in set\ xs \implies (j, a) \in set\ xs \implies i = j$   
**by** (*metis Pair-inject inj-on-contrad snd-conv distinct-map*)

**lemma** *map-of-mapk-inj-on-SomeI*:

$inj\ on\ f\ (fst\ ' (set\ t)) \implies map\ of\ t\ k = Some\ x \implies$   
 $map\ of\ (map\ (case\ prod\ (\lambda k. Pair\ (f\ k)))\ t)\ (f\ k) = Some\ x$   
**by** (*induct t*) (*auto simp add: inj-on-def dest!: map-of-SomeD split: if-split-asm*)

**lemma** *map-abs-nonneg[simp]*:

**fixes**  $xs::'a::ordered-ab-group-add-abs\ list$   
**shows**  $list\ all\ (\lambda x. x \geq 0)\ xs \implies map\ abs\ xs = xs$   
**by** (*induct xs*) *auto*

**lemma** *the-inv-into-image-eq*:  $inj\ on\ f\ A \implies Y \subseteq f\ ' A \implies the\ inv\ into\ A\ f\ ' Y = f\ -\ ' Y \cap A$

**using** *f-the-inv-into-f the-inv-into-f-f* [**where**  $f = f$  **and**  $A = A$ ]  
**by** *force*

**lemma** *image-fst-zip*:  $length\ ys = length\ xs \implies fst\ ' set\ (zip\ ys\ xs) = set\ ys$

**by** (*metis dom-map-of-conv-image-fst dom-map-of-zip*)

**lemma** *inj-on-fst-set-zip-distinct[simp]*:

$distinct\ xs \implies length\ xs = length\ ys \implies inj\ on\ fst\ (set\ (zip\ xs\ ys))$   
**by** (*force simp add: in-set-zip distinct-conv-nth intro!: inj-onI*)

**lemma** *mem-greaterThanLessThan-absI*:

**fixes**  $x::real$   
**assumes**  $abs\ x < 1$   
**shows**  $x \in \{-1 <..< 1\}$   
**using** *assms* **by** (*auto simp: abs-real-def split: if-split-asm*)

**lemma** *minus-one-less-divideI*:  $b > 0 \implies -b < a \implies -1 < a / (b::real)$

**by** (*auto simp: field-simps*)

```

lemma divide-less-oneI:  $b > 0 \implies b > a \implies a / (b::real) < 1$ 
  by (auto simp: field-simps)

lemma closed-segment-real:
  fixes  $a b::real$ 
  shows  $closed\_segment\ a\ b = (if\ a \leq b\ then\ \{a .. b\}\ else\ \{b .. a\})$  (is - = ?if)
proof safe
  fix  $x$  assume  $x \in closed\_segment\ a\ b$ 
  from segment-bound[OF this]
  show  $x \in ?if$  by (auto simp: abs-real-def split: if-split-asm)
next
  fix  $x$ 
  assume  $x \in ?if$ 
  thus  $x \in closed\_segment\ a\ b$ 
    by (auto simp: closed-segment-def intro!: exI[where  $x=(x-a)/(b-a)$ ]
      simp: divide-simps algebra-simps)
qed

```

## 2.2 Partial Deviations

```

typedef (overloaded)  $'a\ pdevs = \{x::nat \Rightarrow 'a::zero.\ finite\ \{i.\ x\ i \neq 0\}\}$ 
  — TODO: unify with polynomials
  morphisms pdevs-apply Abs-pdev
  by (auto intro!: exI[where  $x=\lambda x. 0$ ])

```

**setup-lifting** *type-definition-pdevs*

```

lemma pdevs-eqI:  $(\bigwedge i.\ pdevs\_apply\ x\ i = pdevs\_apply\ y\ i) \implies x = y$ 
  by transfer auto

```

```

definition pdevs-val ::  $(nat \Rightarrow real) \Rightarrow 'a::real-normed-vector\ pdevs \Rightarrow 'a$ 
  where  $pdevs\_val\ e\ x = (\sum i.\ e\ i *_{\mathbb{R}} pdevs\_apply\ x\ i)$ 

```

```

definition valuate::  $((nat \Rightarrow real) \Rightarrow 'a) \Rightarrow 'a\ set$ 
  where  $valuate\ x = x\ ' (UNIV \rightarrow \{-1 .. 1\})$ 

```

```

lemma valuate-ex:  $x \in valuate\ f \iff (\exists e.\ (\forall i.\ e\ i \in \{-1 .. 1\}) \wedge x = f\ e)$ 
  unfolding valuate-def
  by (auto simp add: valuate-def Pi-iff) blast

```

```

instantiation pdevs :: (equal) equal
begin

```

```

definition equal-pdevs::  $'a\ pdevs \Rightarrow 'a\ pdevs \Rightarrow bool$ 
  where  $equal\_pdevs\ a\ b \iff a = b$ 

```

```

instance proof qed (simp add: equal-pdevs-def)
end

```



## 2.3 Affine Forms

The data structure of affine forms represents particular sets, zonotopes

**type-synonym**  $'a \text{ aform} = 'a \times 'a \text{ pdevs}$

## 2.4 Evaluation, Range, Joint Range

**definition**  $\text{aform-val} :: (\text{nat} \Rightarrow \text{real}) \Rightarrow 'a::\text{real-normed-vector} \text{ aform} \Rightarrow 'a$   
**where**  $\text{aform-val } e \ X = \text{fst } X + \text{pdevs-val } e \ (\text{snd } X)$

**definition**  $\text{Affine} ::$   
 $'a::\text{real-normed-vector} \text{ aform} \Rightarrow 'a \text{ set}$   
**where**  $\text{Affine } X = \text{valuate } (\lambda e. \text{aform-val } e \ X)$

**definition**  $\text{Joints} ::$   
 $'a::\text{real-normed-vector} \text{ aform list} \Rightarrow 'a \text{ list set}$   
**where**  $\text{Joints } XS = \text{valuate } (\lambda e. \text{map } (\text{aform-val } e) \ XS)$

**lemma**  $\text{Joints-nthE}$ :  
**assumes**  $zs \in \text{Joints } ZS$   
**obtains**  $e$  **where**  
 $\bigwedge i. i < \text{length } zs \implies zs ! i = \text{aform-val } e \ (ZS ! i)$   
 $\bigwedge i. e \ i \in \{-1..1\}$   
**using**  $\text{assms}$   
**by**  $\text{atomize-elim } (\text{auto simp: Joints-def Pi-iff valuate-ex})$

**lemma**  $\text{Joints-mapE}$ :  
**assumes**  $ys \in \text{Joints } YS$   
**obtains**  $e$  **where**  
 $ys = \text{map } (\lambda x. \text{aform-val } e \ x) \ YS$   
 $\bigwedge i. e \ i \in \{-1 .. 1\}$   
**using**  $\text{assms}$   
**by**  $(\text{force simp: Joints-def valuate-def})$

**lemma**  
 $\text{zipped-subset-mapped-Elem}$ :  
**assumes**  $xs = \text{map } (\text{aform-val } e) \ XS$   
**assumes**  $e: \bigwedge i. e \ i \in \{-1 .. 1\}$   
**assumes**  $[\text{simp}]: \text{length } xs = \text{length } XS$   
**assumes**  $[\text{simp}]: \text{length } ys = \text{length } YS$   
**assumes**  $\text{set } (\text{zip } ys \ YS) \subseteq \text{set } (\text{zip } xs \ XS)$   
**shows**  $ys = \text{map } (\text{aform-val } e) \ YS$   
**proof** –  
**from**  $\text{assms}$  **have**  $ys: \bigwedge i. i < \text{length } xs \implies xs ! i = \text{aform-val } e \ (XS ! i)$   
**by**  $\text{auto}$   
**from**  $\text{assms}$  **have**  $\text{set-eq: } \{(ys ! i, YS ! i) \mid i. i < \text{length } ys \wedge i < \text{length } YS\} \subseteq$   
 $\{(xs ! i, XS ! i) \mid i. i < \text{length } xs \wedge i < \text{length } XS\}$   
**using**  $\text{assms}(2)$   
**by**  $(\text{auto simp: set-zip})$

**hence**  $\forall i < \text{length } YS. \exists j < \text{length } XS. ys ! i = xs ! j \wedge YS ! i = XS ! j$   
**by** *auto*  
**then obtain**  $j$  **where**  $j: \bigwedge i. i < \text{length } YS \implies ys ! i = xs ! (j i)$   
 $\bigwedge i. i < \text{length } YS \implies YS ! i = XS ! (j i)$   
 $\bigwedge i. i < \text{length } YS \implies j i < \text{length } XS$   
**by** *metis*  
**show** *?thesis*  
**using** *assms*  
**by** (*auto simp: Joints-def j ys intro!: exI[where x=e] nth-equalityI*)  
**qed**

**lemma** *Joints-set-zip-subset*:  
**assumes**  $xs \in \text{Joints } XS$   
**assumes**  $\text{length } xs = \text{length } XS$   
**assumes**  $\text{length } ys = \text{length } YS$   
**assumes**  $\text{set } (\text{zip } ys YS) \subseteq \text{set } (\text{zip } xs XS)$   
**shows**  $ys \in \text{Joints } YS$   
**proof** –  
**from** *Joints-mapE assms* **obtain**  $e$  **where**  
 $ys: xs = \text{map } (\lambda x. \text{aform-val } e x) XS$   
**and**  $e: \bigwedge i. e i \in \{-1 .. 1\}$   
**by** *blast*  
**show**  $ys \in \text{Joints } YS$   
**using** *e zipped-subset-mapped-Elem[OF ys e assms(2-4)]*  
**by** (*auto simp: Joints-def valuate-def intro!: exI[where x=e]*)  
**qed**

**lemma** *Joints-set-zip*:  
**assumes**  $ys \in \text{Joints } YS$   
**assumes**  $\text{length } xs = \text{length } XS$   
**assumes**  $\text{length } YS = \text{length } XS$   
**assumes** *sets-eq: set (zip xs XS) = set (zip ys YS)*  
**shows**  $xs \in \text{Joints } XS$   
**proof** –  
**from** *assms* **have**  $\text{length } ys = \text{length } YS$   
**by** (*auto simp: Joints-def valuate-def*)  
**from** *assms(1)* **this** *assms(2)* **show** *?thesis*  
**by** (*rule Joints-set-zip-subset (simp add: assms)*)  
**qed**

**definition** *Joints2* ::  
 $'a::\text{real-normed-vector aform list} \Rightarrow 'b::\text{real-normed-vector aform} \Rightarrow ('a \text{ list} \times 'b)$   
*set*  
**where**  $\text{Joints2 } XS Y = \text{valuate } (\lambda e. (\text{map } (\text{aform-val } e) XS, \text{aform-val } e Y))$

**lemma** *Joints2E*:  
**assumes**  $zs-y \in \text{Joints2 } ZS Y$   
**obtains**  $e$  **where**  
 $\bigwedge i. i < \text{length } (\text{fst } zs-y) \implies (\text{fst } zs-y) ! i = \text{aform-val } e (ZS ! i)$

$snd (zs-y) = aform-val e Y$   
 $\wedge i. e i \in \{-1..1\}$   
**using** *assms*  
**by** *atomize-elim (auto simp: Joints2-def Pi-iff valuate-ex)*

**lemma** *nth-in-AffineI*:  
**assumes**  $xs \in Joints\ XS$   
**assumes**  $i < length\ XS$   
**shows**  $xs ! i \in Affine\ (XS ! i)$   
**using** *assms* **by** *(force simp: Affine-def Joints-def valuate-def)*

**lemma** *Cons-nth-in-Joints1*:  
**assumes**  $xs \in Joints\ XS$   
**assumes**  $i < length\ XS$   
**shows**  $((xs ! i) \# xs) \in Joints\ ((XS ! i) \# XS)$   
**using** *assms* **by** *(force simp: Joints-def valuate-def)*

**lemma** *Cons-nth-in-Joints2*:  
**assumes**  $xs \in Joints\ XS$   
**assumes**  $i < length\ XS$   
**assumes**  $j < length\ XS$   
**shows**  $((xs ! i) \# (xs ! j) \# xs) \in Joints\ ((XS ! i) \# (XS ! j) \# XS)$   
**using** *assms* **by** *(force simp: Joints-def valuate-def)*

**lemma** *Joints-swap*:  
 $x \# y \# xs \in Joints\ (X \# Y \# XS) \longleftrightarrow y \# x \# xs \in Joints\ (Y \# X \# XS)$   
**by** *(force simp: Joints-def valuate-def)*

**lemma** *Joints-swap-Cons-append*:  
 $length\ xs = length\ XS \implies x \# ys @ xs \in Joints\ (X \# YS @ XS) \longleftrightarrow ys @ x \# xs \in Joints\ (YS @ X \# XS)$   
**by** *(auto simp: Joints-def valuate-def)*

**lemma** *Joints-ConsD*:  
 $x \# xs \in Joints\ (X \# XS) \implies xs \in Joints\ XS$   
**by** *(force simp: Joints-def valuate-def)*

**lemma** *Joints-appendD1*:  
 $ys @ xs \in Joints\ (YS @ XS) \implies length\ xs = length\ XS \implies xs \in Joints\ XS$   
**by** *(force simp: Joints-def valuate-def)*

**lemma** *Joints-appendD2*:  
 $ys @ xs \in Joints\ (YS @ XS) \implies length\ ys = length\ YS \implies ys \in Joints\ YS$   
**by** *(force simp: Joints-def valuate-def)*

**lemma** *Joints-imp-length-eq*:  $xs \in Joints\ XS \implies length\ xs = length\ XS$   
**by** *(auto simp: Joints-def valuate-def)*

**lemma** *Joints-rotate[simp]*:  $xs @ [x] \in Joints\ (XS @ [X]) \longleftrightarrow x \# xs \in Joints\ (X \# XS)$

by (auto simp: Joints-def valuate-def)

## 2.5 Domain

**definition**  $pdevs\text{-domain } x = \{i. pdevs\text{-apply } x \ i \neq 0\}$

**lemma**  $finite\text{-pdevs-domain}[intro, simp]: finite (pdevs\text{-domain } x)$   
**unfolding**  $pdevs\text{-domain-def}$  **by**  $transfer$

**lemma**  $in\text{-pdevs-domain}[simp]: i \in pdevs\text{-domain } x \iff pdevs\text{-apply } x \ i \neq 0$   
**by** (auto simp:  $pdevs\text{-domain-def}$ )

## 2.6 Least Fresh Index

**definition**  $degree::'a::real\text{-vector } pdevs \Rightarrow nat$   
**where**  $degree \ x = (LEAST \ i. \forall j \geq i. pdevs\text{-apply } x \ j = 0)$

**lemma**  $degree[rule\text{-format}, intro, simp]:$   
**shows**  $\forall j \geq degree \ x. pdevs\text{-apply } x \ j = 0$   
**unfolding**  $degree\text{-def}$

**proof** (rule  $LeastI\text{-ex}$ )

**have**  $\bigwedge j. j > Max (pdevs\text{-domain } x) \implies j \notin (pdevs\text{-domain } x)$

**by** (metis  $Max\text{-less-iff all-not-in-conv less-irrefl-nat finite-pdevs-domain}$ )

**then show**  $\exists xa. \forall j \geq xa. pdevs\text{-apply } x \ j = 0$

**by** (auto intro!:  $exI[\text{where } x=Max (pdevs\text{-domain } x) + 1]$ )

qed

**lemma**  $degree\text{-le}:$

**assumes**  $d: \forall j \geq d. pdevs\text{-apply } x \ j = 0$

**shows**  $degree \ x \leq d$

**unfolding**  $degree\text{-def}$

**by** (rule  $Least\text{-le}$ ) (rule  $d$ )

**lemma**  $degree\text{-gt}: pdevs\text{-apply } x \ j \neq 0 \implies degree \ x > j$

**by**  $auto$

**lemma**  $pdevs\text{-val-pdevs-domain}: pdevs\text{-val } e \ X = (\sum_{i \in pdevs\text{-domain } X}. e \ i \ *_{\mathbb{R}} pdevs\text{-apply } X \ i)$

**by** (auto simp:  $pdevs\text{-val-def intro!:$   $suminf\text{-finite}$ )

**lemma**  $pdevs\text{-val-sum-le}: degree \ X \leq d \implies pdevs\text{-val } e \ X = (\sum_{i < d}. e \ i \ *_{\mathbb{R}} pdevs\text{-apply } X \ i)$

**by** (force intro!:  $degree\text{-gt sum.mono-neutral-cong-left simp: pdevs\text{-val-pdevs-domain}$ )

**lemmas**  $pdevs\text{-val-sum} = pdevs\text{-val-sum-le}[OF \ order\text{-refl}]$

**lemma**  $pdevs\text{-val-zero}[simp]: pdevs\text{-val } (\lambda \cdot. 0) \ x = 0$

**by** (auto simp:  $pdevs\text{-val-sum}$ )

**lemma**  $degree\text{-eqI}:$

**assumes**  $pdevs\text{-}apply\ x\ d \neq 0$   
**assumes**  $\bigwedge j. j > d \implies pdevs\text{-}apply\ x\ j = 0$   
**shows**  $degree\ x = Suc\ d$   
**unfolding**  $eq\text{-}iff$   
**by** (*auto intro!*:  $degree\text{-}gt\ degree\text{-}le\ assms\ simp: Suc\text{-}le\text{-}eq$ )

**lemma**  $finite\text{-}degree\text{-}nonzero$ [*intro, simp*]:  $finite\ \{i. pdevs\text{-}apply\ x\ i \neq 0\}$   
**by** *transfer* (*auto simp: vimage\text{-}def Collect\text{-}neg\text{-}eq*)

**lemma**  $degree\text{-}eq\text{-}Suc\text{-}max$ :  
 $degree\ x = (if\ (\forall i. pdevs\text{-}apply\ x\ i = 0)\ then\ 0\ else\ Suc\ (Max\ \{i. pdevs\text{-}apply\ x\ i \neq 0\}))$   
**proof** –  
{  
  **assume**  $\bigwedge i. pdevs\text{-}apply\ x\ i = 0$   
  **hence** *?thesis*  
  **by** *auto* (*metis degree\text{-}le le\text{-}0\text{-}eq*)  
} **moreover** {  
  **fix**  $i$  **assume**  $pdevs\text{-}apply\ x\ i \neq 0$   
  **hence** *?thesis*  
  **using**  $Max\text{-}in$ [*OF finite\text{-}degree\text{-}nonzero, of\ x*]  
  **by** (*auto intro!*:  $degree\text{-}eqI$ ) (*metis Max.coboundedI*[*OF finite\text{-}degree\text{-}nonzero*]  
*in\text{-}pdevs\text{-}domain*  
 $le\text{-}eq\text{-}less\text{-}or\text{-}eq\ less\text{-}asym\ pdevs\text{-}domain\text{-}def$ )  
} **ultimately show** *?thesis*  
  **by** *blast*  
**qed**

**lemma**  $pdevs\text{-}val\text{-}degree\text{-}cong$ :  
**assumes**  $b = d$   
**assumes**  $\bigwedge i. i < degree\ b \implies a\ i = c\ i$   
**shows**  $pdevs\text{-}val\ a\ b = pdevs\text{-}val\ c\ d$   
**using** *assms*  
**by** (*auto simp: pdevs\text{-}val\text{-}sum*)

**abbreviation**  $degree\text{-}aform::'a::real\text{-}vector\ aform \Rightarrow nat$   
**where**  $degree\text{-}aform\ X \equiv degree\ (snd\ X)$

**lemma**  $degree\text{-}cong$ :  $(\bigwedge i. (pdevs\text{-}apply\ x\ i = 0) = (pdevs\text{-}apply\ y\ i = 0)) \implies degree\ x = degree\ y$   
**unfolding**  $degree\text{-}def$   
**by** *auto*

**lemma**  $Least\text{-}True\text{-}nat$ [*intro, simp*]:  $(LEAST\ i::nat. True) = 0$   
**by** (*metis* (*lifting*)  $One\text{-}nat\text{-}def\ less\text{-}one\ not\text{-}less\text{-}Least\ not\text{-}less\text{-}eq$ )

**lemma**  $sorted\text{-}list\text{-}of\text{-}pdevs\text{-}domain\text{-}eq$ :  
 $sorted\text{-}list\text{-}of\text{-}set\ (pdevs\text{-}domain\ X) = filter\ ((\neq)\ 0\ o\ pdevs\text{-}apply\ X)\ [0..<degree\ X]$

by (auto simp: degree-gt intro!: sorted-distinct-set-unique sorted-filter[of  $\lambda x. x$ , simplified])

## 2.7 Total Deviation

**definition**  $tdev::'a::ordered-euclidean-space\ pdevs \Rightarrow 'a$  **where**  
 $tdev\ x = (\sum\ i < degree\ x. |pdevs\ apply\ x\ i|)$

**lemma**  $abs-pdevs-val-le-tdev: e \in UNIV \rightarrow \{-1 .. 1\} \Longrightarrow |pdevs-val\ e\ x| \leq tdev\ x$   
**by** (force simp: pdevs-val-sum tdev-def abs-scaleR Pi-iff  
intro!: order-trans[OF sum-abs] sum-mono scaleR-left-le-one-le  
intro: abs-leI)

## 2.8 Binary Pointwise Operations

**definition**  $binop-pdevs-raw::('a::zero \Rightarrow 'b::zero \Rightarrow 'c::zero) \Rightarrow$   
 $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow nat \Rightarrow 'c$   
**where**  $binop-pdevs-raw\ f\ x\ y\ i = (if\ x\ i = 0 \wedge y\ i = 0\ then\ 0\ else\ f\ (x\ i)\ (y\ i))$

**lemma**  $nonzeros-binop-pdevs-subset:$   
 $\{i. binop-pdevs-raw\ f\ x\ y\ i \neq 0\} \subseteq \{i. x\ i \neq 0\} \cup \{i. y\ i \neq 0\}$   
**by** (auto simp: binop-pdevs-raw-def)

**lift-definition**  $binop-pdevs::$   
 $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a::zero\ pdevs \Rightarrow 'b::zero\ pdevs \Rightarrow 'c::zero\ pdevs$   
**is**  $binop-pdevs-raw$   
**using**  $nonzeros-binop-pdevs-subset$   
**by** (rule finite-subset) auto

**lemma**  $pdevs-apply-binop-pdevs[simp]: pdevs-apply\ (binop-pdevs\ f\ x\ y)\ i =$   
 $(if\ pdevs-apply\ x\ i = 0 \wedge pdevs-apply\ y\ i = 0\ then\ 0\ else\ f\ (pdevs-apply\ x\ i)$   
 $(pdevs-apply\ y\ i))$   
**by** transfer (auto simp: binop-pdevs-raw-def)

## 2.9 Addition

**definition**  $add-pdevs::'a::real-vector\ pdevs \Rightarrow 'a\ pdevs \Rightarrow 'a\ pdevs$   
**where**  $add-pdevs = binop-pdevs\ (+)$

**lemma**  $pdevs-apply-add-pdevs[simp]:$   
 $pdevs-apply\ (add-pdevs\ X\ Y)\ n = pdevs-apply\ X\ n + pdevs-apply\ Y\ n$   
**by** (auto simp: add-pdevs-def)

**lemma**  $pdevs-val-add-pdevs[simp]:$   
**fixes**  $x\ y::'a::euclidean-space$   
**shows**  $pdevs-val\ e\ (add-pdevs\ X\ Y) = pdevs-val\ e\ X + pdevs-val\ e\ Y$   
**proof** –  
**let**  $?sum = \lambda m\ X. \sum\ i < m. e\ i\ *_R\ pdevs-apply\ X\ i$   
**let**  $?m = \max\ (degree\ X)\ (degree\ Y)$   
**have**  $pdevs-val\ e\ X + pdevs-val\ e\ Y = ?sum\ (degree\ X)\ X + ?sum\ (degree\ Y)\ Y$

by (simp add: pdevs-val-sum)  
 also have  $?sum (degree X) X = ?sum ?m X$   
 by (rule sum.mono-neutral-cong-left) auto  
 also have  $?sum (degree Y) Y = ?sum ?m Y$   
 by (rule sum.mono-neutral-cong-left) auto  
 also have  $?sum ?m X + ?sum ?m Y = (\sum i < ?m. e i *_R (pdevs-apply X i + pdevs-apply Y i))$   
 by (simp add: scaleR-right-distrib sum.distrib)  
 also have  $\dots = (\sum i. e i *_R (pdevs-apply X i + pdevs-apply Y i))$   
 by (rule suminf-finite[symmetric]) auto  
 also have  $\dots = pdevs-val e (add-pdevs X Y)$   
 by (simp add: pdevs-val-def)  
 finally show  $pdevs-val e (add-pdevs X Y) = pdevs-val e X + pdevs-val e Y$  by  
 simp  
 qed

## 2.10 Total Deviation

**lemma** *tdev-eq-zero-iff*: fixes  $X::real\ pdevs$  shows  $tdev X = 0 \iff (\forall e. pdevs-val e X = 0)$

by (force simp add: pdevs-val-sum tdev-def sum-nonneg-eq-0-iff  
 dest!: spec[where  $x=\lambda i. if\ pdevs-apply\ X\ i \ge 0\ then\ 1\ else\ -1$ ] split: if-split-asm)

**lemma** *tdev-nonneg*[intro, simp]:  $tdev X \ge 0$

by (auto simp: tdev-def)

**lemma** *tdev-nonpos-iff*[simp]:  $tdev X \le 0 \iff tdev X = 0$

by (auto simp: order.antisym)

## 2.11 Unary Operations

**definition** *unop-pdevs-raw*::

$('a::zero \Rightarrow 'b::zero) \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'b$   
 where  $unop-pdevs-raw\ f\ x\ i = (if\ x\ i = 0\ then\ 0\ else\ f\ (x\ i))$

**lemma** *nonzeros-unop-pdevs-subset*:  $\{i. unop-pdevs-raw\ f\ x\ i \neq 0\} \subseteq \{i. x\ i \neq 0\}$

by (auto simp: unop-pdevs-raw-def)

**lift-definition** *unop-pdevs*::

$('a \Rightarrow 'b) \Rightarrow 'a::zero\ pdevs \Rightarrow 'b::zero\ pdevs$

is *unop-pdevs-raw*

using *nonzeros-unop-pdevs-subset*

by (rule finite-subset) auto

**lemma** *pdevs-apply-unop-pdevs*[simp]:  $pdevs-apply (unop-pdevs\ f\ x)\ i =$

$(if\ pdevs-apply\ x\ i = 0\ then\ 0\ else\ f\ (pdevs-apply\ x\ i))$

by transfer (auto simp: unop-pdevs-raw-def)

**lemma** *pdevs-domain-unop-linear*:

assumes *linear*  $f$

**shows**  $pdevs\text{-domain } (unop\text{-pdevs } f \ x) \subseteq pdevs\text{-domain } x$   
**proof** –  
**interpret**  $f$ : *linear*  $f$  **by fact**  
**show** *?thesis*  
**by** (*auto simp: f.zero*)  
**qed**

**lemma**  
*pdevs-val-unop-linear*:  
**assumes** *linear*  $f$   
**shows**  $pdevs\text{-val } e \ (unop\text{-pdevs } f \ x) = f \ (pdevs\text{-val } e \ x)$   
**proof** –  
**interpret**  $f$ : *linear*  $f$  **by fact**  
**have**  $*$ :  $\bigwedge i. (if \ pdevs\text{-apply } x \ i = 0 \ then \ 0 \ else \ f \ (pdevs\text{-apply } x \ i)) = f \ (pdevs\text{-apply } x \ i)$   
**by** (*auto simp: f.zero*)  
**have**  $pdevs\text{-val } e \ (unop\text{-pdevs } f \ x) =$   
 $(\sum i \in pdevs\text{-domain } (unop\text{-pdevs } f \ x). \ e \ i \ *_R \ f \ (pdevs\text{-apply } x \ i))$   
**by** (*auto simp add: pdevs-val-pdevs-domain \**)  
**also have**  $\dots = (\sum xa \in pdevs\text{-domain } x. \ e \ xa \ *_R \ f \ (pdevs\text{-apply } x \ xa))$   
**by** (*auto intro!: sum.mono-neutral-cong-left*)  
**also have**  $\dots = f \ (pdevs\text{-val } e \ x)$   
**by** (*auto simp add: pdevs-val-pdevs-domain f.sum f.scaleR*)  
**finally show** *?thesis* .  
**qed**

## 2.12 Pointwise Scaling of Partial Deviations

**definition**  $scaleR\text{-pdevs}::real \Rightarrow 'a::real\text{-vector} \ pdevs \Rightarrow 'a \ pdevs$   
**where**  $scaleR\text{-pdevs } r \ x = unop\text{-pdevs } ((*_R) \ r) \ x$

**lemma**  $pdevs\text{-apply-scaleR-pdevs[simp]}$ :  
 $pdevs\text{-apply } (scaleR\text{-pdevs } x \ Y) \ n = x \ *_R \ pdevs\text{-apply } Y \ n$   
**by** (*auto simp: scaleR-pdevs-def*)

**lemma**  $degree\text{-scaleR-pdevs[simp]}$ :  $degree \ (scaleR\text{-pdevs } r \ x) = (if \ r = 0 \ then \ 0 \ else \ degree \ x)$   
**unfolding**  $degree\text{-def}$   
**by** *auto*

**lemma**  $pdevs\text{-val-scaleR-pdevs[simp]}$ :  
**fixes**  $x::real$  **and**  $Y::'a::real\text{-normed-vector} \ pdevs$   
**shows**  $pdevs\text{-val } e \ (scaleR\text{-pdevs } x \ Y) = x \ *_R \ pdevs\text{-val } e \ Y$   
**by** (*auto simp: pdevs-val-sum scaleR-sum-right ac-simps*)

## 2.13 Partial Deviations Scale Pointwise

**definition**  $pdevs\text{-scaleR}::real \ pdevs \Rightarrow 'a::real\text{-vector} \Rightarrow 'a \ pdevs$   
**where**  $pdevs\text{-scaleR } r \ x = unop\text{-pdevs } (\lambda r. \ r \ *_R \ x) \ r$



**lemma** *pdevs-apply-pdevs-scaleR[simp]*:  
*pdevs-apply (pdevs-scaleR X y) n = pdevs-apply X n \*<sub>R</sub> y*  
**by** (*auto simp: pdevs-scaleR-def*)

**lemma** *degree-pdevs-scaleR[simp]*: *degree (pdevs-scaleR r x) = (if x = 0 then 0 else degree r)*  
**unfolding** *degree-def*  
**by** *auto*

**lemma** *pdevs-val-pdevs-scaleR[simp]*:  
**fixes** *X::real pdevs and y::'a::real-normed-vector*  
**shows** *pdevs-val e (pdevs-scaleR X y) = pdevs-val e X \*<sub>R</sub> y*  
**by** (*auto simp: pdevs-val-sum scaleR-sum-left*)

## 2.14 Pointwise Unary Minus

**definition** *uminus-pdevs::'a::real-vector pdevs ⇒ 'a pdevs*  
**where** *uminus-pdevs = unop-pdevs uminus*

**lemma** *pdevs-apply-uminus-pdevs[simp]*: *pdevs-apply (uminus-pdevs x) = - pdevs-apply x*  
**by** (*auto simp: uminus-pdevs-def*)

**lemma** *degree-uminus-pdevs[simp]*: *degree (uminus-pdevs x) = degree x*  
**by** (*rule degree-cong simp*)

**lemma** *pdevs-val-uminus-pdevs[simp]*: *pdevs-val e (uminus-pdevs x) = - pdevs-val e x*  
**unfolding** *pdevs-val-sum*  
**by** (*auto simp: sum-negf*)

**definition** *uminus-aform X = (- fst X, uminus-pdevs (snd X))*

**lemma** *fst-uminus-aform[simp]*: *fst (uminus-aform Y) = - fst Y*  
**by** (*simp add: uminus-aform-def*)

**lemma** *aform-val-uminus-aform[simp]*: *aform-val e (uminus-aform X) = - aform-val e X*  
**by** (*auto simp: uminus-aform-def aform-val-def*)

## 2.15 Constant

**lift-definition** *zero-pdevs::'a::zero pdevs is λ-. 0* **by** *simp*

**lemma** *pdevs-apply-zero-pdevs[simp]*: *pdevs-apply zero-pdevs i = 0*  
**by** *transfer simp*

**lemma** *pdevs-val-zero-pdevs[simp]*: *pdevs-val e zero-pdevs = 0*  
**by** (*auto simp: pdevs-val-def*)

**definition** *num-aform*  $f = (f, \text{zero-pdevs})$

## 2.16 Inner Product

**definition** *pdevs-inner*:: $'a::\text{euclidean-space } pdevs \Rightarrow 'a \Rightarrow \text{real } pdevs$   
**where** *pdevs-inner*  $x \ b = \text{unop-pdevs } (\lambda x. x \cdot b) \ x$

**lemma** *pdevs-apply-pdevs-inner[simp]*: *pdevs-apply* (*pdevs-inner*  $p \ a$ )  $i = \text{pdevs-apply } p \ i \cdot a$   
**by** (*simp add: pdevs-inner-def*)

**lemma** *pdevs-val-pdevs-inner[simp]*: *pdevs-val*  $e$  (*pdevs-inner*  $p \ a$ ) = *pdevs-val*  $e \ p \cdot a$   
**by** (*auto simp add: inner-sum-left pdevs-val-pdevs-domain intro!: sum.mono-neutral-cong-left*)

**definition** *inner-aform*:: $'a::\text{euclidean-space } aform \Rightarrow 'a \Rightarrow \text{real } aform$   
**where** *inner-aform*  $X \ b = (\text{fst } X \cdot b, \text{pdevs-inner } (\text{snd } X) \ b)$

## 2.17 Inner Product Pair

**definition** *inner2*:: $'a::\text{euclidean-space} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{real*real}$   
**where** *inner2*  $x \ n \ l = (x \cdot n, x \cdot l)$

**definition** *pdevs-inner2*:: $'a::\text{euclidean-space } pdevs \Rightarrow 'a \Rightarrow 'a \Rightarrow (\text{real*real}) \ pdevs$   
**where** *pdevs-inner2*  $X \ n \ l = \text{unop-pdevs } (\lambda x. \text{inner2 } x \ n \ l) \ X$

**lemma** *pdevs-apply-pdevs-inner2[simp]*: *pdevs-apply* (*pdevs-inner2*  $p \ a \ b$ )  $i = (\text{pdevs-apply } p \ i \cdot a, \text{pdevs-apply } p \ i \cdot b)$   
**by** (*simp add: pdevs-inner2-def inner2-def zero-prod-def*)

**definition** *inner2-aform*:: $'a::\text{euclidean-space } aform \Rightarrow 'a \Rightarrow 'a \Rightarrow (\text{real*real}) \ aform$   
**where** *inner2-aform*  $X \ a \ b = (\text{inner2 } (\text{fst } X) \ a \ b, \text{pdevs-inner2 } (\text{snd } X) \ a \ b)$

**lemma** *linear-inner2[intro, simp]*: *linear*  $(\lambda x. \text{inner2 } x \ n \ i)$   
**by** *unfold-locales (auto simp: inner2-def algebra-simps)*

**lemma** *aform-val-inner2-aform[simp]*: *aform-val*  $e$  (*inner2-aform*  $Z \ n \ i$ ) = *inner2* (*aform-val*  $e \ Z$ )  $n \ i$

**proof** –

**have** *aform-val*  $e$  (*inner2-aform*  $Z \ n \ i$ ) = *inner2* (*fst*  $Z$ )  $n \ i$  + *inner2* (*pdevs-val*  $e$  (*snd*  $Z$ ))  $n \ i$

**by** (*auto simp: aform-val-def inner2-aform-def pdevs-inner2-def pdevs-val-unop-linear*)

**also have**  $\dots = \text{inner2 } (\text{aform-val } e \ Z) \ n \ i$

**by** (*simp add: inner2-def algebra-simps aform-val-def*)

**finally show** *thesis* .

**qed**

## 2.18 Update

**lemma** *pdevs-val-upd[simp]*:

$pdevs\text{-}val\ (e(n := e'))\ X = pdevs\text{-}val\ e\ X - e\ n * pdevs\text{-}apply\ X\ n + e' * pdevs\text{-}apply\ X\ n$   
**unfolding**  $pdevs\text{-}val\text{-}def$   
**by** ( $subst\ suminf\text{-}finite[OF\ finite.insertI[OF\ finite\text{-}degree\text{-}nonzero],\ of\ n\ X]$ ,  
 $auto\ simp: pdevs\text{-}val\text{-}def\ sum.insert\text{-}remove$ ) $+$

**lemma**  $nonzeros\text{-}fun\text{-}upd$ :  
 $\{i. (f(n := a))\ i \neq 0\} \subseteq \{i. f\ i \neq 0\} \cup \{n\}$   
**by** ( $auto\ split: if\text{-}split\text{-}asm$ )

**lift\text{-}definition**  $pdev\text{-}upd::'a::real\text{-}vector\ pdevs \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ pdevs$   
**is**  $\lambda x\ n\ a. x(n:=a)$   
**by** ( $rule\ finite\text{-}subset[OF\ nonzeros\text{-}fun\text{-}upd]$ )  $simp$

**lemma**  $pdevs\text{-}apply\text{-}pdev\text{-}upd[simp]$ :  
 $pdevs\text{-}apply\ (pdev\text{-}upd\ X\ n\ x) = (pdevs\text{-}apply\ X)(n:=x)$   
**by**  $transfer\ simp$

**lemma**  $pdevs\text{-}val\text{-}pdev\text{-}upd[simp]$ :  
 $pdevs\text{-}val\ e\ (pdev\text{-}upd\ X\ n\ x) = pdevs\text{-}val\ e\ X + e\ n *_{\mathbb{R}}\ x - e\ n *_{\mathbb{R}}\ pdevs\text{-}apply\ X\ n$   
**unfolding**  $pdevs\text{-}val\text{-}def$   
**by** ( $subst\ suminf\text{-}finite[OF\ finite.insertI[OF\ finite\text{-}degree\text{-}nonzero],\ of\ n\ X]$ ,  
 $auto\ simp: pdevs\text{-}val\text{-}def\ sum.insert\text{-}remove$ ) $+$

**lemma**  $degree\text{-}pdev\text{-}upd$ :  
**assumes**  $x = 0 \longleftrightarrow pdevs\text{-}apply\ X\ n = 0$   
**shows**  $degree\ (pdev\text{-}upd\ X\ n\ x) = degree\ X$   
**using**  $assms$   
**by** ( $auto\ intro!: degree\text{-}cong\ split: if\text{-}split\text{-}asm$ )

**lemma**  $degree\text{-}pdev\text{-}upd\text{-}le$ :  
**assumes**  $degree\ X \leq n$   
**shows**  $degree\ (pdev\text{-}upd\ X\ n\ x) \leq Suc\ n$   
**using**  $assms$   
**by** ( $auto\ intro!: degree\text{-}le$ )

## 2.19 Inf/Sup

**definition**  $Inf\text{-}aform\ X = fst\ X - tdev\ (snd\ X)$

**definition**  $Sup\text{-}aform\ X = fst\ X + tdev\ (snd\ X)$

**lemma**  $Inf\text{-}aform$ :  
**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows**  $Inf\text{-}aform\ X \leq aform\text{-}val\ e\ X$   
**using**  $order\text{-}trans[OF\ abs\text{-}ge\text{-}minus\text{-}self\ abs\text{-}pdevs\text{-}val\text{-}le\text{-}tdev[OF\ assms]]$   
**by** ( $auto\ simp: Inf\text{-}aform\text{-}def\ aform\text{-}val\text{-}def\ minus\text{-}le\text{-}iff$ )

**lemma** *Sup-aform*:  
**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows**  $aform\text{-}val\ e\ X \leq Sup\text{-}aform\ X$   
**using**  $order\text{-}trans[OF\ abs\text{-}ge\text{-}self\ abs\text{-}pdevs\text{-}val\text{-}le\text{-}tdev[OF\ assms]]$   
**by** (*auto simp: Sup-aform-def aform-val-def*)

## 2.20 Minkowski Sum

**definition**  $msum\text{-}pdevs\text{-}raw::nat \Rightarrow (nat \Rightarrow 'a::real\text{-}vector) \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$   
**where**

$msum\text{-}pdevs\text{-}raw\ n\ x\ y\ i = (if\ i < n\ then\ x\ i\ else\ y\ (i - n))$

**lemma** *nonzeros-msum-pdevs-raw*:  
 $\{i. msum\text{-}pdevs\text{-}raw\ n\ f\ g\ i \neq 0\} = (\{0..<n\} \cap \{i. f\ i \neq 0\}) \cup (+)\ n\ '(\{i. g\ i \neq 0\})$   
**by** (*force simp: msum-pdevs-raw-def not-less split: if-split-asm*)

**lift-definition**  $msum\text{-}pdevs::nat \Rightarrow 'a::real\text{-}vector\ pdevs \Rightarrow 'a\ pdevs \Rightarrow 'a\ pdevs$  **is**  $msum\text{-}pdevs\text{-}raw$   
**unfolding** *nonzeros-msum-pdevs-raw* **by** *simp*

**lemma** *pdevs-apply-msum-pdevs*:  $pdevs\text{-}apply\ (msum\text{-}pdevs\ n\ f\ g)\ i =$   
*(if*  $i < n$  *then*  $pdevs\text{-}apply\ f\ i$  *else*  $pdevs\text{-}apply\ g\ (i - n)$ *)*  
**by** *transfer (auto simp: msum-pdevs-raw-def)*

**lemma** *degree-least-nonzero*:  
**assumes**  $degree\ f \neq 0$   
**shows**  $pdevs\text{-}apply\ f\ (degree\ f - 1) \neq 0$   
**proof**  
**assume**  $H: pdevs\text{-}apply\ f\ (degree\ f - 1) = 0$   
 $\{$   
  **fix**  $j$   
  **assume**  $j \geq degree\ f - 1$   
  **with**  $H$  **have**  $pdevs\text{-}apply\ f\ j = 0$   
  **by** (*cases*  $degree\ f - 1 = j$ ) *auto*  
 $\}$   
**from** *degree-le[rule-format, OF this]*  
**have**  $degree\ f \leq degree\ f - 1$   
  **by** *blast*  
**with** *assms* **show** *False* **by** *simp*  
**qed**

**lemma** *degree-leI*:  
**assumes**  $(\bigwedge i. pdevs\text{-}apply\ y\ i = 0 \implies pdevs\text{-}apply\ x\ i = 0)$   
**shows**  $degree\ x \leq degree\ y$   
**proof** *cases*  
**assume**  $degree\ x \neq 0$   
**from** *degree-least-nonzero[OF this]*  
**have**  $pdevs\text{-}apply\ y\ (degree\ x - 1) \neq 0$   
  **by** (*auto simp: assms split: if-split-asm*)

**from** *degree-gt*[*OF this*] **show** *?thesis*  
**by** *simp*  
**qed** *simp*

**lemma** *degree-msum-pdevs-ge1*:  
**shows**  $\text{degree } f \leq n \implies \text{degree } f \leq \text{degree } (\text{msum-pdevs } n \ f \ g)$   
**by** (*rule degree-leI*) (*auto simp: pdevs-apply-msum-pdevs split: if-split-asm*)

**lemma** *degree-msum-pdevs-ge2*:  
**assumes**  $\text{degree } f \leq n$   
**shows**  $\text{degree } g \leq \text{degree } (\text{msum-pdevs } n \ f \ g) - n$   
**proof** *cases*

**assume**  $\text{degree } g \neq 0$   
**hence**  $\text{pdevs-apply } g \ (\text{degree } g - 1) \neq 0$  **by** (*rule degree-least-nonzero*)  
**hence**  $\text{pdevs-apply } (\text{msum-pdevs } n \ f \ g) \ (n + \text{degree } g - 1) \neq 0$   
**using** *assms*  
**by** (*auto simp: pdevs-apply-msum-pdevs*)  
**from** *degree-gt*[*OF this*]  
**show** *?thesis*  
**by** *simp*  
**qed** *simp*

**lemma** *degree-msum-pdevs-le*:  
**shows**  $\text{degree } (\text{msum-pdevs } n \ f \ g) \leq n + \text{degree } g$   
**by** (*auto intro!: degree-le simp: pdevs-apply-msum-pdevs*)

**lemma**  
*sum-msum-pdevs-cases*:  
**assumes**  $\text{degree } f \leq n$   
**assumes** [*simp*]:  $\bigwedge i. e \ i \ 0 = 0$   
**shows**  
 $(\sum i < \text{degree } (\text{msum-pdevs } n \ f \ g). e \ i \ (\text{if } i < n \ \text{then } \text{pdevs-apply } f \ i \ \text{else } \text{pdevs-apply } g \ (i - n))) =$   
 $(\sum i < \text{degree } f. e \ i \ (\text{pdevs-apply } f \ i)) + (\sum i < \text{degree } g. e \ (i + n) \ (\text{pdevs-apply } g \ i))$   
**(is ?lhs = ?rhs)**  
**proof** -  
**have**  $?lhs = (\sum i \in \{.. < \text{degree } (\text{msum-pdevs } n \ f \ g)\} \cap \{i. i < n\}. e \ i \ (\text{pdevs-apply } f \ i)) +$   
 $(\sum i \in \{.. < \text{degree } (\text{msum-pdevs } n \ f \ g)\} \cap - \{i. i < n\}. e \ i \ (\text{pdevs-apply } g \ (i - n)))$   
**(is - = ?sum-f + ?sum-g)**  
**by** (*simp add: sum.If-cases if-distrib*)  
**also have**  $?sum-f = (\sum i = 0.. < \text{degree } f. e \ i \ (\text{pdevs-apply } f \ i))$   
**using** *assms degree-msum-pdevs-ge1*[*of f n g*]  
**by** (*intro sum.mono-neutral-cong-right*) *auto*  
**also**  
**have**  $?sum-g = (\sum i \in \{0 + n.. < \text{degree } (\text{msum-pdevs } n \ f \ g) - n + n\}. e \ i \ (\text{pdevs-apply } g \ (i - n)))$

by (rule sum.cong) auto  
 also have ... =  $(\sum i = 0..<degree (msum-pdevs\ n\ f\ g) - n. e\ (i + n) (pdevs-apply\ g\ (i + n - n)))$   
 by (rule sum.shift-bounds-nat-ivl)  
 also have ... =  $(\sum i = 0..<degree\ g. e\ (i + n) (pdevs-apply\ g\ i))$   
 using assms degree-msum-pdevs-ge2[of f n]  
 by (intro sum.mono-neutral-cong-right) (auto intro!: sum.mono-neutral-cong-right)  
 finally show ?thesis  
 by (simp add: atLeast0LessThan)  
 qed

**lemma** tdev-msum-pdevs:  $degree\ f \leq n \implies tdev\ (msum-pdevs\ n\ f\ g) = tdev\ f + tdev\ g$   
 by (auto simp: tdev-def pdevs-apply-msum-pdevs intro!: sum-msum-pdevs-cases)

**lemma** pdevs-val-msum-pdevs:  
 $degree\ f \leq n \implies pdevs-val\ e\ (msum-pdevs\ n\ f\ g) = pdevs-val\ e\ f + pdevs-val\ (\lambda i. e\ (i + n))\ g$   
 by (auto simp: pdevs-val-sum pdevs-apply-msum-pdevs intro!: sum-msum-pdevs-cases)

**definition** msum-aform::nat  $\Rightarrow$  'a::real-vector aform  $\Rightarrow$  'a aform  $\Rightarrow$  'a aform  
 where msum-aform n f g = (fst f + fst g, msum-pdevs n (snd f) (snd g))

**lemma** fst-msum-aform[simp]:  $fst\ (msum-aform\ n\ f\ g) = fst\ f + fst\ g$   
 by (simp add: msum-aform-def)

**lemma** snd-msum-aform[simp]:  $snd\ (msum-aform\ n\ f\ g) = msum-pdevs\ n\ (snd\ f)\ (snd\ g)$   
 by (simp add: msum-aform-def)

**lemma** finite-nonzero-summable:  $finite\ \{i. f\ i \neq 0\} \implies summable\ f$   
 by (auto intro!: sums-summable sums-finite)

**lemma** aform-val-msum-aform:  
 assumes degree-aform f  $\leq n$   
 shows aform-val e (msum-aform n f g) = aform-val e f + aform-val ( $\lambda i. e\ (i + n)$ ) g  
 using assms  
 by (auto simp: pdevs-val-msum-pdevs aform-val-def)

**lemma** Inf-aform-msum-aform:  
 $degree-aform\ X \leq n \implies Inf-aform\ (msum-aform\ n\ X\ Y) = Inf-aform\ X + Inf-aform\ Y$   
 by (simp add: Inf-aform-def tdev-msum-pdevs)

**lemma** Sup-aform-msum-aform:  
 $degree-aform\ X \leq n \implies Sup-aform\ (msum-aform\ n\ X\ Y) = Sup-aform\ X + Sup-aform\ Y$   
 by (simp add: Sup-aform-def tdev-msum-pdevs)

**definition** *independent-from*  $d Y = \text{msum-aform } d (0, \text{zero-pdevs}) Y$

**definition** *independent-aform*  $X Y = \text{independent-from } (\text{degree-aform } X) Y$

**lemma** *degree-zero-pdevs[simp]*:  $\text{degree zero-pdevs} = 0$   
**by** (*metis degree-least-nonzero pdevs-apply-zero-pdevs*)

**lemma** *independent-aform-Joints*:

**assumes**  $x \in \text{Affine } X$   
**assumes**  $y \in \text{Affine } Y$   
**shows**  $[x, y] \in \text{Joints } [X, \text{independent-aform } X Y]$   
**using** *assms*  
**unfolding** *Affine-def valuate-def Joints-def*  
**apply** *safe*  
**subgoal premises** *prems for e ea*  
**using** *prems*  
**by** (*intro image-eqI[where x= $\lambda i$ . if  $i < \text{degree-aform } X$  then  $e i$  else  $ea (i - \text{degree-aform } X)$ ]*)  
*(auto simp: aform-val-def pdevs-val-msum-pdevs Pi-iff*  
*independent-aform-def independent-from-def intro!: pdevs-val-degree-cong)*  
**done**

**lemma** *msum-aform-Joints*:

**assumes**  $d \geq \text{degree-aform } X$   
**assumes**  $\bigwedge X. X \in \text{set } XS \implies d \geq \text{degree-aform } X$   
**assumes**  $(x \# xs) \in \text{Joints } (X \# XS)$   
**assumes**  $y \in \text{Affine } Y$   
**shows**  $((x + y) \# x \# xs) \in \text{Joints } (\text{msum-aform } d X Y \# X \# XS)$   
**using** *assms*  
**unfolding** *Joints-def valuate-def Affine-def*  
**proof** (*safe, goal-cases*)  
**case** ( $1 e ea a b zs$ )  
**then show** *?case*  
**by** (*intro image-eqI[where x =  $\lambda i$ . if  $i < d$  then  $e i$  else  $ea (i - d)$ ]*)  
*(force simp: aform-val-def pdevs-val-msum-pdevs intro!: intro!: pdevs-val-degree-cong)*  
**qed**

**lemma** *Joints-msum-aform*:

**assumes**  $d \geq \text{degree-aform } X$   
**assumes**  $\bigwedge Y. Y \in \text{set } YS \implies d \geq \text{degree-aform } Y$   
**shows**  $\text{Joints } (\text{msum-aform } d X Y \# YS) = \{((x + y) \# ys) \mid x y ys. y \in \text{Affine } Y$   
 $\wedge x \# ys \in \text{Joints } (X \# YS)\}$   
**unfolding** *Affine-def valuate-def Joints-def*  
**proof** (*safe, goal-cases*)  
**case** ( $1 x e$ )  
**thus** *?case*  
**using** *assms*  
**by** (*intro exI[where x = aform-val e X] exI[where x = aform-val (( $\lambda i$ . e (i*

+ d))) Y])  
 (auto simp add: aform-val-def pdevs-val-msum-pdevs)  
**next**  
**case** (2 x xa y ys e ea)  
**thus** ?case **using** assms  
**by** (intro image-eqI[**where** x=λi. if i < d then ea i else e (i - d)])  
 (force simp: aform-val-def pdevs-val-msum-pdevs Pi-iff intro!: pdevs-val-degree-cong)+  
**qed**

**lemma** Joints-singleton-image: Joints [x] = (λx. [x]) ‘ Affine x  
**by** (auto simp: Joints-def Affine-def valuate-def)

**lemma** Collect-extract-image: {g (f x y) |x y. P x y} = g ‘ {f x y |x y. P x y}  
**by** auto

**lemma** inj-Cons: inj (λx. x#xs)  
**by** (auto intro!: injI)

**lemma** Joints-Nil[simp]: Joints [] = {[]}  
**by** (force simp: Joints-def valuate-def)

**lemma** msum-pdevs-zero-ident[simp]: msum-pdevs 0 zero-pdevs x = x  
**by** transfer (auto simp: msum-pdevs-raw-def)

**lemma** msum-aform-zero-ident[simp]: msum-aform 0 (0, zero-pdevs) x = x  
**by** (simp add: msum-aform-def)

**lemma** mem-Joints-singleton: (x ∈ Joints [X]) = (∃ y. x = [y] ∧ y ∈ Affine X)  
**by** (auto simp: Affine-def valuate-def Joints-def)

**lemma** singleton-mem-Joints[simp]: [x] ∈ Joints [X] ↔ x ∈ Affine X  
**by** (auto simp: mem-Joints-singleton)

**lemma** msum-aform-Joints-without-first:  
**assumes** d ≥ degree-aform X  
**assumes** ∧ X. X ∈ set XS ⇒ d ≥ degree-aform X  
**assumes** (x#xs) ∈ Joints (X#XS)  
**assumes** y ∈ Affine Y  
**assumes** z = x + y  
**shows** z#xs ∈ Joints (msum-aform d X Y#XS)  
**unfolding** ⟨z = x + y⟩  
**using** msum-aform-Joints[OF assms(1-4)]  
**by** (force simp: Joints-def valuate-def)

**lemma** Affine-msum-aform:  
**assumes** d ≥ degree-aform X  
**shows** Affine (msum-aform d X Y) = {x + y |x y. x ∈ Affine X ∧ y ∈ Affine Y}  
**using** Joints-msum-aform[OF assms, of Nil Y, simplified, unfolded mem-Joints-singleton]



**by** (*auto simp add: Joints-singleton-image Collect-extract-image*[**where**  $g=\lambda x.$   
 $[x]$ ]

*inj-image-eq-iff*[*OF inj-Cons*]

**lemma** *Affine-zero-pdevs*[*simp*]:  $Affine (0, zero-pdevs) = \{0\}$

**by** (*force simp: Affine-def valuate-def aform-val-def*)

**lemma** *Affine-independent-aform*:

$Affine (independent-aform X Y) = Affine Y$

**by** (*auto simp: independent-aform-def independent-from-def Affine-msum-aform*)

**lemma**

*abs-diff-eq1*:

**fixes**  $l u :: 'a :: ordered-euclidean-space$

**shows**  $l \leq u \implies |u - l| = u - l$

**by** (*metis abs-of-nonneg diff-add-cancel le-add-same-cancel2*)

**lemma** *compact-sum*:

**fixes**  $f :: 'a \Rightarrow 'b :: topological-space \Rightarrow 'c :: real-normed-vector$

**assumes** *finite I*

**assumes**  $\bigwedge i. i \in I \implies compact (S i)$

**assumes**  $\bigwedge i. i \in I \implies continuous-on (S i) (f i)$

**assumes**  $I \subseteq J$

**shows**  $compact \{\sum_{i \in I}. f i (x i) \mid x. x \in Pi J S\}$

**using** *assms*

**proof** (*induct I*)

**case** *empty*

**thus** *?case*

**proof** (*cases*  $\exists x. x \in Pi J S$ )

**case** *False*

**hence**  $*$ :  $\{\sum_{i \in \{\}}. f i (x i) \mid x. x \in Pi J S\} = \{\}$

**by** (*auto simp: Pi-iff*)

**show** *?thesis unfolding \* by simp*

**qed** *auto*

**next**

**case** (*insert a I*)

**hence**  $\{\sum_{i \in insert a I}. f i (xa i) \mid xa. xa \in Pi J S\}$

$= \{x + y \mid x y. x \in f a ' S a \wedge y \in \{\sum_{i \in I}. f i (x i) \mid x. x \in Pi J S\}\}$

**proof** *safe*

**fix**  $s x$

**assume**  $s \in S a x \in Pi J S$

**thus**  $\exists xa. f a s + (\sum_{i \in I}. f i (x i)) = (\sum_{i \in insert a I}. f i (xa i)) \wedge xa \in Pi J$

$S$

**using** *insert*

**by** (*auto intro!: exI*[**where**  $x=x(a:=s)$ ] *sum.cong*)

**qed** *force*

**also have** *compact ...*

**using** *insert*

**by** (*intro compact-sums*) (*auto intro!: compact-continuous-image*)

**finally show** *?case* .  
**qed**

**lemma** *compact-Affine*:

**fixes**  $X :: 'a :: \text{ordered-euclidean-space}$  *aform*

**shows** *compact* (*Affine*  $X$ )

**proof** –

**have** *Affine*  $X = \{x + y \mid x \ y. x \in \{\text{fst } X\} \wedge$

$y \in \{(\sum i \in \{0..<\text{degree-aform } X\}. e \ i \ *_R \ \text{pdevs-apply } (\text{snd } X) \ i) \mid e. e \in$   
 $UNIV \rightarrow \{-1 .. 1\}\}$

**by** (*auto simp: Affine-def valuate-def aform-val-def pdevs-val-sum atLeast0LessThan*)

**also have** *compact* ...

**by** (*rule compact-sums*) (*auto intro!: compact-sum continuous-intros*)

**finally show** *?thesis* .

**qed**

**lemma** *Joints2-JointsI*:

$(xs, x) \in \text{Joints2 } XS \ X \implies x \# xs \in \text{Joints } (X \# XS)$

**by** (*auto simp: Joints-def Joints2-def valuate-def*)

## 2.21 Splitting

**definition** *split-aform*  $X \ i =$

(*let*  $xi = \text{pdevs-apply } (\text{snd } X) \ i \ /_R \ 2$

*in* ( $\text{fst } X - xi, \text{pdev-upd } (\text{snd } X) \ i \ xi, \text{fst } X + xi, \text{pdev-upd } (\text{snd } X) \ i \ xi$ ))

**lemma** *split-aformE*:

**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$

**assumes**  $x = \text{aform-val } e \ X$

**obtains** *err* **where**  $x = \text{aform-val } (e(i:=err)) (\text{fst } (\text{split-aform } X \ i)) \ \text{err} \in \{-1 .. 1\}$

$\mid \text{err where } x = \text{aform-val } (e(i:=err)) (\text{snd } (\text{split-aform } X \ i)) \ \text{err} \in \{-1 .. 1\}$

**proof** (*atomize-elim*)

**let** *?thesis* =  $(\exists \text{err}. x = \text{aform-val } (e(i := \text{err})) (\text{fst } (\text{split-aform } X \ i)) \wedge \text{err} \in \{-1 .. 1\}) \vee$

$(\exists \text{err}. x = \text{aform-val } (e(i := \text{err})) (\text{snd } (\text{split-aform } X \ i)) \wedge \text{err} \in \{-1 .. 1\})$

{

**assume**  $\text{pdevs-apply } (\text{snd } X) \ i = 0$

**hence**  $X = \text{fst } (\text{split-aform } X \ i)$

**by** (*auto simp: split-aform-def intro!: prod-eqI pdevs-eqI*)

**with** *assms* **have** *?thesis* **by** (*auto intro!: exI[where x=e i]*)

} **moreover** {

**assume**  $\text{pdevs-apply } (\text{snd } X) \ i \neq 0$

**hence** [*simp*]:  $\text{degree-aform } X > i$

**by** (*rule degree-gt*)

**note** *assms*(2)

**also**

**have**  $\text{aform-val } e \ X = \text{fst } X + (\sum i < \text{degree-aform } X. e \ i \ *_R \ \text{pdevs-apply } (\text{snd } X) \ i)$

by (*simp add: aform-val-def pdevs-val-sum*)  
 also  
 have *rewr*:  $\{..<degree\text{-aform } X\} = \{0..<degree\text{-aform } X\} - \{i\} \cup \{i\}$   
 by *auto*  
 have  $(\sum i < degree\text{-aform } X. e\ i *_{R} pdevs\text{-apply } (snd\ X)\ i) =$   
 $(\sum i \in \{0..<degree\text{-aform } X\} - \{i\}. e\ i *_{R} pdevs\text{-apply } (snd\ X)\ i) +$   
 $e\ i *_{R} pdevs\text{-apply } (snd\ X)\ i$   
 by (*subst rewr, subst sum.union-disjoint*) *auto*  
 finally have  $x = fst\ X + \dots$   
 hence  $x = aform\text{-val } (e(i:=2 * e\ i - 1)) (snd\ (split\text{-aform } X\ i))$   
 $x = aform\text{-val } (e(i:=2 * e\ i + 1)) (fst\ (split\text{-aform } X\ i))$   
 by (*auto simp: aform-val-def split-aform-def Let-def pdevs-val-sum atLeast0LessThan*  
*Diff-eq degree-pdev-upd if-distrib sum.If-cases field-simps*  
*scaleR-left-distrib[symmetric]*)  
 moreover  
 have  $2 * e\ i - 1 \in \{-1 .. 1\} \vee 2 * e\ i + 1 \in \{-1 .. 1\}$   
 using *assms* by (*auto simp: not-le Pi-iff dest!: spec[where x=i]*)  
 ultimately have *?thesis* by *blast*  
 } ultimately show *?thesis* by *blast*  
 qed

**lemma** *pdevs-val-add*:  $pdevs\text{-val } (\lambda i. e\ i + f\ i)\ xs = pdevs\text{-val } e\ xs + pdevs\text{-val } f\ xs$   
 by (*auto simp: pdevs-val-pdevs-domain algebra-simps sum.distrib*)

**lemma** *pdevs-val-minus*:  $pdevs\text{-val } (\lambda i. e\ i - f\ i)\ xs = pdevs\text{-val } e\ xs - pdevs\text{-val } f\ xs$   
 by (*auto simp: pdevs-val-pdevs-domain algebra-simps sum-subtractf*)

**lemma** *pdevs-val-cmul*:  $pdevs\text{-val } (\lambda i. u * e\ i)\ xs = u *_{R} pdevs\text{-val } e\ xs$   
 by (*auto simp: pdevs-val-pdevs-domain scaleR-sum-right*)

**lemma** *atLeastAtMost-absI*:  $- a \leq a \implies |x::real| \leq |a| \implies x \in atLeastAtMost\ (-\ a)\ a$   
 by *auto*

**lemma** *divide-atLeastAtMost-1-absI*:  $|x::real| \leq |a| \implies x/a \in \{-1 .. 1\}$   
 by (*intro atLeastAtMost-absI*) (*auto simp: divide-le-eq-1*)

**lemma** *convex-scaleR-aux*:  $u + v = 1 \implies u *_{R} x + v *_{R} x = (x::'a::real\text{-vector})$   
 by (*metis scaleR-add-left scaleR-one*)

**lemma** *convex-mult-aux*:  $u + v = 1 \implies u * x + v * x = (x::real)$   
 using *convex-scaleR-aux[of u v x]* by *simp*

**lemma** *convex-Affine*: *convex* (*Affine*  $X$ )

**proof** (*rule convexI*)

fix  $x\ y::'a$  and  $u\ v::real$

assume  $x \in Affine\ X\ y \in Affine\ X$  and *convex*:  $0 \leq u \leq v\ u + v = 1$

```

then obtain  $e f$  where  $x: x = \text{aform-val } e \ X \ e \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
and  $y: y = \text{aform-val } f \ X \ f \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
by (auto simp: Affine-def valuate-def)
let  $?conv = \lambda i. u * e \ i + v * f \ i$ 
{
  fix  $i$ 
  have  $|?conv \ i| \leq u * |e \ i| + v * |f \ i|$ 
  using convex by (intro order-trans[OF abs-triangle-ineq]) (simp add: abs-mult)
  also have  $\dots \leq 1$ 
  using convex  $x \ y$ 
  by (intro convex-bound-le) (auto simp: Pi-iff abs-real-def)
  finally have  $?conv \ i \leq 1 \ -1 \leq ?conv \ i$ 
  by (auto simp: abs-real-def split: if-split-asm)
}
thus  $u *_{\mathbb{R}} x + v *_{\mathbb{R}} y \in \text{Affine } X$ 
using convex  $x \ y$ 
by (auto simp: Affine-def valuate-def aform-val-def pdevs-val-add pdevs-val-cmul
algebra-simps
convex-scaleR-aux intro!: image-eqI[where x=?conv])
qed

```

```

lemma segment-in-aform-val:
  assumes  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
  assumes  $f \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
  shows closed-segment (aform-val  $e \ X$ ) (aform-val  $f \ X$ )  $\subseteq \text{Affine } X$ 
proof -
  have aform-val  $e \ X \in \text{Affine } X$  aform-val  $f \ X \in \text{Affine } X$ 
  using assms by (auto simp: Affine-def valuate-def)
  with convex-Affine[of X, simplified convex-contains-segment]
  show thesis
  by simp
qed

```

## 2.22 From List of Generators

```

lift-definition pdevs-of-list::'a::zero list  $\Rightarrow$  'a pdevs
is  $\lambda xs \ i. \text{if } i < \text{length } xs \ \text{then } xs \ ! \ i \ \text{else } 0$ 
by auto

```

```

lemma pdevs-apply-pdevs-of-list:
  pdevs-apply (pdevs-of-list  $xs$ )  $i = (\text{if } i < \text{length } xs \ \text{then } xs \ ! \ i \ \text{else } 0)$ 
by transfer simp

```

```

lemma pdevs-apply-pdevs-of-list-Nil[simp]:
  pdevs-apply (pdevs-of-list  $[]$ )  $i = 0$ 
by transfer auto

```

```

lemma pdevs-apply-pdevs-of-list-Cons:
  pdevs-apply (pdevs-of-list ( $x \ \# \ xs$ ))  $i =$ 

```

(if  $i = 0$  then  $x$  else  $pdevs\text{-}apply$  ( $pdevs\text{-}of\text{-}list$   $xs$ ) ( $i - 1$ ))  
**by** *transfer auto*

**lemma**  $pdevs\text{-}domain\text{-}pdevs\text{-}of\text{-}list\text{-}Cons[simp]$ :  $pdevs\text{-}domain$  ( $pdevs\text{-}of\text{-}list$  ( $x \# xs$ )) =  
 (if  $x = 0$  then  $\{\}$  else  $\{0\}$ )  $\cup$   $(+)$   $1$  ‘  $pdevs\text{-}domain$  ( $pdevs\text{-}of\text{-}list$   $xs$ )  
**by** (*force simp: pdevs-apply-pdevs-of-list-Cons split: if-split-asm*)

**lemma**  $pdevs\text{-}val\text{-}pdevs\text{-}of\text{-}list\text{-}eq[simp]$ :  
 $pdevs\text{-}val$   $e$  ( $pdevs\text{-}of\text{-}list$  ( $x \# xs$ )) =  $e$   $0$   $*_R$   $x$  +  $pdevs\text{-}val$  ( $e$   $o$   $(+)$   $1$ ) ( $pdevs\text{-}of\text{-}list$   $xs$ )

**proof** –

**have**  $pdevs\text{-}val$   $e$  ( $pdevs\text{-}of\text{-}list$  ( $x \# xs$ )) =  
 ( $\sum_{i \in pdevs\text{-}domain$  ( $pdevs\text{-}of\text{-}list$  ( $x \# xs$ ))  $\cap$   $\{0\}}$ .  $e$   $i$   $*_R$   $x$ ) +  
 ( $\sum_{i \in pdevs\text{-}domain$  ( $pdevs\text{-}of\text{-}list$  ( $x \# xs$ ))  $\cap$   $-\{0\}}$ .  
 $e$   $i$   $*_R$   $pdevs\text{-}apply$  ( $pdevs\text{-}of\text{-}list$   $xs$ ) ( $i - Suc$   $0$ ))  
 (is  $- = ?l + ?r$ )  
**by** (*simp add: pdevs-val-pdevs-domain if-distrib sum.If-cases pdevs-apply-pdevs-of-list-Cons*)  
**also**  
**have**  $?r = (\sum_{i \in pdevs\text{-}domain$  ( $pdevs\text{-}of\text{-}list$   $xs$ )).  $e$  ( $Suc$   $i$ )  $*_R$   $pdevs\text{-}apply$  ( $pdevs\text{-}of\text{-}list$   $xs$ )  $i$ )  
**by** (*rule sum.reindex-cong[of  $\lambda i. i + 1$ ] auto*)  
**also have**  $\dots = pdevs\text{-}val$  ( $e$   $o$   $(+)$   $1$ ) ( $pdevs\text{-}of\text{-}list$   $xs$ )  
**by** (*simp add: pdevs-val-pdevs-domain*)  
**also have**  $?l = (\sum_{i \in \{0\}}$ .  $e$   $i$   $*_R$   $x$ )  
**by** (*rule sum.mono-neutral-cong-left auto*)  
**also have**  $\dots = e$   $0$   $*_R$   $x$  **by** *simp*  
**finally show**  $?thesis$  .

**qed**

**lemma**

*less-degree-pdevs-of-list-imp-less-length*:

**assumes**  $i < degree$  ( $pdevs\text{-}of\text{-}list$   $xs$ )

**shows**  $i < length$   $xs$

**proof** –

**from** *assms* **have**  $pdevs\text{-}apply$  ( $pdevs\text{-}of\text{-}list$   $xs$ ) ( $degree$  ( $pdevs\text{-}of\text{-}list$   $xs$ )  $- 1$ )  $\neq$   $0$

**by** (*metis degree-least-nonzero less-nat-zero-code*)

**hence**  $degree$  ( $pdevs\text{-}of\text{-}list$   $xs$ )  $- 1 < length$   $xs$

**by** (*simp add: pdevs-apply-pdevs-of-list split: if-split-asm*)

**with** *assms* **show**  $?thesis$

**by** *simp*

**qed**

**lemma**  $tdev\text{-}pdevs\text{-}of\text{-}list[simp]$ :  $tdev$  ( $pdevs\text{-}of\text{-}list$   $xs$ ) =  $sum\text{-}list$  ( $map$   $abs$   $xs$ )

**by** (*auto simp: tdev-def pdevs-apply-pdevs-of-list sum-list-sum-nth*)

*less-degree-pdevs-of-list-imp-less-length*

*intro!*: *sum.mono-neutral-cong-left degree-gt*)

**lemma** *pdevs-of-list-Nil*[simp]: *pdevs-of-list* [] = *zero-pdevs*  
**by** (*auto intro!*: *pdevs-eqI*)

**lemma** *pdevs-val-inj-sumI*:

**fixes** *K*::'a set **and** *g*::'a  $\Rightarrow$  nat

**assumes** *finite K*

**assumes** *inj-on g K*

**assumes** *pdevs-domain x*  $\subseteq$  *g* ' *K*

**assumes**  $\bigwedge i. i \in K \implies g\ i \notin \text{pdevs-domain } x \implies f\ i = 0$

**assumes**  $\bigwedge i. i \in K \implies g\ i \in \text{pdevs-domain } x \implies f\ i = e\ (g\ i) *_R \text{pdevs-apply } x\ (g\ i)$

**shows** *pdevs-val e x* =  $(\sum_{i \in K}. f\ i)$

**proof** –

**have** [simp]: *inj-on* (*the-inv-into K g*) (*pdevs-domain x*)

**using** *assms*

**by** (*auto simp: intro!*: *subset-inj-on*[*OF inj-on-the-inv-into*])

{

**fix** *y* **assume** *y*: *y*  $\in$  *pdevs-domain x*

**have** *g-inv*: *g* (*the-inv-into K g y*) = *y*

**by** (*meson assms*(2) *assms*(3) *y f-the-inv-into-f subset-eq*)

**have** *inv-in*: *the-inv-into K g y*  $\in$  *K*

**by** (*meson assms*(2) *assms*(3) *y subset-iff in-pdevs-domain the-inv-into-into*)

**have** *inv3*: *the-inv-into* (*pdevs-domain x*) (*the-inv-into K g*) (*the-inv-into K g y*) =

*g* (*the-inv-into K g y*)

**using** *assms y*

**by** (*subst the-inv-into-f-f*) (*auto simp: f-the-inv-into-f*[*OF assms*(2)])

**note** *g-inv inv-in inv3*

} **note** *this*[simp]

**have** *pdevs-val e x* =  $(\sum_{i \in \text{pdevs-domain } x}. e\ i *_R \text{pdevs-apply } x\ i)$

**by** (*simp add: pdevs-val-pdevs-domain*)

**also have** ... =  $(\sum_{i \in \text{the-inv-into } K\ g\ ' \text{pdevs-domain } x}. e\ (g\ i) *_R \text{pdevs-apply } x\ (g\ i))$

**by** (*rule sum.reindex-cong*[*OF inj-on-the-inv-into*]) *auto*

**also have** ... =  $(\sum_{i \in K}. f\ i)$

**using** *assms*

**by** (*intro sum.mono-neutral-cong-left*) (*auto simp: the-inv-into-image-eq*)

**finally show** *?thesis* .

**qed**

**lemma** *pdevs-domain-pdevs-of-list-le*: *pdevs-domain* (*pdevs-of-list xs*)  $\subseteq$  {0..*length xs*}

**by** (*auto simp: pdevs-apply-pdevs-of-list split: if-split-asm*)

**lemma** *pdevs-val-zip*: *pdevs-val e* (*pdevs-of-list xs*) =  $(\sum (i,x) \leftarrow \text{zip } [0..<\text{length } xs] \text{xs}. e\ i *_R x)$

**by** (*auto simp: sum-list-distinct-conv-sum-set*

*in-set-zip image-fst-zip pdevs-apply-pdevs-of-list distinct-zipI1*

*intro!*: *pdevs-val-inj-sumI*[*of - fst*])

*split: if-split-asm*)

**lemma** *pdevs-val-map:*

$\langle pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs)$   
 $= (\sum n \leftarrow [0..<length\ xs].\ e\ n\ *_R\ xs\ !\ n)\rangle$

**proof** –

**have**  $\langle map2\ (\lambda i.\ (*_R)\ (e\ i))\ [0..<length\ xs]\ xs =$   
 $map\ (\lambda n.\ e\ n\ *_R\ xs\ !\ n)\ [0..<length\ xs]\rangle$   
**by** (*rule nth-equalityI simp-all*)

**then show** *?thesis*

**by** (*simp add: pdevs-val-zip*)

**qed**

**lemma** *scaleR-sum-list:*

**fixes** *xs::'a::real-vector list*

**shows**  $a\ *_R\ sum\text{-}list\ xs = sum\text{-}list\ (map\ (scaleR\ a)\ xs)$

**by** (*induct xs*) (*auto simp: algebra-simps*)

**lemma** *pdevs-val-const-pdevs-of-list:*  $pdevs\text{-}val\ (\lambda\cdot.\ c)\ (pdevs\text{-}of\text{-}list\ xs) = c\ *_R\ sum\text{-}list\ xs$

**unfolding** *pdevs-val-zip split-beta' scaleR-sum-list*

**by** (*rule arg-cong*) (*auto intro!: nth-equalityI*)

**lemma** *pdevs-val-partition:*

**assumes**  $e \in UNIV \rightarrow I$

**obtains** *f g where*  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) =$

$pdevs\text{-}val\ f\ (pdevs\text{-}of\text{-}list\ (filter\ p\ xs)) +$   
 $pdevs\text{-}val\ g\ (pdevs\text{-}of\text{-}list\ (filter\ (Not\ o\ p)\ xs))$

$f \in UNIV \rightarrow I$

$g \in UNIV \rightarrow I$

**proof** –

**obtain** *i where*  $i \in I$

**by** (*metis assms funcset-mem iso-tuple-UNIV-I*)

**let**  $?zip = zip\ [0..<length\ xs]\ xs$

**define** *part where*  $part = partition\ (p\ o\ snd)\ ?zip$

**let**  $?f =$

$(\lambda n.\ if\ n < degree\ (pdevs\text{-}of\text{-}list\ (filter\ p\ xs))\ then\ e\ (map\ fst\ (fst\ part)\ !\ n)\ else$

*i*)

**let**  $?g =$

$(\lambda n.\ if\ n < degree\ (pdevs\text{-}of\text{-}list\ (filter\ (Not\ o\ p)\ xs))$

$then\ e\ (map\ fst\ (snd\ part)\ !\ n)$

$else\ i$ )

**show** *?thesis*

**proof**

**have**  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) = (\sum (i,x) \leftarrow ?zip.\ e\ i\ *_R\ x)$

**by** (*rule pdevs-val-zip*)

**also have**  $\dots = (\sum (i,x) \in set\ ?zip.\ e\ i\ *_R\ x)$

**by** (*simp add: sum-list-distinct-conv-sum-set distinct-zipI1*)

**also**

```

have [simp]: set (fst part) ∩ set (snd part) = {}
  by (auto simp: part-def)
from partition-set[of p o snd ?zip fst part snd part]
have set ?zip = set (fst part) ∪ set (snd part)
  by (auto simp: part-def)
also have (∑ a∈set (fst part) ∪ set (snd part). case a of (i, x) ⇒ e i *R x) =
  (∑ (i, x)∈set (fst part). e i *R x) + (∑ (i, x)∈set (snd part). e i *R x)
  by (auto simp: split-beta sum-Un)
also
have (∑ (i, x)∈set (fst part). e i *R x) = (∑ (i, x)←(fst part). e i *R x)
  by (simp add: sum-list-distinct-conv-sum-set distinct-zipI1 part-def)
also have ... = (∑ i<length (fst part). case (fst part ! i) of (i, x) ⇒ e i *R
x)
  by (subst sum-list-sum-nth) (simp add: split-beta' atLeast0LessThan)
also have ... =
  pdevs-val (λn. e (map fst (fst part) ! n)) (pdevs-of-list (map snd (fst part)))
  by (force
      simp: pdevs-val-zip sum-list-distinct-conv-sum-set distinct-zipI1 split-beta'
in-set-zip
      intro!:
          sum.reindex-cong[where l=fst] image-eqI[where x = (x, map snd (fst
part) ! x) for x])
  also
have (∑ (i, x)∈set (snd part). e i *R x) = (∑ (i, x)←(snd part). e i *R x)
  by (simp add: sum-list-distinct-conv-sum-set distinct-zipI1 part-def)
also have ... = (∑ i<length (snd part). case (snd part ! i) of (i, x) ⇒ e i *R
x)
  by (subst sum-list-sum-nth) (simp add: split-beta' atLeast0LessThan)
also have ... =
  pdevs-val (λn. e (map fst (snd part) ! n)) (pdevs-of-list (map snd (snd part)))
  by (force simp: pdevs-val-zip sum-list-distinct-conv-sum-set distinct-zipI1
split-beta'
in-set-zip
      intro!: sum.reindex-cong[where l=fst]
          image-eqI[where x = (x, map snd (snd part) ! x) for x])
  also
have pdevs-val (λn. e (map fst (fst part) ! n)) (pdevs-of-list (map snd (fst
part))) =
  pdevs-val (λn.
    if n < degree (pdevs-of-list (map snd (fst part))) then e (map fst (fst part)
! n) else i)
    (pdevs-of-list (map snd (fst part)))
  by (rule pdevs-val-degree-cong) simp-all
  also
have pdevs-val (λn. e (map fst (snd part) ! n)) (pdevs-of-list (map snd (snd
part))) =
  pdevs-val (λn.
    if n < degree (pdevs-of-list (map snd (snd part))) then e (map fst (snd
part) ! n) else i)

```



```

      (pdevs-of-list (map snd (snd part)))
    by (rule pdevs-val-degree-cong) simp-all
  also have map snd (snd part) = filter (Not o p) xs
    by (simp add: part-def filter-map[symmetric] o-assoc)
  also have map snd (fst part) = filter p xs
    by (simp add: part-def filter-map[symmetric])
  finally
  show
    pdevs-val e (pdevs-of-list xs) =
      pdevs-val ?f (pdevs-of-list (filter p xs)) +
      pdevs-val ?g (pdevs-of-list (filter (Not o p) xs)) .
  show ?f ∈ UNIV → I ?g ∈ UNIV → I
    using assms ⟨i∈I⟩
    by (auto simp: Pi-iff)
qed
qed

lemma pdevs-apply-pdevs-of-list-append:
  pdevs-apply (pdevs-of-list (xs @ zs)) i =
    (if i < length xs
     then pdevs-apply (pdevs-of-list xs) i else pdevs-apply (pdevs-of-list zs) (i - length
xs))
  by (auto simp: pdevs-apply-pdevs-of-list nth-append)

lemma degree-pdevs-of-list-le-length[intro, simp]: degree (pdevs-of-list xs) ≤ length
xs
  by (metis less-irreflt-nat le-less-linear less-degree-pdevs-of-list-imp-less-length)

lemma degree-pdevs-of-list-append:
  degree (pdevs-of-list (xs @ ys)) ≤ length xs + degree (pdevs-of-list ys)
  by (rule degree-le) (auto simp: pdevs-apply-pdevs-of-list-append)

lemma pdevs-val-pdevs-of-list-append:
  assumes f ∈ UNIV → I
  assumes g ∈ UNIV → I
  obtains e where
    pdevs-val f (pdevs-of-list xs) + pdevs-val g (pdevs-of-list ys) =
      pdevs-val e (pdevs-of-list (xs @ ys))
    e ∈ UNIV → I
proof
  let ?e = (λi. if i < length xs then f i else g (i - length xs))
  have f: pdevs-val f (pdevs-of-list xs) =
    (∑ i∈{..R pdevs-apply (pdevs-of-list (xs @ ys)) i)
  by (auto simp: pdevs-val-sum degree-gt pdevs-apply-pdevs-of-list-append
intro: sum.mono-neutral-cong-left)
  have g: pdevs-val g (pdevs-of-list ys) =
    (∑ i=length xs ..<length xs + degree (pdevs-of-list ys).
      ?e i *R pdevs-apply (pdevs-of-list (xs @ ys)) i)
  (is - = ?sg)

```

**by** (*auto simp: pdevs-val-sum pdevs-apply-pdevs-of-list-append*  
*intro!: inj-onI image-eqI[where x=length xs + x for x]*  
*sum.reindex-cong[where l= $\lambda i. i - \text{length } xs$ ]*)  
**show**  $\text{pdevs-val } f (\text{pdevs-of-list } xs) + \text{pdevs-val } g (\text{pdevs-of-list } ys) =$   
 $\text{pdevs-val } ?e (\text{pdevs-of-list } (xs @ ys))$   
**unfolding**  $f g$   
**by** (*subst sum.union-disjoint[symmetric]*)  
*(force simp: pdevs-val-sum ivl-disj-un degree-pdevs-of-list-append*  
*intro!: sum.mono-neutral-cong-right*  
*split: if-split-asm)+*  
**show**  $?e \in UNIV \rightarrow I$   
**using** *assms* **by** (*auto simp: Pi-iff*)  
**qed**

**lemma**

*sum-general-mono:*  
**fixes**  $f::'a \Rightarrow ('b::\text{ordered-ab-group-add})$   
**assumes** [*simp,intro*]: *finite s finite t*  
**assumes**  $f: \bigwedge x. x \in s - t \implies f x \leq 0$   
**assumes**  $g: \bigwedge x. x \in t - s \implies g x \geq 0$   
**assumes**  $fg: \bigwedge x. x \in s \cap t \implies f x \leq g x$   
**shows**  $(\sum x \in s. f x) \leq (\sum x \in t. g x)$

**proof** –

**have**  $s = (s - t) \cup (s \cap t)$  **and** [*intro, simp*]:  $(s - t) \cap (s \cap t) = \{\}$  **by** *auto*  
**hence**  $(\sum x \in s. f x) = (\sum x \in s - t \cup s \cap t. f x)$   
**using** *assms* **by** *simp*  
**also have**  $\dots = (\sum x \in s - t. f x) + (\sum x \in s \cap t. f x)$   
**by** (*simp add: sum-Un*)  
**also have**  $(\sum x \in s - t. f x) \leq 0$   
**by** (*auto intro!: sum-nonpos f*)  
**also have**  $0 \leq (\sum x \in t - s. g x)$   
**by** (*auto intro!: sum-nonneg g*)  
**also have**  $(\sum x \in s \cap t. f x) \leq (\sum x \in s \cap t. g x)$   
**by** (*auto intro!: sum-mono fg*)  
**also**  
**have** [*intro, simp*]:  $(t - s) \cap (s \cap t) = \{\}$  **by** *auto*  
**hence**  $\text{sum } g (t - s) + \text{sum } g (s \cap t) = \text{sum } g ((t - s) \cup (s \cap t))$   
**by** (*simp add: sum-Un*)  
**also have**  $\dots = \text{sum } g t$   
**by** (*auto intro!: sum.cong*)  
**finally show**  $?thesis$  **by** *simp*

**qed**

**lemma** *degree-pdevs-of-list-eq'*:

$\langle \text{degree } (\text{pdevs-of-list } xs) = \text{Min } \{n. n \leq \text{length } xs \wedge (\forall m. n \leq m \implies m < \text{length } xs \implies xs ! m = 0)\} \rangle$

**apply** (*simp add: degree-def pdevs-apply-pdevs-of-list*)

**apply** (*rule Least-equality*)

**apply** *auto*

```

apply (subst (asm) Min-le-iff)
apply auto
apply (subst Min-le-iff)
apply auto
apply (metis le-cases not-less)
done

```

**lemma** *pdevs-val-permuted*:

```

⟨pdevs-val (e ∘ p) (pdevs-of-list (permute-list p xs)) = pdevs-val e (pdevs-of-list
xs)⟩ (is ⟨?r = ?s⟩)
if perm: ⟨p permutes {..length xs}⟩
proof –
from that have ⟨image-mset p (mset-set {0..length xs}) = mset-set {0..length
xs}⟩
by (simp add: permutes-image-mset lessThan-atLeast0)
moreover have ⟨map (λn. e (p n) *R permute-list p xs ! n) [0..length xs] =
map (λn. e n *R xs ! n) (map p [0..length xs])⟩
using that by (simp add: permute-list-nth)
ultimately show ?thesis
using that by (simp add: pdevs-apply-pdevs-of-list pdevs-val-map
flip: map-map sum-mset-sum-list)

```

**qed**

**lemma** *pdevs-val-perm-ex*:

```

assumes xs <~~> ys
assumes mem: e ∈ UNIV → I
shows ∃ e'. e' ∈ UNIV → I ∧ pdevs-val e (pdevs-of-list xs) = pdevs-val e'
(pdevs-of-list ys)
proof –
from ⟨mset xs = mset ys⟩
have ⟨mset ys = mset xs⟩ ..
then obtain p where ⟨p permutes {..length xs}⟩ ⟨permute-list p xs = ys⟩
by (rule mset-eq-permutation)
moreover define e' where ⟨e' = e ∘ p⟩
ultimately have ⟨e' ∈ UNIV → I⟩ ⟨pdevs-val e (pdevs-of-list xs) = pdevs-val e'
(pdevs-of-list ys)⟩
using mem by (auto simp add: pdevs-val-permuted)
then show ?thesis by blast

```

**qed**

**lemma** *pdevs-val-perm*:

```

assumes xs <~~> ys
assumes mem: e ∈ UNIV → I
obtains e' where e' ∈ UNIV → I
  pdevs-val e (pdevs-of-list xs) = pdevs-val e' (pdevs-of-list ys)
using assms
by (metis pdevs-val-perm-ex)

```

**lemma** *set-distinct-permI*: set xs = set ys ⇒ distinct xs ⇒ distinct ys ⇒ xs

$\langle \sim \sim \rangle ys$   
**by** (*metis eq-set-perm-remdups remdups-id-iff-distinct*)

**lemmas** *pdevs-val-permute* = *pdevs-val-perm*[*OF set-distinct-permI*]

**lemma** *partition-permI*:  
*filter p xs @ filter (Not o p) xs*  $\langle \sim \sim \rangle xs$   
**by** *simp*

**lemma** *pdevs-val-eqI*:  
**assumes**  $\bigwedge i. i \in pdevs\text{-domain } y \implies i \in pdevs\text{-domain } x \implies$   
 $e\ i *_R\ pdevs\text{-apply } x\ i = f\ i *_R\ pdevs\text{-apply } y\ i$   
**assumes**  $\bigwedge i. i \in pdevs\text{-domain } y \implies i \notin pdevs\text{-domain } x \implies f\ i *_R\ pdevs\text{-apply}$   
 $y\ i = 0$   
**assumes**  $\bigwedge i. i \in pdevs\text{-domain } x \implies i \notin pdevs\text{-domain } y \implies e\ i *_R\ pdevs\text{-apply}$   
 $x\ i = 0$   
**shows** *pdevs-val e x = pdevs-val f y*  
**using** *assms*  
**by** (*force simp: pdevs-val-pdevs-domain*  
*intro!*  
*sum.reindex-bij-witness-not-neutral*[**where**  
*i=id and j = id and*  
*S'=pdevs-domain x - pdevs-domain y and*  
*T'=pdevs-domain y - pdevs-domain x*])

**definition**  
*filter-pdevs-raw*::(*nat*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*nat*  $\Rightarrow$  '*a*::*real-vector*)  $\Rightarrow$  (*nat*  $\Rightarrow$  '*a*)  
**where** *filter-pdevs-raw I X* = ( $\lambda i. \text{if } I\ i\ (X\ i)\ \text{then } X\ i\ \text{else } 0$ )

**lemma** *filter-pdevs-raw-nonzeros*:  $\{i. \text{filter-pdevs-raw } s\ f\ i \neq 0\} = \{i. f\ i \neq 0\} \cap$   
 $\{x. s\ x\ (f\ x)\}$   
**by** (*auto simp: filter-pdevs-raw-def*)

**lift-definition** *filter-pdevs*::(*nat*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  '*a*::*real-vector* *pdevs*  $\Rightarrow$  '*a* *pdevs*  
**is** *filter-pdevs-raw*  
**by** (*simp add: filter-pdevs-raw-nonzeros*)

**lemma** *pdevs-apply-filter-pdevs*[*simp*]:  
*pdevs-apply (filter-pdevs I x) i* = (*if I i (pdevs-apply x i) then pdevs-apply x i else*  
 $0$ )  
**by** *transfer (auto simp: filter-pdevs-raw-def)*

**lemma** *degree-filter-pdevs-le*: *degree (filter-pdevs I x)*  $\leq$  *degree x*  
**by** (*rule degree-leI*) (*simp split: if-split-asm*)

**lemma** *pdevs-val-filter-pdevs*:  
*pdevs-val e (filter-pdevs I x)* =  
 $(\sum i \in \{..<degree\ x\} \cap \{i. I\ i\ (pdevs\text{-apply } x\ i)\}. e\ i *_R\ pdevs\text{-apply } x\ i)$   
**by** (*auto simp: pdevs-val-sum if-distrib sum.inter-restrict degree-filter-pdevs-le*)

*degree-gt*

*intro!*: *sum.mono-neutral-cong-left split: if-split-asm*)

**lemma** *pdevs-val-filter-pdevs-dom*:

*pdevs-val e (filter-pdevs I x) =*

$(\sum i \in \text{pdevs-domain } x \cap \{i. I i (\text{pdevs-apply } x i)\}. e i *_R \text{pdevs-apply } x i)$

**by** (*auto*

*simp: pdevs-val-pdevs-domain if-distrib sum.inter-restrict degree-filter-pdevs-le*

*degree-gt*

*intro!*: *sum.mono-neutral-cong-left split: if-split-asm*)

**lemma** *pdevs-val-filter-pdevs-eval*:

*pdevs-val e (filter-pdevs p x) = pdevs-val ( $\lambda i. \text{if } p i (\text{pdevs-apply } x i) \text{ then } e i \text{ else } 0) x$*

**by** (*auto split: if-split-asm intro!: pdevs-val-eqI*)

**definition** *pdevs-applys X i = map ( $\lambda x. \text{pdevs-apply } x i$ ) X*

**definition** *pdevs-vals e X = map (pdevs-val e) X*

**definition** *aform-vals e X = map (aform-val e) X*

**definition** *filter-pdevs-list I X = map (filter-pdevs ( $\lambda i -. I i (\text{pdevs-applys } X i)$ )) X*

**lemma** *pdevs-applys-filter-pdevs-list[*simp*]*:

*pdevs-applys (filter-pdevs-list I X) i = (if I i (pdevs-applys X i) then pdevs-applys X i else*

*map ( $\lambda -. 0$ ) X)*

**by** (*auto simp: filter-pdevs-list-def o-def pdevs-applys-def*)

**definition** *degrees X = Max (insert 0 (degree ‘ set X))*

**abbreviation** *degree-aforms X  $\equiv$  degrees (map snd X)*

**lemma** *degrees-leI*:

**assumes**  $\bigwedge x. x \in \text{set } X \implies \text{degree } x \leq K$

**shows** *degrees X  $\leq$  K*

**using** *assms*

**by** (*auto simp: degrees-def intro!: Max.boundedI*)

**lemma** *degrees-leD*:

**assumes** *degrees X  $\leq$  K*

**shows**  $\bigwedge x. x \in \text{set } X \implies \text{degree } x \leq K$

**using** *assms*

**by** (*auto simp: degrees-def intro!: Max.boundedI*)

**lemma** *degree-filter-pdevs-list-le: degrees (filter-pdevs-list I x)  $\leq$  degrees x*

**by** (*rule degrees-leI*) (*auto simp: filter-pdevs-list-def intro!: degree-le dest!: degrees-leD*)

**definition** *dense-list-of-pdevs*  $x = \text{map } (\lambda i. \text{pdevs-apply } x \ i) \ [0..<\text{degree } x]$

### 2.22.1 (reverse) ordered coefficients as list

**definition** *list-of-pdevs*  $x =$   
 $\text{map } (\lambda i. (i, \text{pdevs-apply } x \ i)) \ (\text{rev } (\text{sorted-list-of-set } (\text{pdevs-domain } x)))$

**lemma** *list-of-pdevs-zero-pdevs*[*simp*]: *list-of-pdevs zero-pdevs* = []  
**by** (*auto simp: list-of-pdevs-def*)

**lemma** *sum-list-list-of-pdevs*: *sum-list (map snd (list-of-pdevs x))* = *sum-list (dense-list-of-pdevs x)*  
**by** (*auto intro!: sum.mono-neutral-cong-left simp add: degree-gt sum-list-distinct-conv-sum-set dense-list-of-pdevs-def list-of-pdevs-def*)

**lemma** *sum-list-filter-dense-list-of-pdevs*[*symmetric*]:  
*sum-list (map snd (filter (p o snd) (list-of-pdevs x)))* =  
*sum-list (filter p (dense-list-of-pdevs x))*  
**by** (*auto intro!: sum.mono-neutral-cong-left simp add: degree-gt sum-list-distinct-conv-sum-set dense-list-of-pdevs-def list-of-pdevs-def o-def filter-map*)

**lemma** *pdevs-of-list-dense-list-of-pdevs*: *pdevs-of-list (dense-list-of-pdevs x)* =  $x$   
**by** (*auto simp: pdevs-apply-pdevs-of-list dense-list-of-pdevs-def pdevs-eqI*)

**lemma** *pdevs-val-sum-list*: *pdevs-val*  $(\lambda \cdot. c) \ X = c *_{\mathbb{R}} \text{sum-list } (\text{map snd } (\text{list-of-pdevs } X))$   
**by** (*auto simp: pdevs-val-sum sum-list-list-of-pdevs pdevs-val-const-pdevs-of-list*[*symmetric*] *pdevs-of-list-dense-list-of-pdevs*)

**lemma** *list-of-pdevs-all-nonzero*: *list-all*  $(\lambda x. x \neq 0) \ (\text{map snd } (\text{list-of-pdevs } xs))$   
**by** (*auto simp: list-of-pdevs-def list-all-iff*)

**lemma** *list-of-pdevs-nonzero*:  $x \in \text{set } (\text{map snd } (\text{list-of-pdevs } xs)) \implies x \neq 0$   
**by** (*auto simp: list-of-pdevs-def*)

**lemma** *pdevs-of-list-scaleR-0*[*simp*]:  
**fixes**  $xs::'a::\text{real-vector list}$   
**shows** *pdevs-of-list (map ((\*<sub>R</sub>) 0) xs)* = *zero-pdevs*  
**by** (*auto simp: pdevs-apply-pdevs-of-list intro!: pdevs-eqI*)

**lemma** *degree-pdevs-of-list-scaleR*:  
 $\text{degree } (\text{pdevs-of-list } (\text{map } ((*_{\mathbb{R}}) \ c) \ xs)) = (\text{if } c \neq 0 \ \text{then } \text{degree } (\text{pdevs-of-list } xs) \ \text{else } 0)$   
**by** (*auto simp: pdevs-apply-pdevs-of-list intro!: degree-cong*)

**lemma** *list-of-pdevs-eq*:  
 $\text{rev } (\text{list-of-pdevs } X) = (\text{filter } ((\neq) \ 0 \ o \ \text{snd}) \ (\text{map } (\lambda i. (i, \text{pdevs-apply } X \ i)) \ [0..<\text{degree } X]))$

(is - = filter ?P (map ?f ?xs))  
**using** map-filter[of ?f ?P ?xs]  
**by** (auto simp: list-of-pdevs-def o-def sorted-list-of-pdevs-domain-eq rev-map)

**lemma** sum-list-take-pdevs-val-eq:

sum-list (take d xs) = pdevs-val ( $\lambda i. \text{if } i < d \text{ then } 1 \text{ else } 0$ ) (pdevs-of-list xs)

**proof** -

**have** sum-list (take d xs) = 1 \*<sub>R</sub> sum-list (take d xs) **by** simp

**also note** pdevs-val-const-pdevs-of-list[symmetric]

**also have** pdevs-val ( $\lambda \cdot. 1$ ) (pdevs-of-list (take d xs)) =  
pdevs-val ( $\lambda i. \text{if } i < d \text{ then } 1 \text{ else } 0$ ) (pdevs-of-list xs)

**by** (auto simp: pdevs-apply-pdevs-of-list split: if-split-asm intro!: pdevs-val-eqI)

**finally show** ?thesis .

**qed**

**lemma** zero-in-range-pdevs-apply[intro, simp]:

**fixes** X::'a::real-vector pdevs **shows** 0  $\in$  range (pdevs-apply X)

**by** (metis degree-gt less-irrefl rangeI)

**lemma** dense-list-in-range:  $x \in \text{set} (\text{dense-list-of-pdevs } X) \implies x \in \text{range} (\text{pdevs-apply } X)$

**by** (auto simp: dense-list-of-pdevs-def)

**lemma** not-in-dense-list-zeroD:

**assumes** pdevs-apply X  $i \notin \text{set} (\text{dense-list-of-pdevs } X)$

**shows** pdevs-apply X  $i = 0$

**proof** (rule ccontr)

**assume** pdevs-apply X  $i \neq 0$

**hence**  $i < \text{degree } X$

**by** (rule degree-gt)

**thus** False **using** assms

**by** (auto simp: dense-list-of-pdevs-def)

**qed**

**lemma** list-all-list-of-pdevsI:

**assumes**  $\bigwedge i. i \in \text{pdevs-domain } X \implies P (\text{pdevs-apply } X i)$

**shows** list-all ( $\lambda x. P x$ ) (map snd (list-of-pdevs X))

**using** assms **by** (auto simp: list-all-iff list-of-pdevs-def)

**lemma** pdevs-of-list-map-scaleR:

pdevs-of-list (map (scaleR r) xs) = scaleR-pdevs r (pdevs-of-list xs)

**by** (auto intro!: pdevs-eqI simp: pdevs-apply-pdevs-of-list)

**lemma**

map-permI:

**assumes**  $xs < \sim \sim > ys$

**shows** map f xs  $< \sim \sim >$  map f ys

**using** assms **by** induct auto

**lemma** *rev-perm*:  $\text{rev } xs \langle \sim \sim \rangle ys \longleftrightarrow xs \langle \sim \sim \rangle ys$   
**by** *simp*

**lemma** *list-of-pdevs-perm-filter-nonzero*:

$\text{map } \text{snd } (\text{list-of-pdevs } X) \langle \sim \sim \rangle (\text{filter } ((\neq) 0) (\text{dense-list-of-pdevs } X))$

**proof** –

**have** *zip-map*:

$\text{zip } [0..<\text{degree } X] (\text{dense-list-of-pdevs } X) = \text{map } (\lambda i. (i, \text{pdevs-apply } X i))$   
 $[0..<\text{degree } X]$

**by** (*auto simp: dense-list-of-pdevs-def intro!: nth-equalityI*)

**have**  $\text{rev } (\text{list-of-pdevs } X) \langle \sim \sim \rangle$

$\text{filter } ((\neq) 0 \circ \text{snd}) (\text{zip } [0..<\text{degree } X] (\text{dense-list-of-pdevs } X))$

**by** (*auto simp: list-of-pdevs-eq o-def zip-map*)

**from** *map-permI[OF this, of snd]*

**have**  $\text{map } \text{snd } (\text{list-of-pdevs } X) \langle \sim \sim \rangle$

$\text{map } \text{snd } (\text{filter } ((\neq) 0 \circ \text{snd}) (\text{zip } [0..<\text{degree } X] (\text{dense-list-of-pdevs } X)))$

**by** (*simp add: rev-map[symmetric] rev-perm*)

**also have**  $\text{map } \text{snd } (\text{filter } ((\neq) 0 \circ \text{snd}) (\text{zip } [0..<\text{degree } X] (\text{dense-list-of-pdevs } X))) =$

$\text{filter } ((\neq) 0) (\text{dense-list-of-pdevs } X)$

**using** *map-filter[of snd ( $\neq$ ) 0 (zip [0..<degree X] (dense-list-of-pdevs X))]*

**by** (*simp add: o-def dense-list-of-pdevs-def*)

**finally**

**show** *?thesis* .

**qed**

**lemma** *pdevs-val-filter*:

**assumes** *mem*:  $e \in UNIV \rightarrow I$

**assumes**  $0 \in I$

**obtains** *e'* **where**

$\text{pdevs-val } e (\text{pdevs-of-list } (\text{filter } p \text{ } xs)) = \text{pdevs-val } e' (\text{pdevs-of-list } xs)$

$e' \in UNIV \rightarrow I$

**unfolding** *pdevs-val-filter-pdevs-eval*

**proof** –

**have**  $(\lambda-. \text{nat. } 0) \in UNIV \rightarrow I$  **using** *assms* **by** *simp*

**have**  $\text{pdevs-val } e (\text{pdevs-of-list } (\text{filter } p \text{ } xs)) =$

$\text{pdevs-val } e (\text{pdevs-of-list } (\text{filter } p \text{ } xs)) +$

$\text{pdevs-val } (\lambda-. 0) (\text{pdevs-of-list } (\text{filter } (\text{Not } o \text{ } p) \text{ } xs))$

**by** (*simp add: pdevs-val-sum*)

**also**

**from** *pdevs-val-pdevs-of-list-append[OF  $\langle e \in - \rangle \langle (\lambda-. 0) \in - \rangle$ ]*

**obtain** *e'* **where**  $e' \in UNIV \rightarrow I$

$\dots = \text{pdevs-val } e' (\text{pdevs-of-list } (\text{filter } p \text{ } xs @ \text{filter } (\text{Not } o \text{ } p) \text{ } xs))$

**by** *metis*

**note** *this(2)*

**also**

**from** *pdevs-val-perm[OF partition-permI  $\langle e' \in - \rangle$ ]*

**obtain** *e''* **where**  $\dots = \text{pdevs-val } e'' (\text{pdevs-of-list } xs)$   $e'' \in UNIV \rightarrow I$  **by** *metis*

**note** *this(1)*



**finally show** *?thesis* using  $\langle e'' \in \rightarrow \dots \rangle$   
**qed**

**lemma**

*pdevs-val-of-list-of-pdevs:*

**assumes**  $e \in UNIV \rightarrow I$

**assumes**  $0 \in I$

**obtains**  $e'$  **where**

$pdevs\text{-val } e (pdevs\text{-of-list } (map\ snd (list\text{-of-pdevs } X))) = pdevs\text{-val } e' X$   
 $e' \in UNIV \rightarrow I$

**proof** –

**obtain**  $e'$  **where**  $e' \in UNIV \rightarrow I$

**and**  $pdevs\text{-val } e (pdevs\text{-of-list } (map\ snd (list\text{-of-pdevs } X))) =$   
 $pdevs\text{-val } e' (pdevs\text{-of-list } (filter ((\neq) 0) (dense\text{-list-of-pdevs } X)))$

**by** (*rule pdevs-val-perm[OF list-of-pdevs-perm-filter-nonzero assms(1)]*)

**note** *this(2)*

**also from** *pdevs-val-filter[OF  $\langle e' \in \rightarrow \langle 0 \in I \rangle$ , of  $(\neq) 0$  dense-list-of-pdevs X]*

**obtain**  $e''$  **where**  $e'' \in UNIV \rightarrow I$

**and**  $\dots = pdevs\text{-val } e'' (pdevs\text{-of-list } (dense\text{-list-of-pdevs } X))$

**by** *metis*

**note** *this(2)*

**also have**  $\dots = pdevs\text{-val } e'' X$  **by** (*simp add: pdevs-of-list-dense-list-of-pdevs*)

**finally show** *?thesis* using  $\langle e'' \in UNIV \rightarrow I \rangle \dots$

**qed**

**lemma**

*pdevs-val-of-list-of-pdevs2:*

**assumes**  $e \in UNIV \rightarrow I$

**obtains**  $e'$  **where**

$pdevs\text{-val } e X = pdevs\text{-val } e' (pdevs\text{-of-list } (map\ snd (list\text{-of-pdevs } X)))$   
 $e' \in UNIV \rightarrow I$

**proof** –

**from** *list-of-pdevs-perm-filter-nonzero[of X]*

**have** *perm: (filter ((\neq) 0) (dense-list-of-pdevs X))  $\langle \sim \sim \rangle$  map snd (list-of-pdevs X)*

**by** *simp*

**have**  $pdevs\text{-val } e X = pdevs\text{-val } e (pdevs\text{-of-list } (dense\text{-list-of-pdevs } X))$

**by** (*simp add: pdevs-of-list-dense-list-of-pdevs*)

**also from** *pdevs-val-partition[OF  $\langle e \in \rightarrow \rangle$ , of dense-list-of-pdevs X  $(\neq) 0$ ]*

**obtain**  $f g$  **where**  $f \in UNIV \rightarrow I$   $g \in UNIV \rightarrow I$

$\dots = pdevs\text{-val } f (pdevs\text{-of-list } (filter ((\neq) 0) (dense\text{-list-of-pdevs } X))) +$

$pdevs\text{-val } g (pdevs\text{-of-list } (filter (Not \circ (\neq) 0) (dense\text{-list-of-pdevs } X)))$

(**is**  $= ?f + ?g$ )

**by** *metis*

**note** *this(3)*

**also**

**have**  $pdevs\text{-of-list } [x \leftarrow dense\text{-list-of-pdevs } X . x = 0] = zero\text{-pdevs}$

**by** (*auto intro!: pdevs-eqI simp: pdevs-apply-pdevs-of-list dest!: nth-mem*)

**hence**  $?g = 0$  **by** (*auto simp: o-def*)

**also**  
**obtain**  $e'$  **where**  $e' \in UNIV \rightarrow I$   
**and**  $?f = pdevs\text{-}val\ e' (pdevs\text{-}of\text{-}list (map\ snd (list\ of\ pdevs\ X)))$   
**by**  $(rule\ pdevs\text{-}val\text{-}perm[OF\ perm\ \langle f \in \cdot \rangle])$   
**note**  $this(2)$   
**finally show**  $?thesis$  **using**  $\langle e' \in UNIV \rightarrow I \rangle$  **by**  $(auto\ intro!: that)$   
**qed**

**lemma**  $dense\text{-}list\text{-}of\text{-}pdevs\text{-}scaleR$ :  
 $r \neq 0 \implies map\ ((*_R)\ r)\ (dense\text{-}list\text{-}of\text{-}pdevs\ x) = dense\text{-}list\text{-}of\text{-}pdevs\ (scaleR\ pdevs\ r\ x)$   
**by**  $(auto\ simp: dense\text{-}list\text{-}of\text{-}pdevs\text{-}def)$

**lemma**  $degree\text{-}pdevs\text{-}of\text{-}list\text{-}eq$ :  
 $(\bigwedge x. x \in set\ xs \implies x \neq 0) \implies degree\ (pdevs\text{-}of\text{-}list\ xs) = length\ xs$   
**by**  $(cases\ xs)\ (auto\ simp\ add: pdevs\text{-}apply\ pdevs\text{-}of\text{-}list\ nth\ Cons\ intro!: degree\text{-}eqI\ split: nat.\ split)$

**lemma**  $dense\text{-}list\text{-}of\text{-}pdevs\text{-}pdevs\text{-}of\text{-}list$ :  
 $(\bigwedge x. x \in set\ xs \implies x \neq 0) \implies dense\text{-}list\text{-}of\text{-}pdevs\ (pdevs\text{-}of\text{-}list\ xs) = xs$   
**by**  $(auto\ simp: dense\text{-}list\text{-}of\text{-}pdevs\text{-}def\ degree\text{-}pdevs\text{-}of\text{-}list\text{-}eq\ pdevs\text{-}apply\ pdevs\text{-}of\text{-}list\ intro!: nth\ equalityI)$

**lemma**  $pdevs\text{-}of\text{-}list\text{-}sum$ :  
**assumes**  $distinct\ xs$   
**assumes**  $e \in UNIV \rightarrow I$   
**obtains**  $f$  **where**  $f \in UNIV \rightarrow I$   $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) = (\sum P \in set\ xs. f\ P\ *_R\ P)$   
**proof** –  
**define**  $f$  **where**  $f\ X = e\ (the\ (map\ of\ (zip\ xs\ [0..\<length\ xs])\ X))$  **for**  $X$   
**from**  $assms$  **have**  $f \in UNIV \rightarrow I$   
**by**  $(auto\ simp: f\text{-}def)$   
**moreover**  
**have**  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) = (\sum P \in set\ xs. f\ P\ *_R\ P)$   
**by**  $(auto\ simp\ add: pdevs\text{-}val\text{-}zip\ f\text{-}def\ assms\ sum\ list\ distinct\ conv\ sum\ set[symmetric]\ in\ set\ zip\ map\ of\ zip\ upto2\ length\ eq\ nth\ intro!: sum\ list\ nth\ eqI)$   
**ultimately show**  $?thesis ..$   
**qed**

**lemma**  $pdevs\text{-}domain\text{-}eq\ pdevs\text{-}of\text{-}list$ :  
**assumes**  $nz: \bigwedge x. x \in set\ (xs) \implies x \neq 0$   
**shows**  $pdevs\text{-}domain\ (pdevs\text{-}of\text{-}list\ xs) = \{0..\<length\ xs\}$   
**using**  $nz$   
**by**  $(auto\ simp: pdevs\text{-}apply\ pdevs\text{-}of\text{-}list\ split: if\ split\ asm)$

**lemma**  $length\text{-}list\text{-}of\text{-}pdevs\text{-}pdevs\text{-}of\text{-}list$ :  
**assumes**  $nz: \bigwedge x. x \in set\ xs \implies x \neq 0$

**shows**  $\text{length } (\text{list-of-pdevs } (\text{pdevs-of-list } xs)) = \text{length } xs$   
**using**  $\text{nz}$  **by** (*auto simp: list-of-pdevs-def pdevs-domain-eq-pdevs-of-list*)

**lemma** *nth-list-of-pdevs-pdevs-of-list*:  
**assumes**  $\text{nz}: \bigwedge x. x \in \text{set } xs \implies x \neq 0$   
**assumes**  $l: n < \text{length } xs$   
**shows**  $\text{list-of-pdevs } (\text{pdevs-of-list } xs) ! n = ((\text{length } xs - \text{Suc } n), xs ! (\text{length } xs - \text{Suc } n))$   
**using**  $\text{nz } l$   
**by** (*auto simp: list-of-pdevs-def pdevs-domain-eq-pdevs-of-list rev-nth pdevs-apply-pdevs-of-list*)

**lemma** *list-of-pdevs-pdevs-of-list-eq*:  
 $(\bigwedge x. x \in \text{set } xs \implies x \neq 0) \implies$   
 $\text{list-of-pdevs } (\text{pdevs-of-list } xs) = \text{zip } (\text{rev } [0..<\text{length } xs]) (\text{rev } xs)$   
**by** (*auto simp: nth-list-of-pdevs-pdevs-of-list length-list-of-pdevs-pdevs-of-list rev-nth intro!: nth-equalityI*)

**lemma** *sum-list-filter-list-of-pdevs-of-list*:  
**fixes**  $xs::'a::\text{comm-monoid-add list}$   
**assumes**  $\bigwedge x. x \in \text{set } xs \implies x \neq 0$   
**shows**  $\text{sum-list } (\text{filter } p (\text{map } \text{snd } (\text{list-of-pdevs } (\text{pdevs-of-list } xs)))) = \text{sum-list } (\text{filter } p xs)$   
**using** *assms*  
**by** (*auto simp: list-of-pdevs-pdevs-of-list-eq rev-filter[symmetric]*)

**lemma**  
*sum-list-partition*:  
**fixes**  $xs::'a::\text{comm-monoid-add list}$   
**shows**  $\text{sum-list } (\text{filter } p xs) + \text{sum-list } (\text{filter } (\text{Not } o p) xs) = \text{sum-list } xs$   
**by** (*induct xs*) (*auto simp: ac-simps*)

## 2.23 2d zonotopes

**definition** *prod-of-pdevs*  $x y = \text{binop-pdevs Pair } x y$

**lemma** *apply-pdevs-prod-of-pdevs[simp]*:  
 $\text{pdevs-apply } (\text{prod-of-pdevs } x y) i = (\text{pdevs-apply } x i, \text{pdevs-apply } y i)$   
**unfolding** *prod-of-pdevs-def*  
**by** (*simp add: zero-prod-def*)

**lemma** *pdevs-domain-prod-of-pdevs[simp]*:  
 $\text{pdevs-domain } (\text{prod-of-pdevs } x y) = \text{pdevs-domain } x \cup \text{pdevs-domain } y$   
**by** (*auto simp: zero-prod-def*)

**lemma** *pdevs-val-prod-of-pdevs[simp]*:  
 $\text{pdevs-val } e (\text{prod-of-pdevs } x y) = (\text{pdevs-val } e x, \text{pdevs-val } e y)$   
**proof** –  
**have**  $\text{pdevs-val } e x = (\sum_{i \in \text{pdevs-domain } x \cup \text{pdevs-domain } y. e i *_R \text{pdevs-apply } x i}$

**(is - = ?x)**  
**unfolding** *pdevs-val-pdevs-domain*  
**by** (*rule sum.mono-neutral-cong-left*) *auto*  
**moreover have**  $pdevs\text{-val } e \ y = (\sum_{i \in pdevs\text{-domain } x \cup pdevs\text{-domain } y} e \ i \ *R$   
*pdevs-apply y i)*  
**(is - = ?y)**  
**unfolding** *pdevs-val-pdevs-domain*  
**by** (*rule sum.mono-neutral-cong-left*) *auto*  
**ultimately have** (*pdevs-val e x, pdevs-val e y*) = (*?x, ?y*)  
**by** *auto*  
**also have**  $\dots = pdevs\text{-val } e \ (prod\text{-of-pdevs } x \ y)$   
**by** (*simp add: sum-prod pdevs-val-pdevs-domain*)  
**finally show** *?thesis* **by** *simp*  
**qed**

**definition** *prod-of-aforms* (**infixr**  $\langle \times_a \rangle$  80)  
**where** *prod-of-aforms x y* = (*(fst x, fst y), prod-of-pdevs (snd x) (snd y)*)

## 2.24 Intervals

**definition** *One-pdevs-raw::nat  $\Rightarrow$  'a::executable-euclidean-space*  
**where** *One-pdevs-raw i* = (*if i < length (Basis-list::'a list) then Basis-list ! i else 0*)

**lemma** *zeros-One-pdevs-raw:*  
*One-pdevs-raw - ' {0::'a::executable-euclidean-space} = {length (Basis-list::'a list)..}*  
**by** (*auto simp: One-pdevs-raw-def nonzero-Basis split: if-split-asm dest!: nth-mem*)

**lemma** *nonzeros-One-pdevs-raw:*  
*{i. One-pdevs-raw i  $\neq$  (0::'a::executable-euclidean-space)} = - {length (Basis-list::'a list)..}*  
**using** *zeros-One-pdevs-raw*  
**by** *blast*

**lift-definition** *One-pdevs::'a::executable-euclidean-space pdevs is One-pdevs-raw*  
**by** (*auto simp: nonzeros-One-pdevs-raw*)

**lemma** *pdevs-apply-One-pdevs[simp]: pdevs-apply One-pdevs i =*  
*(if i < length (Basis-list::'a::executable-euclidean-space list) then Basis-list ! i else 0::'a)*  
**by** *transfer (simp add: One-pdevs-raw-def)*

**lemma** *Max-Collect-less-nat: Max {i::nat. i < k} = (if k = 0 then Max {} else k - 1)*  
**by** (*auto intro!: Max-eqI*)

**lemma** *degree-One-pdevs[simp]: degree (One-pdevs::'a pdevs) =*  
*length (Basis-list::'a::executable-euclidean-space list)*

**by** (*auto simp: degree-eq-Suc-max Basis-list-nth-nonzero Max-Collect-less-nat intro!: Max-eqI DIM-positive*)

**definition** *inner-scaleR-pdevs::'a::euclidean-space  $\Rightarrow$  'a pdevs  $\Rightarrow$  'a pdevs*  
**where** *inner-scaleR-pdevs b x = unop-pdevs ( $\lambda x. (b \cdot x) *_R x$ ) x*

**lemma** *pdevs-apply-inner-scaleR-pdevs[simp]:*  
*pdevs-apply (inner-scaleR-pdevs a x) i = (a  $\cdot$  (pdevs-apply x i)) \*\_R (pdevs-apply x i)*  
**by** (*simp add: inner-scaleR-pdevs-def*)

**lemma** *degree-inner-scaleR-pdevs-le:*  
*degree (inner-scaleR-pdevs (l::'a::executable-euclidean-space) One-pdevs)  $\leq$*   
*degree (One-pdevs::'a pdevs)*  
**by** (*rule degree-leI (auto simp: inner-scaleR-pdevs-def One-pdevs-raw-def)*)

**definition** *pdevs-of-ivl l u = scaleR-pdevs (1/2) (inner-scaleR-pdevs (u - l) One-pdevs)*

**lemma** *degree-pdevs-of-ivl-le:*  
*degree (pdevs-of-ivl l u::'a::executable-euclidean-space pdevs)  $\leq$  DIM('a)*  
**using** *degree-inner-scaleR-pdevs-le*  
**by** (*simp add: pdevs-of-ivl-def*)

**lemma** *pdevs-apply-pdevs-of-ivl:*  
**defines** *B  $\equiv$  Basis-list::'a::executable-euclidean-space list*  
**shows** *pdevs-apply (pdevs-of-ivl l u) i = (if i < length B then ((u - l)  $\cdot$  (B!i) / 2) \*\_R (B!i) else 0)*  
**by** (*auto simp: pdevs-of-ivl-def B-def*)

**lemma** *deg-length-less-imp[simp]:*  
*k < degree (pdevs-of-ivl l u::'a::executable-euclidean-space pdevs)  $\implies$*   
*k < length (Basis-list::'a list)*  
**by** (*metis (no-types, opaque-lifting) degree-One-pdevs degree-inner-scaleR-pdevs-le degree-scaleR-pdevs dual-order.strict-trans length-Basis-list-pos nat-neq-iff not-le pdevs-of-ivl-def*)

**lemma** *tdev-pdevs-of-ivl: tdev (pdevs-of-ivl l u) = |u - l| /<sub>R</sub> 2*

**proof** –

**have** *tdev (pdevs-of-ivl l u) =*  
*( $\sum i < \text{degree (pdevs-of-ivl l u)}. |pdevs-apply (pdevs-of-ivl l u) i|)$*   
**by** (*auto simp: tdev-def*)  
**also have**  $\dots = (\sum i = 0..<\text{length (Basis-list::'a list)}. |pdevs-apply (pdevs-of-ivl l u) i|)$   
**using** *degree-pdevs-of-ivl-le[of l u]*  
**by** (*intro sum.mono-neutral-cong-left auto*)  
**also have**  $\dots = (\sum i = 0..<\text{length (Basis-list::'a list)}. |((u - l) \cdot \text{Basis-list ! } i / 2) *_R \text{Basis-list ! } i|)$   
**by** (*auto simp: pdevs-apply-pdevs-of-ivl*)  
**also have**  $\dots = (\sum b \leftarrow \text{Basis-list}. |((u - l) \cdot b / 2) *_R b|)$

by (auto simp: sum-list-sum-nth)  
 also have ... =  $(\sum_{b \in \text{Basis}} |((u - l) \cdot b / 2) *_R b|)$   
 by (auto simp: sum-list-distinct-conv-sum-set)  
 also have ... =  $|u - l| /_R 2$   
 by (subst euclidean-representation[symmetric, of  $|u - l| /_R 2$ ])  
 (simp add: abs-inner abs-scaleR)  
 finally show ?thesis .  
 qed

**definition**  $\text{aform-of-ivl } l \ u = ((l + u) /_R 2, \text{pdevs-of-ivl } l \ u)$

**definition**  $\text{aform-of-point } x = \text{aform-of-ivl } x \ x$

**lemma** *Elem-affine-of-ivl-le:*

assumes  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
 assumes  $l \leq u$   
 shows  $l \leq \text{aform-val } e \ (\text{aform-of-ivl } l \ u)$   
**proof** –  
 have  $l = (1 / 2) *_R l + (1 / 2) *_R l$   
 by (simp add: scaleR-left-distrib[symmetric])  
 also have ... =  $(l + u) /_R 2 - \text{tdev } (\text{pdevs-of-ivl } l \ u)$   
 by (auto simp: assms tdev-pdevs-of-ivl algebra-simps)  
 also have ...  $\leq \text{aform-val } e \ (\text{aform-of-ivl } l \ u)$   
 using abs-pdevs-val-le-tdev[OF assms(1), of pdevs-of-ivl l u]  
 by (auto simp: aform-val-def aform-of-ivl-def minus-le-iff dest!: abs-le-D2)  
 finally show ?thesis .  
 qed

**lemma** *Elem-affine-of-ivl-ge:*

assumes  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
 assumes  $l \leq u$   
 shows  $\text{aform-val } e \ (\text{aform-of-ivl } l \ u) \leq u$   
**proof** –  
 have  $\text{aform-val } e \ (\text{aform-of-ivl } l \ u) \leq (l + u) /_R 2 + \text{tdev } (\text{pdevs-of-ivl } l \ u)$   
 using abs-pdevs-val-le-tdev[OF assms(1), of pdevs-of-ivl l u]  
 by (auto simp: aform-val-def aform-of-ivl-def minus-le-iff dest!: abs-le-D1)  
 also have ... =  $(1 / 2) *_R u + (1 / 2) *_R u$   
 by (auto simp: assms tdev-pdevs-of-ivl algebra-simps)  
 also have ... =  $u$   
 by (simp add: scaleR-left-distrib[symmetric])  
 finally show ?thesis .  
 qed

**lemma**

*map-of-zip-upto-length-eq-nth:*  
 assumes  $i < \text{length } B$   
 assumes  $d = \text{length } B$   
 shows  $(\text{map-of } (\text{zip } [0..<d] B) \ i) = \text{Some } (B \ ! \ i)$   
**proof** –

**have**  $\text{length } [0..<\text{length } B] = \text{length } B$   
**by** *simp*  
**from** *map-of-zip-is-Some*[*OF this, of i*] *assms*  
**have** *map-of* (*zip*  $[0..<\text{length } B]$   $B$ )  $i = \text{Some } (B ! i)$   
**by** (*auto simp: in-set-zip*)  
**thus** *?thesis* **by** (*simp add: assms*)  
**qed**

**lemma** *in-ivl-affine-of-ivlE*:

**assumes**  $k \in \{l .. u\}$   
**obtains**  $e$  **where**  $e \in UNIV \rightarrow \{-1 .. 1\}$   $k = \text{aform-val } e (\text{aform-of-ivl } l \ u)$   
**proof** *atomize-elim*  
**define**  $e$  **where** [*abs-def*]:  $e \ i = (\text{let } b = \text{if } i < \text{length } (\text{Basis-list}::'a \ \text{list}) \ \text{then } (\text{the } (\text{map-of } (\text{zip } [0..<\text{length } (\text{Basis-list}::'a \ \text{list}]) (\text{Basis-list}::'a \ \text{list})) \ i)) \ \text{else } 0$   
*in*  
 $((k - (l + u) /_R 2) \cdot b) / (((u - l) /_R 2) \cdot b))$  **for**  $i$   
**let**  $?B = \text{Basis-list}::'a \ \text{list}$   
  
**have**  $k = (1 / 2) *_R (l + u) +$   
 $(\sum b \in \text{Basis}. (\text{if } (u - l) \cdot b = 0 \ \text{then } 0 \ \text{else } ((k - (1 / 2) *_R (l + u)) \cdot b))$   
 $*_R b)$   
**(is - = - + ?dots)**  
**using** *assms*  
**by** (*force simp add: algebra-simps eucl-le*[**where**  $'a='a$ ] *intro!*: *euclidean-eqI*[**where**  $'a='a$ ])  
**also have**  
 $?dots = (\sum b \in \text{Basis}. (\text{if } (u - l) \cdot b = 0 \ \text{then } 0 \ \text{else } ((k - (1 / 2) *_R (l + u)) \cdot b) *_R b))$   
**by** (*auto intro!: sum.cong*)  
**also have**  $\dots = (\sum b \leftarrow ?B. (\text{if } (u - l) \cdot b = 0 \ \text{then } 0 \ \text{else } ((k - (1 / 2) *_R (l + u)) \cdot b) *_R b))$   
**by** (*auto simp: sum-list-distinct-conv-sum-set*)  
**also have**  $\dots =$   
 $(\sum i = 0..<\text{length } ?B.$   
 $(\text{if } (u - l) \cdot ?B ! i = 0 \ \text{then } 0 \ \text{else } ((k - (1 / 2) *_R (l + u)) \cdot ?B ! i) *_R ?B ! i))$   
**by** (*auto simp: sum-list-sum-nth*)  
**also have**  $\dots =$   
 $(\sum i = 0..<\text{degree } (\text{inner-scaleR-pdevs } (u - l) \ \text{One-pdevs}).$   
 $(\text{if } (u - l) \cdot \text{Basis-list} ! i = 0 \ \text{then } 0$   
 $\ \text{else } ((k - (1 / 2) *_R (l + u)) \cdot \text{Basis-list} ! i) *_R \text{Basis-list} ! i))$   
**using** *degree-inner-scaleR-pdevs-le*[*of*  $u - l$ ]  
**by** (*intro sum.mono-neutral-cong-right*) (*auto dest!: degree*)  
**also have**  $(1 / 2) *_R (l + u) +$   
 $(\sum i = 0..<\text{degree } (\text{inner-scaleR-pdevs } (u - l) \ \text{One-pdevs}).$   
 $(\text{if } (u - l) \cdot \text{Basis-list} ! i = 0 \ \text{then } 0$   
 $\ \text{else } ((k - (1 / 2) *_R (l + u)) \cdot \text{Basis-list} ! i) *_R \text{Basis-list} ! i)) =$   
 $\ \text{aform-val } e (\text{aform-of-ivl } l \ u)$   
**using** *degree-inner-scaleR-pdevs-le*[*of*  $u - l$ ]

**by** (*auto simp: aform-val-def aform-of-ivl-def pdevs-of-ivl-def map-of-zip-upto-length-eq-nth*  
*e-def Let-def pdevs-val-sum*  
*intro!: sum.cong*)  
**finally have**  $k = \text{aform-val } e \text{ (aform-of-ivl } l \ u) .$

**moreover**  
**{**  
**fix**  $k \ l \ u :: \text{real}$  **assume**  $l \leq k \ k \leq u$   
**let**  $?m = l / 2 + u / 2$   
**have**  $|k - ?m| \leq |if \ k \leq ?m \ \text{then } ?m - l \ \text{else } u - ?m|$   
**using**  $*$  **by** *auto*  
**also have**  $\dots \leq |u / 2 - l / 2|$   
**by** (*auto simp: abs-real-def*)  
**finally have**  $|k - (l / 2 + u / 2)| \leq |u / 2 - l / 2| .$   
**} note** *midpoint-abs = this*  
**have**  $e \in UNIV \rightarrow \{-1..1\}$   
**using** *assms*  
**unfolding** *e-def Let-def*  
**by** (*intro Pi-I divide-atLeastAtMost-1-absI*)  
*(auto simp: map-of-zip-upto-length-eq-nth eucl-le[where 'a='a]*  
*divide-le-eq-1 not-less inner-Basis algebra-simps intro!: midpoint-abs*  
*dest!: nth-mem)*  
**ultimately show**  $\exists e. e \in UNIV \rightarrow \{-1..1\} \wedge k = \text{aform-val } e \text{ (aform-of-ivl } l$   
 $u)$   
**by** *blast*  
**qed**

**lemma** *Inf-aform-aform-of-ivl:*  
**assumes**  $l \leq u$   
**shows**  $\text{Inf-aform (aform-of-ivl } l \ u) = l$   
**using** *assms*  
**by** (*auto simp: Inf-aform-def aform-of-ivl-def tdev-pdevs-of-ivl abs-diff-eq1 alge-*  
*bra-simps*)  
*(metis field-sum-of-halves scaleR-add-left scaleR-one)*

**lemma** *Sup-aform-aform-of-ivl:*  
**assumes**  $l \leq u$   
**shows**  $\text{Sup-aform (aform-of-ivl } l \ u) = u$   
**using** *assms*  
**by** (*auto simp: Sup-aform-def aform-of-ivl-def tdev-pdevs-of-ivl abs-diff-eq1 alge-*  
*bra-simps*)  
*(metis field-sum-of-halves scaleR-add-left scaleR-one)*

**lemma** *Affine-aform-of-ivl:*  
 $a \leq b \implies \text{Affine (aform-of-ivl } a \ b) = \{a .. b\}$   
**by** (*force simp: Affine-def valuate-def intro!: Elem-affine-of-ivl-ge Elem-affine-of-ivl-le*  
*elim!: in-ivl-affine-of-ivlE*)

**end**



### 3 Operations on Expressions

```
theory Floatarith-Expression
imports
  HOL-Decision-Proc.Approximation
  Affine-Arithmetic-Auxiliarities
  Executable-Euclidean-Space
begin
```

Much of this could move to the distribution...

#### 3.1 Approximating Expression\*s\*

```
unbundle floatarith-syntax
```

```
primrec interpret-floatariths :: floatarith list  $\Rightarrow$  real list  $\Rightarrow$  real list
where
```

```
  interpret-floatariths [] vs = []
  | interpret-floatariths (a#bs) vs = interpret-floatarith a vs#interpret-floatariths
    bs vs
```

```
lemma length-interpret-floatariths[simp]: length (interpret-floatariths fas xs) = length
fas
by (induction fas) auto
```

```
lemma interpret-floatariths-nth[simp]:
  interpret-floatariths fas xs ! n = interpret-floatarith (fas ! n) xs
if n < length fas
using that
by (induction fas arbitrary: n) (auto simp: nth-Cons split: nat.splits)
```

```
abbreviation einterpret  $\equiv$   $\lambda$ fas vs. eucl-of-list (interpret-floatariths fas vs)
```

#### 3.2 Syntax

```
syntax interpret-floatarith::floatarith  $\Rightarrow$  real list  $\Rightarrow$  real
```

```
instantiation floatarith :: {plus, minus, uminus, times, inverse, zero, one}
begin
```

```
definition  $- f = \text{Minus } f$ 
```

```
lemma interpret-floatarith-uminus[simp]:
  interpret-floatarith (- f) xs = - interpret-floatarith f xs
by (auto simp: uminus-floatarith-def)
```

```
definition  $f + g = \text{Add } f g$ 
```

```
lemma interpret-floatarith-plus[simp]:
  interpret-floatarith (f + g) xs = interpret-floatarith f xs + interpret-floatarith g
xs
by (auto simp: plus-floatarith-def)
```

**definition**  $f - g = \text{Add } f \text{ (Minus } g)$   
**lemma** *interpret-floatarith-minus[simp]*:  
 $\text{interpret-floatarith } (f - g) \text{ } xs = \text{interpret-floatarith } f \text{ } xs - \text{interpret-floatarith } g \text{ } xs$   
**by** (*auto simp: minus-floatarith-def*)

**definition**  $\text{inverse } f = \text{Inverse } f$   
**lemma** *interpret-floatarith-inverse[simp]*:  
 $\text{interpret-floatarith } (\text{inverse } f) \text{ } xs = \text{inverse } (\text{interpret-floatarith } f \text{ } xs)$   
**by** (*auto simp: inverse-floatarith-def*)

**definition**  $f * g = \text{Mult } f \text{ } g$   
**lemma** *interpret-floatarith-times[simp]*:  
 $\text{interpret-floatarith } (f * g) \text{ } xs = \text{interpret-floatarith } f \text{ } xs * \text{interpret-floatarith } g \text{ } xs$   
**by** (*auto simp: times-floatarith-def*)

**definition**  $f \text{ div } g = f * \text{Inverse } g$   
**lemma** *interpret-floatarith-divide[simp]*:  
 $\text{interpret-floatarith } (f \text{ div } g) \text{ } xs = \text{interpret-floatarith } f \text{ } xs / \text{interpret-floatarith } g \text{ } xs$   
**by** (*auto simp: divide-floatarith-def inverse-eq-divide*)

**definition**  $1 = \text{Num } 1$   
**lemma** *interpret-floatarith-one[simp]*:  
 $\text{interpret-floatarith } 1 \text{ } xs = 1$   
**by** (*auto simp: one-floatarith-def*)

**definition**  $0 = \text{Num } 0$   
**lemma** *interpret-floatarith-zero[simp]*:  
 $\text{interpret-floatarith } 0 \text{ } xs = 0$   
**by** (*auto simp: zero-floatarith-def*)

**instance proof qed**  
**end**

### 3.3 Derived symbols

**definition**  $R_e \text{ } r = (\text{case quotient-of } r \text{ of } (n, d) \Rightarrow \text{Num } (\text{of-int } n) / \text{Num } (\text{of-int } d))$   
**declare**  $[[\text{coercion } R_e ]]$

**lemma** *interpret-Re[simp]*:  $\text{interpret-floatarith } (R_e \text{ } x) \text{ } xs = \text{real-of-rat } x$   
**by** (*auto simp: R\_e-def of-rat-divide dest!: quotient-of-div split: prod.splits*)

**definition**  $\text{Sin } x = \text{Cos } ((\text{Pi} * (\text{Num } (\text{Float } 1 \text{ } (-1)))) - x)$

**lemma** *interpret-floatarith-Sin[simp]*:  
 $\text{interpret-floatarith } (\text{Sin } x) \text{ } vs = \text{sin } (\text{interpret-floatarith } x \text{ } vs)$

**by** (auto simp: Sin-def approximation-preproc-floatarith(11))

**definition** Half  $x = \text{Mult } (\text{Num } (\text{Float } 1 \ (-1))) \ x$

**lemma** interpret-Half[simp]:  $\text{interpret-floatarith } (\text{Half } x) \ xs = \text{interpret-floatarith } x \ xs / 2$

**by** (auto simp: Half-def)

**definition** Tan  $x = (\text{Sin } x) / (\text{Cos } x)$

**lemma** interpret-floatarith-Tan[simp]:

$\text{interpret-floatarith } (\text{Tan } x) \ vs = \tan (\text{interpret-floatarith } x \ vs)$

**by** (auto simp: Tan-def approximation-preproc-floatarith(12) inverse-eq-divide)

**primrec** Sum<sub>e</sub> **where**

$\text{Sum}_e \ f \ [] = 0$

|  $\text{Sum}_e \ f \ (x\#xs) = f \ x + \text{Sum}_e \ f \ xs$

**lemma** interpret-floatarith-Sum<sub>e</sub>[simp]:

$\text{interpret-floatarith } (\text{Sum}_e \ f \ x) \ vs = (\sum i \leftarrow x. \text{interpret-floatarith } (f \ i) \ vs)$

**by** (induction x) auto

**definition** Norm **where** Norm  $is = \text{Sqrt } (\text{Sum}_e \ (\lambda i. \ i * i) \ is)$

**lemma** interpret-floatarith-norm[simp]:

**assumes** [simp]:  $\text{length } x = \text{DIM } ('a)$

**shows**  $\text{interpret-floatarith } (\text{Norm } x) \ vs = \text{norm } (\text{einterpret } x \ vs :: 'a :: \text{executable-euclidean-space})$

**apply** (auto simp: Norm-def norm-eq-sqrt-inner)

**apply** (subst euclidean-inner[**where** 'a='a])

**apply** (auto simp: power2-eq-square[symmetric] )

**apply** (subst sum-list-Basis-list[symmetric])

**apply** (rule sum-list-nth-eqI)

**by** (auto simp: in-set-zip eucl-of-list-inner)

**notation** floatarith.Power (**infixr**  $\langle \hat{\ }_e \rangle$  80)

### 3.4 Constant Folding

**fun** dest-Num-fa **where**

$\text{dest-Num-fa } (\text{floatarith.Num } x) = \text{Some } x$

|  $\text{dest-Num-fa } \_ = \text{None}$

**fun-cases** dest-Num-fa-None:  $\text{dest-Num-fa } fa = \text{None}$

**and** dest-Num-fa-Some:  $\text{dest-Num-fa } fa = \text{Some } x$

**fun** fold-const-fa **where**

$\text{fold-const-fa } (\text{Add } fa1 \ fa2) =$

$(\text{let } (ffa1, ffa2) = (\text{fold-const-fa } fa1, \text{fold-const-fa } fa2)$

$\text{in case } (\text{dest-Num-fa } ffa1, \text{dest-Num-fa } (ffa2)) \text{ of}$

$(\text{Some } a, \text{Some } b) \Rightarrow \text{Num } (a + b)$

```

| (Some a, None) ⇒ (if a = 0 then ffa2 else Add (Num a) ffa2)
| (None, Some a) ⇒ (if a = 0 then ffa1 else Add ffa1 (Num a))
| (None, None) ⇒ Add ffa1 ffa2)
| fold-const-fa (Minus a) =
  (case (fold-const-fa a) of
    (Num x) ⇒ Num (-x)
  | x ⇒ Minus x)
| fold-const-fa (Mult fa1 fa2) =
  (let (ffa1, ffa2) = (fold-const-fa fa1, fold-const-fa fa2)
  in case (dest-Num-fa ffa1, dest-Num-fa (ffa2)) of
    (Some a, Some b) ⇒ Num (a * b)
  | (Some a, None) ⇒ (if a = 0 then Num 0 else if a = 1 then ffa2 else Mult (Num
a) ffa2)
  | (None, Some a) ⇒ (if a = 0 then Num 0 else if a = 1 then ffa1 else Mult ffa1
(Num a))
  | (None, None) ⇒ Mult ffa1 ffa2)
| fold-const-fa (Inverse a) = Inverse (fold-const-fa a)
| fold-const-fa (Abs a) =
  (case (fold-const-fa a) of
    (Num x) ⇒ Num (abs x)
  | x ⇒ Abs x)
| fold-const-fa (Max a b) =
  (case (fold-const-fa a, fold-const-fa b) of
    (Num x, Num y) ⇒ Num (max x y)
  | (x, y) ⇒ Max x y)
| fold-const-fa (Min a b) =
  (case (fold-const-fa a, fold-const-fa b) of
    (Num x, Num y) ⇒ Num (min x y)
  | (x, y) ⇒ Min x y)
| fold-const-fa (Floor a) =
  (case (fold-const-fa a) of
    (Num x) ⇒ Num (floor-fl x)
  | x ⇒ Floor x)
| fold-const-fa (Power a b) =
  (case (fold-const-fa a) of
    (Num x) ⇒ Num (x ^ b)
  | x ⇒ Power x b)
| fold-const-fa (Cos a) = Cos (fold-const-fa a)
| fold-const-fa (Arctan a) = Arctan (fold-const-fa a)
| fold-const-fa (Sqrt a) = Sqrt (fold-const-fa a)
| fold-const-fa (Exp a) = Exp (fold-const-fa a)
| fold-const-fa (Ln a) = Ln (fold-const-fa a)
| fold-const-fa (Powr a b) = Powr (fold-const-fa a) (fold-const-fa b)
| fold-const-fa Pi = Pi
| fold-const-fa (Var v) = Var v
| fold-const-fa (Num x) = Num x

```

**fun-cases** *fold-const-fa-Num*: *fold-const-fa fa = Num y*  
**and** *fold-const-fa-Add*: *fold-const-fa fa = Add x y*

**and** *fold-const-fa-Minus*: *fold-const-fa fa = Minus y*

**lemma** *fold-const-fa[simp]*: *interpret-floatarith (fold-const-fa fa) xs = interpret-floatarith fa xs*

**by** (*induction fa*) (*auto split!*: *prod.splits floatarith.splits option.splits*  
*elim!*: *dest-Num-fa-None dest-Num-fa-Some*  
*simp*: *max-def min-def floor-fl-def*)

### 3.5 Free Variables

**primrec** *max-Var-floatarith* **where**— TODO: include bound in predicate

*max-Var-floatarith (Add a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-floatarith (Mult a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-floatarith (Inverse a) = max-Var-floatarith a*  
| *max-Var-floatarith (Minus a) = max-Var-floatarith a*  
| *max-Var-floatarith (Num a) = 0*  
| *max-Var-floatarith (Var i) = Suc i*  
| *max-Var-floatarith (Cos a) = max-Var-floatarith a*  
| *max-Var-floatarith (floatarith.Arctan a) = max-Var-floatarith a*  
| *max-Var-floatarith (Abs a) = max-Var-floatarith a*  
| *max-Var-floatarith (floatarith.Max a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-floatarith (floatarith.Min a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-floatarith (floatarith.Pi) = 0*  
| *max-Var-floatarith (Sqrt a) = max-Var-floatarith a*  
| *max-Var-floatarith (Exp a) = max-Var-floatarith a*  
| *max-Var-floatarith (Powr a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-floatarith (floatarith.Ln a) = max-Var-floatarith a*  
| *max-Var-floatarith (Power a n) = max-Var-floatarith a*  
| *max-Var-floatarith (Floor a) = max-Var-floatarith a*

**primrec** *max-Var-floatariths* **where**

*max-Var-floatariths [] = 0*  
| *max-Var-floatariths (x#xs) = max (max-Var-floatarith x) (max-Var-floatariths xs)*

**primrec** *max-Var-form* **where**

*max-Var-form (Conj a b) = max (max-Var-form a) (max-Var-form b)*  
| *max-Var-form (Disj a b) = max (max-Var-form a) (max-Var-form b)*  
| *max-Var-form (Less a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-form (LessEqual a b) = max (max-Var-floatarith a) (max-Var-floatarith b)*  
| *max-Var-form (Bound a b c d) = linorder-class.Max {max-Var-floatarith a, max-Var-floatarith b, max-Var-floatarith c, max-Var-form d}*  
| *max-Var-form (AtLeastAtMost a b c) = linorder-class.Max {max-Var-floatarith*

$a, \text{max-Var-floatarith } b, \text{max-Var-floatarith } c\}$   
 $| \text{max-Var-form } (\text{Assign } a \ b \ c) = \text{linorder-class.Max } \{\text{max-Var-floatarith } a, \text{max-Var-floatarith } b, \text{max-Var-form } c\}$

**lemma**

*interpret-floatarith-eq-take-max-VarI:*

**assumes**  $\text{take } (\text{max-Var-floatarith } ra) \ ys = \text{take } (\text{max-Var-floatarith } ra) \ zs$

**shows**  $\text{interpret-floatarith } ra \ ys = \text{interpret-floatarith } ra \ zs$

**using** *assms*

**by** (*induct ra*) (*auto dest!: take-max-eqD simp: take-Suc-eq split: if-split-asm*)

**lemma**

*interpret-floatariths-eq-take-max-VarI:*

**assumes**  $\text{take } (\text{max-Var-floatariths } ea) \ ys = \text{take } (\text{max-Var-floatariths } ea) \ zs$

**shows**  $\text{interpret-floatariths } ea \ ys = \text{interpret-floatariths } ea \ zs$

**using** *assms*

**apply** (*induction ea*)

**subgoal by** *simp*

**subgoal by** (*clarsimp*) (*metis interpret-floatarith-eq-take-max-VarI take-map take-max-eqD*)

**done**

**lemma** *Max-Image-distrib:*

**includes** *no floatarith-syntax*

**assumes**  $\text{finite } X \ X \neq \{\}$

**shows**  $\text{Max } ((\lambda x. \text{max } (f1 \ x) \ (f2 \ x)) \ ' X) = \text{max } (\text{Max } (f1 \ ' X)) \ (\text{Max } (f2 \ ' X))$

**apply** (*rule Max-eqI*)

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

**subgoal for**  $y$

**using** *assms*

**by** (*force intro: max.coboundedI1 max.coboundedI2 Max-ge*)

**subgoal**

**proof** –

**have**  $\text{Max } (f1 \ ' X) \in f1 \ ' X$  **using** *assms* **by** *auto*

**then obtain**  $x1$  **where**  $x1: x1 \in X \ \text{Max } (f1 \ ' X) = f1 \ x1$  **by** *auto*

**have**  $\text{Max } (f2 \ ' X) \in f2 \ ' X$  **using** *assms* **by** *auto*

**then obtain**  $x2$  **where**  $x2: x2 \in X \ \text{Max } (f2 \ ' X) = f2 \ x2$  **by** *auto*

**show** *?thesis*

**apply** (*rule image-eqI*[**where**  $x = \text{if } f1 \ x1 \leq f2 \ x2 \ \text{then } x2 \ \text{else } x1$ ])

**using**  $x1 \ x2$  *assms*

**apply** (*auto simp: max-def*)

**apply** (*metis Max-ge dual-order.trans finite-imageI image-eqI assms(1)*)

**apply** (*metis Max-ge dual-order.trans finite-imageI image-eqI assms(1)*)

**done**

**qed**

**done**

**lemma** *max-Var-floatarith-simps[simp]:*

$\text{max-Var-floatarith } (a \ / \ b) = \text{max } (\text{max-Var-floatarith } a) \ (\text{max-Var-floatarith } b)$

$\text{max-Var-floatarith } (a * b) = \text{max } (\text{max-Var-floatarith } a) (\text{max-Var-floatarith } b)$   
 $\text{max-Var-floatarith } (a + b) = \text{max } (\text{max-Var-floatarith } a) (\text{max-Var-floatarith } b)$   
 $\text{max-Var-floatarith } (a - b) = \text{max } (\text{max-Var-floatarith } a) (\text{max-Var-floatarith } b)$   
 $\text{max-Var-floatarith } (- b) = (\text{max-Var-floatarith } b)$   
**by** (auto simp: divide-floatarith-def times-floatarith-def plus-floatarith-def minus-floatarith-def  
 uminus-floatarith-def)

**lemma** *max-Var-floatariths-Max*:

$\text{max-Var-floatariths } xs = (\text{if set } xs = \{\} \text{ then } 0 \text{ else linorder-class.Max } (\text{max-Var-floatarith } ' \text{ set } xs))$   
**by** (induct xs) auto

**lemma** *max-Var-floatariths-map-plus[simp]*:

$\text{max-Var-floatariths } (\text{map } (\lambda i. \text{fa1 } i + \text{fa2 } i) \text{ xs}) = \text{max } (\text{max-Var-floatariths } (\text{map } \text{fa1 } \text{ xs})) (\text{max-Var-floatariths } (\text{map } \text{fa2 } \text{ xs}))$   
**by** (auto simp: max-Var-floatariths-Max image-image Max-Image-distrib)

**lemma** *max-Var-floatariths-map-times[simp]*:

$\text{max-Var-floatariths } (\text{map } (\lambda i. \text{fa1 } i * \text{fa2 } i) \text{ xs}) = \text{max } (\text{max-Var-floatariths } (\text{map } \text{fa1 } \text{ xs})) (\text{max-Var-floatariths } (\text{map } \text{fa2 } \text{ xs}))$   
**by** (auto simp: max-Var-floatariths-Max image-image Max-Image-distrib)

**lemma** *max-Var-floatariths-map-divide[simp]*:

$\text{max-Var-floatariths } (\text{map } (\lambda i. \text{fa1 } i / \text{fa2 } i) \text{ xs}) = \text{max } (\text{max-Var-floatariths } (\text{map } \text{fa1 } \text{ xs})) (\text{max-Var-floatariths } (\text{map } \text{fa2 } \text{ xs}))$   
**by** (auto simp: max-Var-floatariths-Max image-image Max-Image-distrib)

**lemma** *max-Var-floatariths-map-uminus[simp]*:

$\text{max-Var-floatariths } (\text{map } (\lambda i. - \text{fa1 } i) \text{ xs}) = \text{max-Var-floatariths } (\text{map } \text{fa1 } \text{ xs})$   
**by** (auto simp: max-Var-floatariths-Max image-image Max-Image-distrib)

**lemma** *max-Var-floatariths-map-const[simp]*:

$\text{max-Var-floatariths } (\text{map } (\lambda i. \text{fa}) \text{ xs}) = (\text{if } \text{xs} = [] \text{ then } 0 \text{ else max-Var-floatarith } \text{fa})$   
**by** (auto simp: max-Var-floatariths-Max image-image image-constant-conv)

**lemma** *max-Var-floatariths-map-minus[simp]*:

$\text{max-Var-floatariths } (\text{map } (\lambda i. \text{fa1 } i - \text{fa2 } i) \text{ xs}) = \text{max } (\text{max-Var-floatariths } (\text{map } \text{fa1 } \text{ xs})) (\text{max-Var-floatariths } (\text{map } \text{fa2 } \text{ xs}))$   
**by** (auto simp: max-Var-floatariths-Max image-image Max-Image-distrib)

**primrec** *fresh-floatarith where*

$\text{fresh-floatarith } (\text{Var } y) \text{ } x \longleftrightarrow (x \neq y)$   
 $\text{fresh-floatarith } (\text{Num } a) \text{ } x \longleftrightarrow \text{True}$   
 $\text{fresh-floatarith } \text{Pi } x \longleftrightarrow \text{True}$   
 $\text{fresh-floatarith } (\text{Cos } a) \text{ } x \longleftrightarrow \text{fresh-floatarith } a \text{ } x$

| *fresh-floatarith* (*Abs a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Arctan a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Sqrt a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Exp a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Floor a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Power a n*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Minus a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Ln a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Inverse a*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x$   
 | *fresh-floatarith* (*Add a b*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x \wedge \text{fresh-floatarith } b \ x$   
 | *fresh-floatarith* (*Mult a b*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x \wedge \text{fresh-floatarith } b \ x$   
 | *fresh-floatarith* (*Max a b*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x \wedge \text{fresh-floatarith } b \ x$   
 | *fresh-floatarith* (*Min a b*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x \wedge \text{fresh-floatarith } b \ x$   
 | *fresh-floatarith* (*Powr a b*)  $x \longleftrightarrow \text{fresh-floatarith } a \ x \wedge \text{fresh-floatarith } b \ x$

**lemma** *fresh-floatarith-subst*:  
 fixes  $v::\text{float}$   
 assumes *fresh-floatarith e x*  
 assumes  $x < \text{length } vs$   
 shows  $\text{interpret-floatarith } e \ (vs[x:=v]) = \text{interpret-floatarith } e \ vs$   
 using *assms*  
 by (*induction e*) (*auto simp: map-update*)

**lemma** *fresh-floatarith-max-Var*:  
 assumes  $\text{max-Var-floatarith } ea \leq i$   
 shows *fresh-floatarith ea i*  
 using *assms*  
 by (*induction ea*) *auto*

**primrec** *fresh-floatariths* **where**  
*fresh-floatariths* []  $x \longleftrightarrow \text{True}$   
 | *fresh-floatariths* ( $a\#as$ )  $x \longleftrightarrow \text{fresh-floatarith } a \ x \wedge \text{fresh-floatariths } as \ x$

**lemma** *fresh-floatariths-max-Var*:  
 assumes  $\text{max-Var-floatariths } ea \leq i$   
 shows *fresh-floatariths ea i*  
 using *assms*  
 by (*induction ea*) (*auto simp: fresh-floatarith-max-Var*)

**lemma**  
*interpret-floatariths-take-eqI*:  
 assumes  $\text{take } n \ ys = \text{take } n \ zs$   
 assumes  $\text{max-Var-floatariths } ea \leq n$   
 shows  $\text{interpret-floatariths } ea \ ys = \text{interpret-floatariths } ea \ zs$   
 by (*rule interpret-floatariths-eq-take-max-VarI*) (*rule take-greater-eqI[OF assms]*)

**lemma**  
*interpret-floatarith-fresh-eqI*:  
 assumes  $\bigwedge i. \text{fresh-floatarith } ea \ i \vee (i < \text{length } ys \wedge i < \text{length } zs \wedge ys ! i = zs$



!  $i$ )  
**shows**  $\text{interpret-floatarith } ea \ ys = \text{interpret-floatarith } ea \ zs$   
**using**  $assms$   
**by**  $(\text{induction } ea) \text{ force+}$

**lemma**  
*interpret-floatariths-fresh-eqI:*  
**assumes**  $\bigwedge i. \text{fresh-floatariths } ea \ i \vee (i < \text{length } ys \wedge i < \text{length } zs \wedge ys ! i = zs$   
!  $i$ )  
**shows**  $\text{interpret-floatariths } ea \ ys = \text{interpret-floatariths } ea \ zs$   
**using**  $assms$   
**apply**  $(\text{induction } ea)$   
**subgoal by**  $(\text{force simp: interpret-floatarith-fresh-eqI intro: interpret-floatarith-fresh-eqI})$   
**subgoal for**  $e \ ea$   
**apply**  $\text{clarsimp}$   
**apply**  $(\text{auto simp: list-eq-iff-nth-eq})$   
**using**  $\text{interpret-floatarith-fresh-eqI by blast}$   
**done**

**lemma**  
*interpret-floatarith-max-Var-cong:*  
**assumes**  $\bigwedge i. i < \text{max-Var-floatarith } f \implies xs ! i = ys ! i$   
**shows**  $\text{interpret-floatarith } f \ ys = \text{interpret-floatarith } f \ xs$   
**using**  $assms$   
**by**  $(\text{induction } f) \text{ auto}$

**lemma**  
*interpret-floatarith-fresh-cong:*  
**assumes**  $\bigwedge i. \neg \text{fresh-floatarith } f \ i \implies xs ! i = ys ! i$   
**shows**  $\text{interpret-floatarith } f \ ys = \text{interpret-floatarith } f \ xs$   
**using**  $assms$   
**by**  $(\text{induction } f) \text{ auto}$

**lemma** *max-Var-floatarith-le-max-Var-floatariths:*  
 $fa \in \text{set } fas \implies \text{max-Var-floatarith } fa \leq \text{max-Var-floatariths } fas$   
**by**  $(\text{induction } fas) (\text{auto simp: nth-Cons max-def split: nat.splits})$

**lemma** *max-Var-floatarith-le-max-Var-floatariths-nth:*  
 $n < \text{length } fas \implies \text{max-Var-floatarith } (fas ! n) \leq \text{max-Var-floatariths } fas$   
**by**  $(\text{rule max-Var-floatarith-le-max-Var-floatariths}) \text{ auto}$

**lemma** *max-Var-floatariths-leI:*  
**assumes**  $\bigwedge i. i < \text{length } xs \implies \text{max-Var-floatarith } (xs ! i) \leq F$   
**shows**  $\text{max-Var-floatariths } xs \leq F$   
**using**  $assms$   
**by**  $(\text{auto simp: max-Var-floatariths-Max in-set-conv-nth})$

**lemma** *fresh-floatariths-map-Var[simp]:*  
 $\text{fresh-floatariths } (\text{map floatarith. Var } xs) \ i \longleftrightarrow i \notin \text{set } xs$

by (induction xs) auto

**lemma** *max-Var-floatarith-fold-const-fa*:  
max-Var-floatarith (fold-const-fa fa) ≤ max-Var-floatarith fa  
by (induction fa) (auto simp: fold-const-fa.simps split!: option.splits floatarith.splits)

**lemma** *max-Var-floatariths-fold-const-fa*:  
max-Var-floatariths (map fold-const-fa xs) ≤ max-Var-floatariths xs  
by (auto simp: intro!: max-Var-floatariths-leI max-Var-floatarith-le-max-Var-floatariths-nth  
max-Var-floatarith-fold-const-fa[THEN order-trans])

**lemma** *interpret-form-max-Var-cong*:  
assumes  $\bigwedge i. i < \text{max-Var-form } f \implies xs ! i = ys ! i$   
shows  $\text{interpret-form } f \text{ } xs = \text{interpret-form } f \text{ } ys$   
using *assms*  
by (induction f) (auto simp: interpret-floatarith-max-Var-cong[where xs=xs and  
ys=ys])

**lemma** *max-Var-floatariths-lessI*:  $i < \text{max-Var-floatarith } (fas ! j) \implies j < \text{length } fas \implies i < \text{max-Var-floatariths } fas$   
by (metis leD le-trans less-le max-Var-floatarith-le-max-Var-floatariths nth-mem)

**lemma** *interpret-floatariths-max-Var-cong*:  
assumes  $\bigwedge i. i < \text{max-Var-floatariths } f \implies xs ! i = ys ! i$   
shows  $\text{interpret-floatariths } f \text{ } ys = \text{interpret-floatariths } f \text{ } xs$   
by (auto intro!: nth-equalityI interpret-floatarith-max-Var-cong *assms* max-Var-floatariths-lessI)

**lemma** *max-Var-floatarithimage-Var[simp]*:  $\text{max-Var-floatarith } \text{‘ Var ‘ } X = \text{Suc ‘ } X$   
by force

**lemma** *max-Var-floatariths-map-Var[simp]*:  
 $\text{max-Var-floatariths } (\text{map Var } xs) = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Suc } (\text{linorder-class.Max } (\text{set } xs)))$   
by (auto simp: max-Var-floatariths-Max hom-Max-commute split: if-splits)

**lemma** *Max-atLeastLessThan-nat[simp]*:  $a < b \implies \text{linorder-class.Max } \{a..<b\} = b - 1$  for  $a b :: \text{nat}$   
by (auto intro!: Max-eqI)

### 3.6 Derivatives

**lemma** *isDERIV-Power-iff*:  $\text{isDERIV } j \text{ } (\text{Power } fa \text{ } n) \text{ } xs = (\text{if } n = 0 \text{ then } \text{True} \text{ else } \text{isDERIV } j \text{ } fa \text{ } xs)$   
by (cases n) auto

**lemma** *isDERIV-max-Var-floatarithI*:  
assumes  $\text{isDERIV } n \text{ } f \text{ } ys$

**assumes**  $\bigwedge i. i < \text{max-Var-floatarith } f \implies xs ! i = ys ! i$   
**shows**  $\text{isDERIV } n \ f \ xs$   
**using**  $\text{assms}$   
**proof** ( $\text{induction } f$ )  
**case** ( $\text{Power } f \ n$ ) **then show**  $?case$  **by** ( $\text{cases } n$ )  $\text{auto}$   
**qed** ( $\text{auto simp: max-def interpret-floatarith-max-Var-cong[of - } xs \ ys] \text{ split: if-splits}$ )

**definition**  $\text{isFDERIV}$  **where**  $\text{isFDERIV } n \ xs \ fas \ vs \longleftrightarrow$   
 $(\forall i < n. \forall j < n. \text{isDERIV } (xs ! i) \ (fas ! j) \ vs) \wedge \text{length } fas = n \wedge \text{length } xs = n$

**lemma**  $\text{isFDERIV-I}$ :  $(\bigwedge i \ j. i < n \implies j < n \implies \text{isDERIV } (xs ! i) \ (fas ! j) \ vs)$   
 $\implies$   
 $\text{length } fas = n \implies \text{length } xs = n \implies \text{isFDERIV } n \ xs \ fas \ vs$   
**by** ( $\text{auto simp: isFDERIV-def}$ )

**lemma**  $\text{isFDERIV-isDERIV-D}$ :  $\text{isFDERIV } n \ xs \ fas \ vs \implies i < n \implies j < n \implies$   
 $\text{isDERIV } (xs ! i) \ (fas ! j) \ vs$   
**by** ( $\text{auto simp: isFDERIV-def}$ )

**lemma**  $\text{isFDERIV-lengthD}$ :  $\text{length } fas = n \ \text{length } xs = n$  **if**  $\text{isFDERIV } n \ xs \ fas \ vs$   
**using**  $\text{that}$  **by** ( $\text{auto simp: isFDERIV-def}$ )

**lemma**  $\text{isFDERIV-uptD}$ :  $\text{isFDERIV } n \ [0..<n] \ fas \ vs \implies i < n \implies j < n \implies$   
 $\text{isDERIV } i \ (fas ! j) \ vs$   
**by** ( $\text{auto simp: isFDERIV-def}$ )

**lemma**  $\text{isFDERIV-max-Var-congI}$ :  $\text{isFDERIV } n \ xs \ fas \ ws$   
**if**  $f$ :  $\text{isFDERIV } n \ xs \ fas \ vs$  **and**  $c$ :  $(\bigwedge i. i < \text{max-Var-floatariths } fas \implies vs ! i = ws ! i)$   
**using**  $c \ f$   
**by** ( $\text{auto simp: isFDERIV-def max-Var-floatariths-lessI}$   
 $\text{intro!: isFDERIV-I isDERIV-max-Var-floatarithI[OF isFDERIV-isDERIV-D[OF } f]])$

**lemma**  $\text{isFDERIV-max-Var-cong}$ :  $\text{isFDERIV } n \ xs \ fas \ ws \longleftrightarrow \text{isFDERIV } n \ xs \ fas \ vs$   
**if**  $c$ :  $(\bigwedge i. i < \text{max-Var-floatariths } fas \implies vs ! i = ws ! i)$   
**using**  $c$  **by** ( $\text{auto intro: isFDERIV-max-Var-congI}$ )

**lemma**  $\text{isDERIV-max-VarI}$ :  
 $i \geq \text{max-Var-floatarith } fa \implies \text{isDERIV } j \ fa \ xs \implies \text{isDERIV } i \ fa \ xs$   
**by** ( $\text{induction } fa$ ) ( $\text{auto simp: isDERIV-Power-iff}$ )

**lemmas**  $\text{max-Var-floatarith-le-max-Var-floatariths-nthI} =$   
 $\text{max-Var-floatarith-le-max-Var-floatariths-nth[THEN order-trans]}$

**lemma**  
 $\text{isFDERIV-appendD1}$ :

```

assumes isFDERIV (J + K) [0.. $J + K$ ] (es @ rs) xs
assumes length es = J
assumes length rs = K
assumes max-Var-floatariths es  $\leq$  J
shows isFDERIV J [0.. $J$ ] (es) xs
unfolding isFDERIV-def
apply (safe)
subgoal for i j
  using assms
  apply (cases i < length es)
subgoal by (auto simp: nth-append isFDERIV-def) (metis add.commute trans-less-add2)
subgoal
  apply (rule isDERIV-max-VarI[where j=0])
  apply (rule max-Var-floatarith-le-max-Var-floatariths-nthI)
    apply force
    apply force
    apply force
  done
done
subgoal by (auto simp: assms)
subgoal by (auto simp: assms)
done

```

```

lemma interpret-floatariths-Var[simp]:
  interpret-floatariths (map floatarith. Var xs) vs = (map (nth vs) xs)
by (induction xs) auto

```

```

lemma max-Var-floatariths-append[simp]: max-Var-floatariths (xs @ ys) = max
(max-Var-floatariths xs) (max-Var-floatariths ys)
by (induction xs) (auto)

```

```

lemma map-nth-append-upt[simp]:
  assumes a  $\geq$  length xs
  shows map (!) (xs @ ys) [a.. $b$ ] = map (!) ys [a - length xs.. $b$  - length xs]
  using assms
  by (auto intro!: nth-equalityI simp: nth-append)

```

```

lemma map-nth-Cons-upt[simp]:
  assumes a > 0
  shows map (!) (x # ys) [a.. $b$ ] = map (!) ys [a - Suc 0.. $b$  - Suc 0]
  using assms
  by (auto intro!: nth-equalityI simp: nth-append)

```

```

lemma map-nth-eq-self[simp]:
  shows length fas = l  $\implies$  (map (!) fas) [0.. $l$ ] = fas
  by (auto simp: intro!: nth-equalityI)

```

**lemma**

*isFDERIV-appendI1*:  
**assumes** *isFDERIV J [0..<J] (es) xs*  
**assumes**  $\bigwedge i j. i < J + K \implies j < K \implies \text{isDERIV } i \text{ (rs ! j) xs}$   
**assumes** *length es = J*  
**assumes** *length rs = K*  
**assumes** *max-Var-floatariths es  $\leq$  J*  
**shows** *isFDERIV (J + K) [0..<J + K] (es @ rs) xs*  
**unfolding** *isFDERIV-def*  
**apply** *safe*  
**subgoal for** *i j*  
  **using** *assms*  
  **apply** (*cases j < length es*)  
  **subgoal**  
    **apply** (*auto simp: nth-append isFDERIV-def*)  
    **by** (*metis (no-types, opaque-lifting) isDERIV-max-VarI le-trans less-le*  
      *max-Var-floatarith-le-max-Var-floatariths-nthI nat-le-linear*)  
    **subgoal by** (*auto simp: nth-append*)  
  **done**  
**subgoal by** (*auto simp: assms*)  
**subgoal by** (*auto simp: assms*)  
**done**

**lemma** *matrix-matrix-mult-zero[simp]*:  
 $a ** 0 = 0 0 ** a = 0$   
**by** (*vector matrix-matrix-mult-def*)+

**lemma** *scaleR-blinfun-compose-left*:  $i *_R (A \ o_L \ B) = i *_R A \ o_L \ B$   
**and** *scaleR-blinfun-compose-right*:  $i *_R (A \ o_L \ B) = A \ o_L \ i *_R B$   
**by** (*auto intro!: blinfun-eqI simp: blinfun.bilinear-simps*)

**lemma**  
*matrix-blinfun-compose*:  
**fixes**  $A \ B :: (\text{real } ^n) \Rightarrow_L (\text{real } ^n)$   
**shows**  $\text{matrix } (A \ o_L \ B) = (\text{matrix } A) ** (\text{matrix } B)$   
**by** *transfer (auto simp: matrix-compose linear-linear)*

**lemma** *matrix-add-rdistrib*:  $((B + C) ** A) = (B ** A) + (C ** A)$   
**by** (*vector matrix-matrix-mult-def sum.distrib[symmetric] field-simps*)

**lemma** *matrix-scaleR-right*:  $r *_R (a :: 'a :: \text{real-algebra-1 } ^n \ ^m) ** b = r *_R (a ** b)$   
**by** (*vector matrix-matrix-mult-def algebra-simps scaleR-sum-right*)

**lemma** *matrix-scaleR-left*:  $(a :: 'a :: \text{real-algebra-1 } ^n \ ^m) ** r *_R b = r *_R (a ** b)$   
**by** (*vector matrix-matrix-mult-def algebra-simps scaleR-sum-right*)

**lemma** *bounded-bilinear-matrix-matrix-mult[bounded-bilinear]*:  
*bounded-bilinear ((\*\*))*:

('a::{euclidean-space, real-normed-algebra-1}^n^m) ⇒  
 ('a::{euclidean-space, real-normed-algebra-1}^p^n) ⇒  
 ('a::{euclidean-space, real-normed-algebra-1}^p^m))  
**unfolding** *bilinear-conv-bounded-bilinear[symmetric]*  
**unfolding** *bilinear-def*  
**apply** *safe*  
**by** *unfold-locales (auto simp: matrix-add-ldistrib matrix-add-rdistrib matrix-scaleR-right matrix-scaleR-left)*

**lemma** *norm-axis: norm (axis ia 1::'a::{real-normed-algebra-1}^n) = 1*  
**by** *(auto simp: axis-def norm-vec-def L2-set-def if-distrib if-distribR sum.delta cong: if-cong)*

**lemma** *abs-vec-nth-blinfun-apply-lemma:*  
**fixes** *x::(real^n) ⇒<sub>L</sub> (real^m)*  
**shows** *abs (vec-nth (blinfun-apply x (axis ia 1)) i) ≤ norm x*  
**apply** *(rule component-le-norm-cart[THEN order-trans])*  
**apply** *(rule norm-blinfun[THEN order-trans])*  
**by** *(auto simp: norm-axis)*

**lemma** *bounded-linear-matrix-blinfun-apply: bounded-linear (λx::(real^n) ⇒<sub>L</sub> (real^m)). matrix (blinfun-apply x)*  
**apply** *standard*  
**subgoal** **by** *(vector blinfun.bilinear-simps matrix-def)*  
**subgoal** **by** *(vector blinfun.bilinear-simps matrix-def)*  
**apply** *(rule exI[where x=real (CARD('n) \* CARD('m))])*  
**apply** *(auto simp: matrix-def)*  
**apply** *(subst norm-vec-def)*  
**apply** *(rule L2-set-le-sum[THEN order-trans])*  
**apply** *simp*  
**apply** *auto*  
**apply** *(rule sum-mono[THEN order-trans])*  
**apply** *(subst norm-vec-def)*  
**apply** *(rule L2-set-le-sum)*  
**apply** *simp*  
**apply** *(rule sum-mono[THEN order-trans])*  
**apply** *(rule sum-mono)*  
**apply** *simp*  
**apply** *(rule abs-vec-nth-blinfun-apply-lemma)*  
**apply** *(simp add: abs-vec-nth-blinfun-apply-lemma)*  
**done**

**lemma** *matrix-has-derivative:*  
**shows** *((λx::(real^n) ⇒<sub>L</sub> (real^n)). matrix (blinfun-apply x)) has-derivative (λh. matrix (blinfun-apply h)) (at x)*  
**apply** *(auto simp: has-derivative-at2)*  
**unfolding** *linear-linear*  
**subgoal** **by** *(rule bounded-linear-matrix-blinfun-apply)*  
**subgoal**

by (auto simp: blinfun.bilinear-simps matrix-def) vector  
done

**lemma**

matrix-comp-has-derivative[derivative-intros]:  
fixes  $f :: 'a :: \text{real-normed-vector} \Rightarrow ((\text{real}^n) \Rightarrow_L (\text{real}^n))$   
assumes (f has-derivative f') (at x within S)  
shows  $((\lambda x. \text{matrix} (\text{blinfun-apply} (f x))) \text{ has-derivative } (\lambda x. \text{matrix} (f' x)))$  (at x within S)  
using has-derivative-compose[OF assms matrix-has-derivative]  
by auto

**fun** inner-floatariths **where**

inner-floatariths [] = Num 0  
| inner-floatariths - [] = Num 0  
| inner-floatariths (x#xs) (y#ys) = Add (Mult x y) (inner-floatariths xs ys)

**lemma** interpret-floatarith-inner-eq:

assumes length xs = length ys  
shows interpret-floatarith (inner-floatariths xs ys) vs =  
 $(\sum i < \text{length } ys. (\text{interpret-floatariths } xs \text{ vs } ! i) * (\text{interpret-floatariths } ys \text{ vs } ! i))$   
using assms  
**proof** (induction rule: list-induct2)  
case Nil  
then show ?case by simp  
**next**  
case (Cons x xs y ys)  
then show ?case  
unfolding length-Cons sum.lessThan-Suc-shift  
by simp  
**qed**

**lemma**

interpret-floatarith-inner-floatariths:  
assumes length xs = DIM('a::executable-euclidean-space)  
assumes length ys = DIM('a)  
assumes eucl-of-list (interpret-floatariths xs vs) = (x::'a)  
assumes eucl-of-list (interpret-floatariths ys vs) = y  
shows interpret-floatarith (inner-floatariths xs ys) vs = x · y  
using assms  
by (subst euclidean-inner)  
(auto simp: interpret-floatarith-inner-eq sum-Basis-sum-nth-Basis-list eucl-of-list-inner  
index-nth-id  
intro!: euclidean-eqI[where 'a='a] sum.cong)

**lemma** max-Var-floatarith-inner-floatariths[simp]:

assumes length f = length g  
shows max-Var-floatarith (inner-floatariths f g) = max (max-Var-floatariths f)  
(max-Var-floatariths g)

**using** *assms*  
**by** (*induction f g rule: list-induct2*) *auto*

**definition** *FDERIV-floatarith where*

*FDERIV-floatarith fa xs d = inner-floatariths (map (λx. fold-const-fa (DERIV-floatarith x fa)) xs) d*

— **TODO:** specialize to *FDERIV-floatarith fa [0..<n] [m..<m + n]* and do the rest with *subst-floatarith?* **TODO:** introduce approximation on type  $((real, 'i) vec, 'j) vec$  and use *jacobian?*

**lemma** *interpret-floatariths-map: interpret-floatariths (map f xs) vs = map (λx. interpret-floatarith (f x) vs) xs*  
**by** (*induct xs*) *auto*

**lemma** *max-Var-floatarith-DERIV-floatarith:*

*max-Var-floatarith (DERIV-floatarith x fa) ≤ max-Var-floatarith fa*  
**by** (*induction x fa rule: DERIV-floatarith.induct*) (*auto*)

**lemma** *max-Var-floatarith-FDERIV-floatarith:*

*length xs = length d ⇒*  
*max-Var-floatarith (FDERIV-floatarith fa xs d) ≤ max (max-Var-floatarith fa)*  
*(max-Var-floatariths d)*

**unfolding** *FDERIV-floatarith-def*

**by** (*auto simp: max-Var-floatariths-Max intro!: max-Var-floatarith-DERIV-floatarith[THEN order-trans]*)  
*max-Var-floatarith-fold-const-fa[THEN order-trans]*)

**definition** *FDERIV-floatariths where*

*FDERIV-floatariths fas xs d = map (λfa. FDERIV-floatarith fa xs d) fas*

**lemma** *max-Var-floatarith-FDERIV-floatariths:*

*length xs = length d ⇒ max-Var-floatariths (FDERIV-floatariths fa xs d) ≤ max*  
*(max-Var-floatariths fa) (max-Var-floatariths d)*

**by** (*auto simp: FDERIV-floatariths-def max-Var-floatariths-Max intro!: max-Var-floatarith-FDERIV-floatarith[THEN order-trans]*)  
*(auto simp: max-def)*

**lemma** *length-FDERIV-floatariths[simp]:*

*length (FDERIV-floatariths fas xs ds) = length fas*  
**by** (*auto simp: FDERIV-floatariths-def*)

**lemma** *FDERIV-floatariths-nth[simp]:*

*i < length fas ⇒ FDERIV-floatariths fas xs ds ! i = FDERIV-floatarith (fas ! i) xs ds*  
**by** (*auto simp: FDERIV-floatariths-def*)

**definition** *FDERIV-n-floatariths fas xs ds n = ((λx. FDERIV-floatariths x xs ds)  $\widehat{\sim}$  n) fas*



**lemma** *FDERIV-n-floatariths-Suc*[simp]:  
*FDERIV-n-floatariths fa xs ds 0 = fa*  
*FDERIV-n-floatariths fa xs ds (Suc n) = FDERIV-n-floatariths (FDERIV-n-floatariths fa xs ds n) xs ds*  
**by** (*auto simp: FDERIV-n-floatariths-def*)

**lemma** *length-FDERIV-n-floatariths*[simp]: *length (FDERIV-n-floatariths fa xs ds n) = length fa*  
**by** (*induction n*) (*auto simp: FDERIV-n-floatariths-def*)

**lemma** *max-Var-floatarith-FDERIV-n-floatariths*:  
*length xs = length d  $\implies$  max-Var-floatariths (FDERIV-n-floatariths fa xs d n)  $\leq$  max (max-Var-floatariths fa) (max-Var-floatariths d)*  
**by** (*induction n*)  
*(auto intro!: max-Var-floatarith-FDERIV-floatariths[THEN order-trans] simp: FDERIV-n-floatariths-def)*

**lemma** *interpret-floatarith-FDERIV-floatarith-cong*:  
**assumes** *rq:  $\bigwedge i. i < \text{max-Var-floatarith } f \implies rs ! i = qs ! i$*   
**assumes** [simp]: *length ds = length xs length es = length xs*  
**assumes** *interpret-floatariths ds qs = interpret-floatariths es rs*  
**shows** *interpret-floatarith (FDERIV-floatarith f xs ds) qs = interpret-floatarith (FDERIV-floatarith f xs es) rs*  
**apply** (*auto simp: FDERIV-floatarith-def interpret-floatarith-inner-eq*)  
**apply** (*rule sum.cong[OF refl]*)  
**subgoal premises prems for i**  
**proof** –  
**have** *interpret-floatarith (DERIV-floatarith (xs ! i) f) qs = interpret-floatarith (DERIV-floatarith (xs ! i) f) rs*  
**apply** (*rule interpret-floatarith-max-Var-cong*)  
**apply** (*auto simp: intro!: rq*)  
**by** (*metis leD le-trans max-Var-floatarith-DERIV-floatarith nat-less-le*)  
**moreover**  
**have** *interpret-floatarith (ds ! i) qs = interpret-floatarith (es ! i) rs*  
**using** *assms*  
**by** (*metis  $\langle i \in \{..<\text{length } xs\} \rangle$  interpret-floatariths-nth lessThan-iff*)  
**ultimately show** *?thesis* **by** *auto*  
**qed**  
**done**

**theorem** *matrix-vector-mult-eq-list-of-eucl-nth*:  
*(M::real<sup>'n</sup>::enum<sup>'m</sup>::enum) \* v =*  
*( $\sum i < \text{CARD}('m).$*   
*( $\sum j < \text{CARD}('n).$  list-of-eucl M ! (i \* CARD('n) + j) \* list-of-eucl v ! j) \*<sub>R</sub>*  
*Basis-list ! i)*  
**using** *eucl-of-list-matrix-vector-mult-eq-sum-nth-Basis-list*[*of list-of-eucl M list-of-eucl v,*

where 'n='n and 'm = 'm]  
 by auto

**definition** *mmult-fa* l m n AS BS =  
 concat (map (λi. map (λk. inner-floatariths  
 (map (λj. AS ! (i \* m + j)) [0..
 [0..

**lemma** *length-mmult-fa[simp]*: length (mmult-fa l m n AS BS) = l \* n  
 by (auto simp: mmult-fa-def length-concat o-def sum-list-distinct-conv-sum-set)

**lemma** *einterpret-mmult-fa*:  
 assumes [simp]: Dn = CARD('n::enum) Dm = CARD('m::enum) Dl = CARD('l::enum)  
 length A = CARD('l)\*CARD('m) length B = CARD('m)\*CARD('n)  
 shows einterpret (mmult-fa Dl Dm Dn A B) vs = (einterpret A vs::((real,  
 'm::enum) vec, 'l) vec) \*\* (einterpret B vs::((real, 'n::enum) vec, 'm) vec)  
 apply (vector matrix-matrix-mult-def)  
 apply (auto simp: mmult-fa-def vec-nth-eucl-of-list-eq2 index-Basis-list-axis2  
 concat-map-map-index length-concat o-def sum-list-distinct-conv-sum-set  
 interpret-floatarith-inner-eq)  
 apply (subst sum-index-enum-eq)  
 apply simp  
 done

**lemma** *max-Var-floatariths-mmult-fa*:  
 assumes [simp]: length A = D \* E length B = E \* F  
 shows max-Var-floatariths (mmult-fa D E F A B) ≤ max (max-Var-floatariths  
 A) (max-Var-floatariths B)  
 apply (auto simp: mmult-fa-def concat-map-map-index intro!: max-Var-floatariths-leI)  
 apply (rule max.coboundedI1)  
 apply (auto intro!: max-Var-floatarith-le-max-Var-floatariths-nth max.coboundedI2)  
 apply (cases F = 0)  
 apply simp-all  
 done

**lemma** *isDERIV-inner-iff*:  
 assumes length xs = length ys  
 shows isDERIV i (inner-floatariths xs ys) vs ↔  
 (∀ k < length xs. isDERIV i (xs ! k) vs) ∧ (∀ k < length ys. isDERIV i (ys ! k)  
 vs)  
 using assms  
 by (induction xs ys rule: list-induct2) (auto simp: nth-Cons split: nat.splits)

**lemma** *isDERIV-Power*: isDERIV x (fa) vs ⇒ isDERIV x (fa  $\widehat{e}$  n) vs  
 by (induction n) (auto simp: FDERIV-floatarith-def isDERIV-inner-iff)

**lemma** *isDERIV-mmult-fa-nth*:  
 assumes ∧j. j < D \* E ⇒ isDERIV i (A ! j) xs  
 assumes ∧j. j < E \* F ⇒ isDERIV i (B ! j) xs

**assumes** [*simp*]:  $\text{length } A = D * E \text{ length } B = E * F \ j < D * F$   
**shows** *isDERIV*  $i$  (*mmult-fa*  $D \ E \ F \ A \ B \ ! \ j$ )  $xs$   
**using** *assms*  
**apply** (*cases*  $F = 0$ )  
**apply** (*auto simp: mmult-fa-def concat-map-map-index isDERIV-inner-iff ac-simps*)  
**apply** (*metis add.commute assms(5) in-square-lemma less-square-imp-div-less mult.commute*)  
**done**

**definition** *mvmult-fa*  $n \ m \ AS \ B =$   
 $\text{map } (\lambda i. \text{inner-floatariths } (\text{map } (\lambda j. AS \ ! \ (i * m + j)) \ [0..<m])) \ (\text{map } (\lambda j. B \ ! \ j) \ [0..<n])$

**lemma** *einterpret-mvmult-fa*:  
**assumes** [*simp*]:  $Dn = \text{CARD}('n::\text{enum}) \ Dm = \text{CARD}('m::\text{enum})$   
 $\text{length } A = \text{CARD}('n) * \text{CARD}('m) \ \text{length } B = \text{CARD}('m)$   
**shows** *einterpret* (*mvmult-fa*  $Dn \ Dm \ A \ B$ )  $vs = (\text{einterpret } A \ vs::(\text{real}, 'm::\text{enum})$   
 $\text{vec}, 'n) \ \text{vec}) * v \ (\text{einterpret } B \ vs::(\text{real}, 'm) \ \text{vec})$   
**apply** (*vector matrix-vector-mult-def*)  
**apply** (*auto simp: mvmult-fa-def vec-nth-eucl-of-list-eq2 index-Basis-list-axis2 index-Basis-list-axis1 vec-nth-eucl-of-list-eq*  
 $\text{concat-map-map-index length-concat o-def sum-list-distinct-conv-sum-set}$   
 $\text{interpret-floatarith-inner-eq}$ )  
**apply** (*subst sum-index-enum-eq*)  
**apply** *simp*  
**done**

**lemma** *max-Var-floatariths-mvult-fa*:  
**assumes** [*simp*]:  $\text{length } A = D * E \ \text{length } B = E$   
**shows** *max-Var-floatariths* (*mvmult-fa*  $D \ E \ A \ B$ )  $\leq \text{max } (\text{max-Var-floatariths } A)$   
 $(\text{max-Var-floatariths } B)$   
**apply** (*auto simp: mvmult-fa-def concat-map-map-index intro!: max-Var-floatariths-leI*)  
**apply** (*rule max.coboundedI1*)  
**by** (*auto intro!: max-Var-floatarith-le-max-Var-floatariths-nth max.coboundedI2*)

**lemma** *isDERIV-mvmult-fa-nth*:  
**assumes**  $\bigwedge j. j < D * E \implies \text{isDERIV } i \ (A \ ! \ j) \ xs$   
**assumes**  $\bigwedge j. j < E \implies \text{isDERIV } i \ (B \ ! \ j) \ xs$   
**assumes** [*simp*]:  $\text{length } A = D * E \ \text{length } B = E \ j < D$   
**shows** *isDERIV*  $i$  (*mvmult-fa*  $D \ E \ A \ B \ ! \ j$ )  $xs$   
**using** *assms*  
**apply** (*auto simp: mvmult-fa-def concat-map-map-index isDERIV-inner-iff ac-simps*)  
**by** (*metis assms(5) in-square-lemma semiring-normalization-rules(24) semiring-normalization-rules(7)*)

**lemma** *max-Var-floatariths-mapI*:  
**assumes**  $\bigwedge x. x \in \text{set } xs \implies \text{max-Var-floatarith } (f \ x) \leq m$   
**shows** *max-Var-floatariths* ( $\text{map } f \ xs$ )  $\leq m$

**using** *assms*  
**by** (*force intro!*: *max-Var-floatariths-leI simp: in-set-conv-nth*)

**lemma**

*max-Var-floatariths-list-updateI*:  
**assumes** *max-Var-floatariths xs ≤ m*  
**assumes** *max-Var-floatarith v ≤ m*  
**assumes** *i < length xs*  
**shows** *max-Var-floatariths (xs[i := v]) ≤ m*  
**using** *assms*  
**apply** (*auto simp: nth-list-update intro!: max-Var-floatariths-leI* )  
**using** *max-Var-floatarith-le-max-Var-floatariths-nthI* **by** *blast*

**lemma**

*max-Var-floatariths-replicateI*:  
**assumes** *max-Var-floatarith v ≤ m*  
**shows** *max-Var-floatariths (replicate n v) ≤ m*  
**using** *assms*  
**by** (*auto intro!: max-Var-floatariths-leI* )

**definition** *FDERIV-n-floatarith fa xs ds n = ((λx. FDERIV-floatarith x xs ds)  $\sim^n$ ) fa*

**lemma** *FDERIV-n-floatariths-nth: i < length fas  $\implies$  FDERIV-n-floatariths fas xs ds n ! i = FDERIV-n-floatarith (fas ! i) xs ds n*  
**by** (*induction n*)  
*(auto simp: FDERIV-n-floatarith-def FDERIV-floatariths-nth)*

**lemma** *einterpret-fold-const-fa[simp]*:

*(einterpret (map (λi. fold-const-fa (fa i)) xs) vs::'a::executable-euclidean-space)*  
 $=$   
*einterpret (map fa xs) vs* **if** *length xs = DIM('a)*  
**using** *that*  
**by** (*auto intro!: euclidean-eqI[where 'a='a] simp: algebra-simps eucl-of-list-inner*)

**lemma** *einterpret-plus[simp]*:

**shows** *(einterpret (map (λi. fa1 i + fa2 i) [0..*DIM('a)*]) vs::'a) =*  
*einterpret (map fa1 [0..*DIM('a)::executable-euclidean-space*]) vs + einterpret*  
*(map fa2 [0..*DIM('a)*]) vs*  
**by** (*auto intro!: euclidean-eqI[where 'a='a] simp: algebra-simps eucl-of-list-inner*)

**lemma** *einterpret-uminus[simp]*:

**shows** *(einterpret (map (λi. - fa1 i) [0..*DIM('a)*]) vs::'a::executable-euclidean-space)*  
 $=$   
*- einterpret (map fa1 [0..*DIM('a)*]) vs*  
**by** (*auto intro!: euclidean-eqI[where 'a='a] simp: algebra-simps eucl-of-list-inner*)

**lemma** *diff-floatarith-conv-add-uminus: a - b = a + - b* **for** *a b::floatarith*

**by** (*auto simp: minus-floatarith-def plus-floatarith-def uminus-floatarith-def*)

**lemma** *einterpret-minus*[simp]:  
**shows** (*einterpret* (*map* ( $\lambda i. fa1\ i - fa2\ i$ ) [ $0..<DIM('a)$ ]) *vs::'a::executable-euclidean-space*)  
 =  
*einterpret* (*map* *fa1* [ $0..<DIM('a)$ ]) *vs* - *einterpret* (*map* *fa2* [ $0..<DIM('a)$ ])  
*vs*  
**by** (*simp add: diff-floatarith-conv-add-uminus*)

**lemma** *einterpret-scaleR*[simp]:  
**shows** (*einterpret* (*map* ( $\lambda i. fa1 * fa2\ i$ ) [ $0..<DIM('a)$ ]) *vs::'a::executable-euclidean-space*)  
 =  
*interpret-floatarith* (*fa1*) *vs* \*<sub>R</sub> *einterpret* (*map* *fa2* [ $0..<DIM('a)$ ]) *vs*  
**by** (*auto intro!: euclidean-eqI*[**where** *'a='a*] *simp: algebra-simps eucl-of-list-inner*)

**lemma** *einterpret-nth*[simp]:  
**assumes** [simp]: *length xs = DIM('a)*  
**shows** (*einterpret* (*map* (!) *xs*) [ $0..<DIM('a)$ ]) *vs::'a::executable-euclidean-space*)  
 = *einterpret xs vs*  
**by** (*auto intro!: euclidean-eqI*[**where** *'a='a*] *simp: algebra-simps eucl-of-list-inner*)

**type-synonym** *'n rvec* = (*real*, *'n*) *vec*

**lemma** *length-mvmult-fa*[simp]: *length (mvmult-fa D E xs ys) = D*  
**by** (*auto simp: mvmult-fa-def*)

**lemma** *interpret-mvmult-nth*:  
**assumes** *D = CARD('n::enum)*  
**assumes** *E = CARD('m::enum)*  
**assumes** *length xs = D \* E*  
**assumes** *length ys = E*  
**assumes** *n < CARD('n)*  
**shows** *interpret-floatarith (mvmult-fa D E xs ys ! n) vs =*  
 (*einterpret xs vs::(real, 'm) vec, 'n) vec* \**v einterpret ys vs*) \* (*Basis-list ! n*)  
**proof** -  
**have** *interpret-floatarith (mvmult-fa D E xs ys ! n) vs = einterpret (mvmult-fa*  
*D E xs ys) vs \* (Basis-list ! n::'n rvec)*  
**using** *assms*  
**by** (*auto simp: eucl-of-list-inner*)  
**also**  
**from** *einterpret-mvmult-fa*[*OF assms(1,2), of xs ys vs*]  
**have** *einterpret (mvmult-fa D E xs ys) vs = (einterpret xs vs::(real, 'm) vec, 'n*  
*vec) \*v einterpret ys vs*  
**using** *assms by simp*  
**finally show** *?thesis by simp*  
**qed**

**lemmas** [*simp del*] = *fold-const-fa.simps*

**lemma** *take-eq-map-nth*:  $n < \text{length } xs \implies \text{take } n \text{ } xs = \text{map } (!) \text{ } xs [0..<n]$   
**by** (*induction xs*) (*auto intro!: nth-equalityI*)

**lemmas** [*simp del*] = *upt-rec-numeral*

**lemmas** *map-nth-eq-take* = *take-eq-map-nth*[*symmetric*]

### 3.7 Definition of Approximating Function using Affine Arithmetic

**lemma** *interpret-Floatreal*: *interpret-floatarith* (*floatarith.Num f*) *vs* = (*real-of-float f*)

**by** *simp*

**ML** <

(\* *Make a congruence rule out of a defining equation for the interpretation*

*th* is one defining equation of *f*,  
*i.e.* *th* is  $f (Cp \ ?t1 \dots \ ?tn) = P(f \ ?t1, \dots, f \ ?tn)$   
*Cp* is a constructor pattern and *P* is a pattern

The result is:

$[[\ ?A1 = f \ ?t1 \ ; \dots \ ; \ ?An = f \ ?tn \ ]] \implies P \ (?A1, \dots, \ ?An) = f \ (Cp \ ?t1 \dots \ ?tn)$   
+ the a list of names of the *A1 .. An*, Those are fresh in the *ctxt* \*)

*fun mk-congeq ctxt fs th* =

*let*

*val Const (fN, -) = th |> Thm.prop-of |> HOLogic.dest-Trueprop |> HO-Logic.dest-eq*

*|> fst |> strip-comb |> fst;*

*val ((-, [th']), ctxt') = Variable.import true [th] ctxt;*

*val (lhs, rhs) = HOLogic.dest-eq (HOLogic.dest-Trueprop (Thm.prop-of th'));*

*fun add-fterms (t as t1 \$ t2) =*

*if exists (fn f => Term.could-unify (t |> strip-comb |> fst, f)) fs*

*then insert (op aconv) t*

*else add-fterms t1 #> add-fterms t2*

*| add-fterms (t as Abs -) =*

*if exists-Const (fn (c, -) => c = fN) t*

*then K [t]*

*else K []*

*| add-fterms - = I;*

*val fterms = add-fterms rhs [];*

*val (xs, ctxt'') = Variable.variant-fixes (replicate (length fterms) x) ctxt';*

*val tys = map fastype-of fterms;*

*val vs = map Free (xs ~~~ tys);*

*val env = fterms ~~~ vs; (\*FIXME\*)*

*fun replace-fterms (t as t1 \$ t2) =*

*(case AList.lookup (op aconv) env t of*

*SOME v => v*

*| NONE => replace-fterms t1 \$ replace-fterms t2)*

```

| replace-fterms t =
  (case AList.lookup (op aconv) env t of
    SOME v => v
  | NONE => t);
fun mk-def (Abs (x, xT, t), v) =
  HOLogic.mk-Trueprop (HOLogic.all-const xT $ Abs (x, xT, HOLogic.mk-eq
(v $ Bound 0, t)))
| mk-def (t, v) = HOLogic.mk-Trueprop (HOLogic.mk-eq (v, t));
fun tryext x =
  (x RS @ {lemma (∀ x. f x = g x) ==> f = g by blast} handle THM - => x);
val cong =
  (Goal.prove ctxt'' [] (map mk-def env)
  (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, replace-fterms rhs)))
  (fn {context = goal-ctxt, prems} =>
    Local-Defs.unfold0-tac goal-ctxt (map tryext prems) THEN resolve-tac
goal-ctxt [th^ 1]))
  RS sym;
val (cong' :: vars') =
  Variable.export ctxt'' ctxt (cong :: map (Drule.mk-term o Thm.cterm-of ctxt'')
vs);
val vs' = map (fst o fst o Term.dest-Var o Thm.term-of o Drule.dest-term)
vars';

in (vs', cong') end;

fun mk-congs ctxt eqs =
  let
    val fs = fold-rev (fn eq => insert (op =) (eq |> Thm.prop-of |> HOLogic.dest-Trueprop
|> HOLogic.dest-eq |> fst |> strip-comb
|> fst)) eqs [];
    val tys = fold-rev (fn f => fold (insert (op =)) (f |> fastype-of |> binder-types
|> tl)) fs [];
    val (vs, ctxt') = Variable.variant-fixes (replicate (length tys) vs) ctxt;
    val subst =
      the o AList.lookup (op =)
      (map2 (fn T => fn v => (T, Thm.cterm-of ctxt' (Free (v, T)))) tys vs);
    fun prep-eq eq =
      let
        val (-, - :: vs) = eq |> Thm.prop-of |> HOLogic.dest-Trueprop
|> HOLogic.dest-eq |> fst |> strip-comb;
        val subst = map-filter (fn Var v => SOME (v, subst (#2 v)) | - => NONE)
vs;
        in Thm.instantiate (TVars.empty, Vars.make subst) eq end;
        val (ps, congs) = map-split (mk-congeq ctxt' fs o prep-eq) eqs;
        val bds = AList.make (K [], []) tys;
        in (ps ~ Variable.export ctxt' ctxt congs, bds) end
      >
  >

```

ML <

```

fun interpret-floatariths-congs ctxt =
  mk-congs ctxt @ { thms interpret-floatarith.simps interpret-floatariths.simps }
  |> fst
  |> map snd
  >

```

```

ML <
fun preproc-form-conv ctxt =
  Simplifier.rewrite
  (put-simpset HOL-basic-ss ctxt addsimps
   (Named-Theorems.get ctxt @ { named-theorems approximation-preproc })))
  >

```

```

ML < fun reify-floatariths-tac ctxt i =
  CONVERSION (preproc-form-conv ctxt) i
  THEN REPEAT-ALL-NEW (fn i => resolve-tac ctxt (interpret-floatariths-congs
  ctxt) i) i >

```

```

method-setup reify-floatariths = <
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (reify-floatariths-tac ctxt))
  > reification of floatariths expression

```

```

schematic-goal reify-example:
  [xs!i * xs!j, xs!i + xs!j pour (sin (xs!0)), xs!k + (2 / 3 * xs!i * xs!j)] = inter-
  pret-floatariths ?fas xs
  by (reify-floatariths)

```

```

ML < fun interpret-floatariths-step-tac ctxt i = resolve-tac ctxt (interpret-floatariths-congs
  ctxt) i >

```

```

method-setup reify-floatariths-step = <
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (interpret-floatariths-step-tac ctxt))
  > reification of floatariths expression (step)

```

```

lemma eucl-of-list-interpret-floatariths-cong:
  fixes y::'a::executable-euclidean-space
  assumes  $\bigwedge b. b \in \text{Basis} \implies \text{interpret-floatarith} (fa (\text{index Basis-list } b)) \text{ vs} = y$ 
  . b
  assumes length xs = DIM('a)
  shows eucl-of-list (interpret-floatariths (map fa [0.. $\text{DIM}('a)$ ]) vs) = y
  apply (rule euclidean-eqI)
  apply (subst eucl-of-list-inner)
  by (auto simp: assms)

```

```

lemma interpret-floatariths-fold-const-fa[simp]:
  interpret-floatariths (map fold-const-fa ds) = interpret-floatariths ds
  by (auto intro!: nth-equalityI)

```

```

fun subst-floatarith where
  subst-floatarith s (Add a b) = Add (subst-floatarith s a) (subst-floatarith s b)

```



$\mid$		
$\text{subst-floatarith } s \text{ (Mult } a \ b)$	$=$	$\text{Mult (subst-floatarith } s \ a) \ (\text{subst-floatarith } s \ b)$
$\mid$		
$\text{subst-floatarith } s \ \text{(Minus } a)$	$=$	$\text{Minus (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Inverse } a)$	$=$	$\text{Inverse (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Cos } a)$	$=$	$\text{Cos (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Arctan } a)$	$=$	$\text{Arctan (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Min } a \ b)$	$=$	$\text{Min (subst-floatarith } s \ a) \ (\text{subst-floatarith } s \ b)$
$\mid$		
$\text{subst-floatarith } s \ \text{(Max } a \ b)$	$=$	$\text{Max (subst-floatarith } s \ a) \ (\text{subst-floatarith } s \ b)$
$\mid$		
$\text{subst-floatarith } s \ \text{(Abs } a)$	$=$	$\text{Abs (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{Pi}$	$=$	$\text{Pi} \ \mid$
$\text{subst-floatarith } s \ \text{(Sqrt } a)$	$=$	$\text{Sqrt (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Exp } a)$	$=$	$\text{Exp (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Powr } a \ b)$	$=$	$\text{Powr (subst-floatarith } s \ a) \ (\text{subst-floatarith } s \ b)$
$\mid$		
$\text{subst-floatarith } s \ \text{(Ln } a)$	$=$	$\text{Ln (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Power } a \ i)$	$=$	$\text{Power (subst-floatarith } s \ a) \ i \ \mid$
$\text{subst-floatarith } s \ \text{(Floor } a)$	$=$	$\text{Floor (subst-floatarith } s \ a) \ \mid$
$\text{subst-floatarith } s \ \text{(Num } f)$	$=$	$\text{Num } f \ \mid$
$\text{subst-floatarith } s \ \text{(Var } n)$	$=$	$s \ n$

**lemma** *interpret-floatarith-subst-floatarith*:

**assumes**  $\text{max-Var-floatarith } fa \leq D$

**shows**  $\text{interpret-floatarith (subst-floatarith } s \ fa) \ vs =$

$\text{interpret-floatarith } fa \ (\text{map } (\lambda i. \text{interpret-floatarith } (s \ i) \ vs) \ [0..<D])$

**using** *assms*

**by** (*induction fa*) *auto*

**lemma** *max-Var-floatarith-subst-floatarith-le*[*THEN order-trans*]:

**assumes**  $\text{length } xs \geq \text{max-Var-floatarith } fa$

**shows**  $\text{max-Var-floatarith (subst-floatarith (!) } xs) \ fa) \leq \text{max-Var-floatariths } xs$

**using** *assms*

**by** (*induction fa*) (*auto intro!*: *max-Var-floatarith-le-max-Var-floatariths-nth*)

**lemma** *max-Var-floatariths-subst-floatarith-le*[*THEN order-trans*]:

**assumes**  $\text{length } xs \geq \text{max-Var-floatariths } fas$

**shows**  $\text{max-Var-floatariths (map (subst-floatarith (!) } xs)) \ fas) \leq \text{max-Var-floatariths } xs$

*xs*

**using** *assms*

**by** (*induction fas*) (*auto simp*: *max-Var-floatarith-subst-floatarith-le*)

**fun** *continuous-on-floatarith* :: *floatarith*  $\Rightarrow$  *bool* **where**

$\text{continuous-on-floatarith (Add } a \ b) = (\text{continuous-on-floatarith } a \ \wedge \ \text{continuous-on-floatarith } b) \ \mid$

$\text{continuous-on-floatarith (Mult } a \ b) = (\text{continuous-on-floatarith } a \ \wedge \ \text{continuous-on-floatarith } b) \ \mid$

$\text{continuous-on-floatarith (Minus } a) = \text{continuous-on-floatarith } a \ \mid$

$\text{continuous-on-floatarith } (\text{Inverse } a) = \text{False} \mid$   
 $\text{continuous-on-floatarith } (\text{Cos } a) = \text{continuous-on-floatarith } a \mid$   
 $\text{continuous-on-floatarith } (\text{Arctan } a) = \text{continuous-on-floatarith } a \mid$   
 $\text{continuous-on-floatarith } (\text{Min } a \ b) = (\text{continuous-on-floatarith } a \ \wedge \ \text{continuous-on-floatarith } b) \mid$   
 $\text{continuous-on-floatarith } (\text{Max } a \ b) = (\text{continuous-on-floatarith } a \ \wedge \ \text{continuous-on-floatarith } b) \mid$   
 $\text{continuous-on-floatarith } (\text{Abs } a) = \text{continuous-on-floatarith } a \mid$   
 $\text{continuous-on-floatarith } \text{Pi} = \text{True} \mid$   
 $\text{continuous-on-floatarith } (\text{Sqrt } a) = \text{False} \mid$   
 $\text{continuous-on-floatarith } (\text{Exp } a) = \text{continuous-on-floatarith } a \mid$   
 $\text{continuous-on-floatarith } (\text{Powr } a \ b) = \text{False} \mid$   
 $\text{continuous-on-floatarith } (\text{Ln } a) = \text{False} \mid$   
 $\text{continuous-on-floatarith } (\text{Floor } a) = \text{False} \mid$   
 $\text{continuous-on-floatarith } (\text{Power } a \ n) = (\text{if } n = 0 \ \text{then } \text{True} \ \text{else } \text{continuous-on-floatarith } a) \mid$   
 $\text{continuous-on-floatarith } (\text{Num } f) = \text{True} \mid$   
 $\text{continuous-on-floatarith } (\text{Var } n) = \text{True}$

**definition**  $\text{Maxs}_e \ xs = \text{fold } (\lambda a \ b. \ \text{floatarith.Max } a \ b) \ xs$

**definition**  $\text{norm2}_e \ n = \text{Maxs}_e \ (\text{map } (\lambda j. \ \text{Norm } (\text{map } (\lambda i. \ \text{Var } (\text{Suc } j * n + i)) \ [0..<n]))) \ [0..<n]) \ (\text{Num } 0)$

**definition**  $N_r \ l = \text{Num } (\text{float-of } l)$

**lemma** *interpret-floatarith-Norm:*

$\text{interpret-floatarith } (\text{Norm } xs) \ vs = \text{L2-set } (\lambda i. \ \text{interpret-floatarith } (xs \ ! \ i) \ vs) \ \{0..<\text{length } xs\}$

**by** (*auto simp: Norm-def L2-set-def sum-list-sum-nth power2-eq-square*)

**lemma** *interpret-floatarith-Nr[simp]:*  $\text{interpret-floatarith } (N_r \ U) \ vs = \text{real-of-float } (\text{float-of } U)$

**by** (*auto simp: N\_r-def*)

**fun** *list-updates where*

$\text{list-updates } [] \ - \ xs = xs$   
 $\mid \text{list-updates } - \ [] \ xs = xs$   
 $\mid \text{list-updates } (i\#is) \ (u\#us) \ xs = \text{list-updates } is \ us \ (xs[i:=u])$

**lemma** *list-updates-nth-notmem:*

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

**assumes**  $i \notin \text{set } xs$

**shows**  $\text{list-updates } xs \ ys \ vs \ ! \ i = vs \ ! \ i$

**using** *assms*

**by** (*induction xs ys arbitrary: i vs rule: list-induct2*) *auto*

**lemma** *list-updates-nth-less:*

**assumes**  $length\ xs = length\ ys\ distinct\ xs$   
**assumes**  $i < length\ vs$   
**shows**  $list\ updates\ xs\ ys\ vs\ !\ i = (if\ i \in set\ xs\ then\ ys\ !\ (index\ xs\ i)\ else\ vs\ !\ i)$   
**using**  $assms$   
**by**  $(induction\ xs\ ys\ arbitrary:\ i\ vs\ rule:\ list\ induct2)\ (auto\ simp:\ list\ updates\ nth\ notmem)$

**lemma**  $length\ list\ updates[simp]:\ length\ (list\ updates\ xs\ ys\ vs) = length\ vs$   
**by**  $(induction\ xs\ ys\ vs\ rule:\ list\ updates.\ induct)\ simp\ all$

**lemma**  $list\ updates\ nth\ ge[simp]:$   
 $x \geq length\ vs \implies length\ xs = length\ ys \implies list\ updates\ xs\ ys\ vs\ !\ x = vs\ !\ x$   
**apply**  $(induction\ xs\ ys\ vs\ rule:\ list\ updates.\ induct)$   
**apply**  $(auto\ simp:\ nth\ list\ update)$   
**by**  $(metis\ list\ update\ beyond\ nth\ list\ update\ neq)$

**lemma**  
 $list\ updates\ nth:$   
**assumes**  $[simp]:\ length\ xs = length\ ys\ distinct\ xs$   
**shows**  $list\ updates\ xs\ ys\ vs\ !\ i = (if\ i < length\ vs \wedge i \in set\ xs\ then\ ys\ !\ index\ xs\ i\ else\ vs\ !\ i)$   
**by**  $(auto\ simp:\ list\ updates\ nth\ less\ list\ updates\ nth\ notmem)$

**lemma**  $list\ of\ eucl\ coord\ update:$   
**assumes**  $[simp]:\ length\ xs = DIM('a::executable\ euclidean\ space)$   
**assumes**  $[simp]:\ distinct\ xs$   
**assumes**  $[simp]:\ i \in Basis$   
**assumes**  $[simp]:\ \bigwedge n.\ n \in set\ xs \implies n < length\ vs$   
**shows**  $list\ updates\ xs\ (list\ of\ eucl\ (x + (p - x \cdot i) *_{\mathbb{R}} i)::'a)\ vs =$   
 $(list\ updates\ xs\ (list\ of\ eucl\ x)\ vs)[xs\ !\ index\ Basis\ list\ i := p]$   
**apply**  $(auto\ intro!:\ nth\ equalityI\ simp:\ list\ updates\ nth\ nth\ list\ update)$   
**apply**  $(simp\ add:\ algebra\ simps\ inner\ Basis\ index\ nth\ id)$   
**apply**  $(auto\ simp\ add:\ algebra\ simps\ inner\ Basis\ index\ nth\ id)$   
**done**

**definition**  $eucl\ of\ env\ is\ vs = eucl\ of\ list\ (map\ (nth\ vs)\ is)$

**lemma**  $list\ updates\ list\ of\ eucl\ of\ env[simp]:$   
**assumes**  $[simp]:\ length\ xs = DIM('a::executable\ euclidean\ space)\ distinct\ xs$   
**shows**  $list\ updates\ xs\ (list\ of\ eucl\ (eucl\ of\ env\ xs\ vs)::'a)\ vs = vs$   
**by**  $(auto\ intro!:\ nth\ equalityI\ simp:\ list\ updates\ nth\ nth\ list\ update\ eucl\ of\ env\ def)$

**lemma**  $nth\ nth\ eucl\ of\ env\ inner:$   
 $b \in Basis \implies length\ is = DIM('a) \implies vs\ !\ (is\ !\ index\ Basis\ list\ b) = eucl\ of\ env\ is\ vs \cdot b$   
**for**  $b::'a::executable\ euclidean\ space$   
**by**  $(auto\ simp:\ eucl\ of\ env\ def\ eucl\ of\ list\ inner)$

**lemma**  $list\ updates\ idem[simp]:$   
**assumes**  $(\bigwedge i.\ i \in set\ X0 \implies i < length\ vs)$

**shows**  $(list\text{-}updates\ X0\ (map\ (!)\ vs)\ X0)\ vs) = vs$   
**using** *assms*  
**by**  $(induction\ X0)\ auto$

**lemma** *pairwise-orthogonal-Basis*[*intro, simp*]: *pairwise orthogonal Basis*  
**by**  $(auto\ simp:\ pairwise\text{-}alt\ orthogonal\text{-}def\ inner\text{-}Basis)$

**primrec** *freshs-floatarith* **where**

$freshs\text{-}floatarith\ (Var\ y)\ x \longleftrightarrow (y \notin set\ x)$   
 $freshs\text{-}floatarith\ (Num\ a)\ x \longleftrightarrow True$   
 $freshs\text{-}floatarith\ Pi\ x \longleftrightarrow True$   
 $freshs\text{-}floatarith\ (Cos\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Abs\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Arctan\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Sqrt\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Exp\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Floor\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Power\ a\ n)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Minus\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Ln\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Inverse\ a)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x$   
 $freshs\text{-}floatarith\ (Add\ a\ b)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x \wedge freshs\text{-}floatarith\ b\ x$   
 $freshs\text{-}floatarith\ (Mult\ a\ b)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x \wedge freshs\text{-}floatarith\ b\ x$   
 $freshs\text{-}floatarith\ (floatarith.Max\ a\ b)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x \wedge freshs\text{-}floatarith\ b\ x$   
 $freshs\text{-}floatarith\ (floatarith.Min\ a\ b)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x \wedge freshs\text{-}floatarith\ b\ x$   
 $freshs\text{-}floatarith\ (Powr\ a\ b)\ x \longleftrightarrow freshs\text{-}floatarith\ a\ x \wedge freshs\text{-}floatarith\ b\ x$

**lemma** *freshs-floatarith*[*simp*]:

**assumes**  $freshs\text{-}floatarith\ fa\ ds\ length\ ds = length\ xs$   
**shows**  $interpret\text{-}floatarith\ fa\ (list\text{-}updates\ ds\ xs\ vs) = interpret\text{-}floatarith\ fa\ vs$   
**using** *assms*  
**by**  $(induction\ fa)\ (auto\ simp:\ list\text{-}updates\text{-}nth\text{-}notmem)$

**lemma** *freshs-floatarith-max-Var-floatarithI*:

**assumes**  $\bigwedge x. x \in set\ xs \implies max\text{-}Var\text{-}floatarith\ f \leq x$   
**shows**  $freshs\text{-}floatarith\ f\ xs$   
**using** *assms Suc-n-not-le-n*  
**by**  $(induction\ f;\ force)$

**definition** *freshs-floatariths fas xs* =  $(\forall fa \in set\ fas. freshs\text{-}floatarith\ fa\ xs)$

**lemma** *freshs-floatariths-max-Var-floatarithsI*:

**assumes**  $\bigwedge x. x \in set\ xs \implies max\text{-}Var\text{-}floatariths\ f \leq x$   
**shows**  $freshs\text{-}floatariths\ f\ xs$   
**using** *assms le-trans max-Var-floatarith-le-max-Var-floatariths*  
**by**  $(force\ simp:\ freshs\text{-}floatariths\text{-}def\ intro!\ freshs\text{-}floatarith\text{-}max\text{-}Var\text{-}floatarithI)$

**lemma** *freshs-floatariths-freshs-floatarithI*:  
**assumes**  $\bigwedge fa. fa \in \text{set fas} \implies \text{freshs-floatarith } fa \text{ } xs$   
**shows** *freshs-floatariths fas xs*  
**by** (*auto simp: freshs-floatariths-def assms*)

**lemma** *fresh-floatariths-fresh-floatarithI*:  
**assumes** *freshs-floatariths fas xs*  
**assumes**  $fa \in \text{set fas}$   
**shows** *fresh-floatarith fa xs*  
**using** *assms*  
**by** (*auto simp: fresh-floatariths-def*)

**lemma** *fresh-floatariths-fresh-floatarith[simp]*:  
*fresh-floatariths (fas) i  $\implies$  fa  $\in$  set fas  $\implies$  fresh-floatarith fa i*  
**by** (*induction fas*) *auto*

**lemma** *interpret-floatariths-fresh-cong*:  
**assumes**  $\bigwedge i. \neg \text{fresh-floatariths } f \text{ } i \implies xs ! i = ys ! i$   
**shows** *interpret-floatariths f ys = interpret-floatariths f xs*  
**by** (*auto intro!: nth-equalityI assms interpret-floatarith-fresh-cong simp:*)

**fun** *subterms* :: *floatarith  $\Rightarrow$  floatarith set where*  
*subterms (Add a b) = insert (Add a b) (subterms a  $\cup$  subterms b) |*  
*subterms (Mult a b) = insert (Mult a b) (subterms a  $\cup$  subterms b) |*  
*subterms (Min a b) = insert (Min a b) (subterms a  $\cup$  subterms b) |*  
*subterms (floatarith.Max a b) = insert (floatarith.Max a b) (subterms a  $\cup$  subterms b) |*  
*subterms (Powr a b) = insert (Powr a b) (subterms a  $\cup$  subterms b) |*  
*subterms (Inverse a) = insert (Inverse a) (subterms a) |*  
*subterms (Cos a) = insert (Cos a) (subterms a) |*  
*subterms (Arctan a) = insert (Arctan a) (subterms a) |*  
*subterms (Abs a) = insert (Abs a) (subterms a) |*  
*subterms (Sqrt a) = insert (Sqrt a) (subterms a) |*  
*subterms (Exp a) = insert (Exp a) (subterms a) |*  
*subterms (Ln a) = insert (Ln a) (subterms a) |*  
*subterms (Power a n) = insert (Power a n) (subterms a) |*  
*subterms (Floor a) = insert (Floor a) (subterms a) |*  
*subterms (Minus a) = insert (Minus a) (subterms a) |*  
*subterms Pi = {Pi} |*  
*subterms (Var v) = {Var v} |*  
*subterms (Num n) = {Num n}*

**lemma** *subterms-self[simp]*:  $fa2 \in \text{subterms } fa2$   
**by** (*induction fa2*) *auto*

**lemma** *interpret-floatarith-FDERIV-floatarith-eucl-of-env*:— **TODO**: cleanup, reduce to DERIV?!  
**assumes**  $iD: \bigwedge i. i < DIM('a) \implies isDERIV (xs ! i) fa \text{ } vs$

```

assumes ds-fresh: freshs-floatarith fa ds
assumes [simp]: length xs = DIM ('a) length ds = DIM ('a)
   $\wedge i. i \in \text{set } xs \implies i < \text{length } vs \text{ distinct } xs$ 
   $\wedge i. i \in \text{set } ds \implies i < \text{length } vs \text{ distinct } ds$ 
shows (( $\lambda x::'a::\text{executable-euclidean-space.}$ 
  (interpret-floatarith fa (list-updates xs (list-of-eucl x) vs))) has-derivative
  ( $\lambda d. \text{interpret-floatarith (FDERIV-floatarith fa xs (map Var ds)) (list-updates}$ 
ds (list-of-eucl d) vs) )
  ) (at (eucl-of-env xs vs))
using iD ds-fresh
proof (induction fa)
case (Add fa1 fa2)
then show ?case
by (auto intro!: derivative-eq-intros simp: FDERIV-floatarith-def interpret-floatarith-inner-floatariths

  interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric])
next
case (Minus fa)
then show ?case
by (auto intro!: derivative-eq-intros simp: FDERIV-floatarith-def interpret-floatarith-inner-floatariths

  interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric])
next
case (Mult fa1 fa2)
then show ?case
by (auto intro!: derivative-eq-intros simp: FDERIV-floatarith-def interpret-floatarith-inner-floatariths

  interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric])
next
case (Inverse fa)
then show ?case
by (force intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
  interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric]
power2-eq-square)
next
case (Cos fa)
then show ?case
by (auto intro!: derivative-eq-intros ext simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
  interpret-floatariths-map add.commute minus-sin-cos-eq
  simp flip: mult-minus-left list-of-eucl-coord-update cos-pi-minus)
next
case (Arctan fa)
then show ?case
by (auto intro!: derivative-eq-intros simp: FDERIV-floatarith-def interpret-floatarith-inner-floatariths

  interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric])
next

```

```

case (Abs fa)
then show ?case
  by (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
next
  case (Max fa1 fa2)
  then show ?case
    by (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
  next
    case (Min fa1 fa2)
    then show ?case
      by (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
    next
      case Pi
      then show ?case
        by (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
      next
        case (Sqrt fa)
        then show ?case
          by (force intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
        next
          case (Exp fa)
          then show ?case
            by (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
            next
              case (Powr fa1 fa2)
              then show ?case
                by (force intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps divide-simps list-of-eucl-coord-update[symmetric]
)
              next
                case (Ln fa)
                then show ?case
                  by (force intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
                next

```

```

case (Power fa x2a)
then show ?case
  apply (cases x2a)
  apply (auto intro!: DIM-positive derivative-eq-intros simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
    interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric])
  apply (auto intro!: ext simp: )
  by (simp add: semiring-normalization-rules(27))
next
case (Floor fa)
then show ?case
  by (force intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
    interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
next
case (Var x)
then show ?case
  apply (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
    interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric]
if-distrib)
  apply (subst list-updates-nth)
  apply (auto intro!: derivative-eq-intros ext split: if-splits
    cong: if-cong simp: if-distribR eucl-of-list-if)
  apply (subst inner-commute)
  apply (rule arg-cong[where f= $\lambda b. a \cdot b$  for a])
  apply (auto intro!: euclidean-eqI[where 'a='a] simp: eucl-of-list-inner list-updates-nth
index-nth-id)
  done
next
case (Num x)
then show ?case
  by (auto intro!: derivative-eq-intros DIM-positive simp: FDERIV-floatarith-def
interpret-floatarith-inner-floatariths
    interpret-floatariths-map algebra-simps list-of-eucl-coord-update[symmetric] )
qed

```

**lemma** *interpret-floatarith-FDERIV-floatarith-append:*  
**assumes** *iD*:  $\bigwedge i j. i < DIM('a) \implies isDERIV\ i\ (fa)\ (list-of-eucl\ x\ @\ params)$   
**assumes** *m*:  $max-Var-floatarith\ fa \leq DIM('a) + length\ params$   
**shows**  $((\lambda x::'a::executable-euclidean-space.$   
*interpret-floatarith fa (list-of-eucl x @ params)) has-derivative*  
 $(\lambda d. interpret-floatarith$   
 $(FDERIV-floatarith fa [0..<DIM('a)] (map Var [length params + DIM('a)..<length$   
 $params + 2*DIM('a)]))$   
 $(list-of-eucl x @ params @ list-of-eucl d))) (at x)$   
**proof** –  
**have** *m-nth*:  $ia < max-Var-floatarith\ fa \implies ia < DIM('a) + length\ params$  **for**  
*ia*



**using** *less-le-trans m by blast*  
**have** (( $\lambda xa::'a$ . *interpret-floatarith fa*  
*(list-updates [0.. $DIM('a)$ ] (list-of-eucl xa) (list-of-eucl x @ params @*  
*replicate  $DIM('a)$  0))) has-derivative*  
*( $\lambda d$ . *interpret-floatarith (FDERIV-floatarith fa [0.. $DIM('a)$ ] (map Var [length*  
*params +  $DIM('a)$ .. $length\ params + 2 * DIM('a)$ ]))*  
*(list-updates [length params +  $DIM('a)$ .. $length\ params + 2 * DIM('a)$ ]*  
*(list-of-eucl d)*  
*(list-of-eucl x @ params @ replicate  $DIM('a)$  0))))*  
*(at (eucl-of-env [0.. $DIM('a)$ ] (list-of-eucl x @ params @ replicate  $DIM('a)$  0)))*  
**by** (*rule interpret-floatarith-FDERIV-floatarith-eucl-of-env*)  
*(auto intro!: iD freshs-floatarith-max-Var-floatarithI isDERIV-max-Var-floatarithI [OF*  
*iD]*  
*max-Var-floatarith-le-max-Var-floatariths[THEN order-trans] m[THEN or-*  
*der-trans]*  
*simp: nth-append add commute less-diff-conv2 m-nth)*  
**moreover have** *interpret-floatarith fa (list-updates [0.. $DIM('a)$ ] (list-of-eucl*  
*xa) (list-of-eucl x @ params @ replicate  $DIM('a)$  0)) =*  
*interpret-floatarith fa (list-of-eucl xa @ params) for  $xa::'a$*   
**apply** (*auto intro!: nth-equalityI interpret-floatarith-max-Var-cong simp:*)  
**apply** (*auto simp: list-updates-nth nth-append dest: m-nth*)  
**done**  
**moreover have** *(list-updates [length params +  $DIM('a)$ .. $length\ params + 2 *$*   
 *$DIM('a)$ ] (list-of-eucl d) (list-of-eucl x @ params @ replicate  $DIM('a)$  0)) =*  
*(list-of-eucl x @ params @ list-of-eucl d) for  $d::'a$*   
**by** (*auto simp: intro!: nth-equalityI simp: list-updates-nth nth-append add commute*)  
**moreover have** *(eucl-of-env [0.. $DIM('a)$ ] (list-of-eucl x @ params @ replicate*  
 *$DIM('a)$  0)) = x*  
**by** (*auto intro!: euclidean-eqI[where 'a='a] simp: eucl-of-env-def eucl-of-list-inner*  
*nth-append*)  
**ultimately show** *?thesis by simp*  
**qed***

**lemma** *interpret-floatarith-FDERIV-floatarith:*  
**assumes** *iD:  $\bigwedge i j. i < DIM('a) \implies isDERIV i (fa) (list-of-eucl x)$*   
**assumes** *m: max-Var-floatarith fa  $\leq DIM('a)$*   
**shows** (*( $\lambda x::'a::executable-euclidean-space$ .*  
*interpret-floatarith fa (list-of-eucl x)) has-derivative*  
*( $\lambda d$ . *interpret-floatarith*  
*(FDERIV-floatarith fa [0.. $DIM('a)$ ] (map Var [ $DIM('a)$ .. $2 * DIM('a)$ ]))*  
*(list-of-eucl x @ list-of-eucl d))) (at x)*  
**using** *interpret-floatarith-FDERIV-floatarith-append[where params=Nil,simplified,*  
*OF assms]*  
**by** *simp**

**lemma** *interpret-floatarith-eventually-isDERIV:*  
**assumes** *iD:  $\bigwedge i j. i < DIM('a) \implies isDERIV i fa (list-of-eucl x @ params)$*   
**assumes** *m: max-Var-floatarith fa  $\leq DIM('a::executable-euclidean-space) + length$*   
*params*

**shows**  $\forall i < DIM('a). \forall_F (x::'a) \text{ in at } x. \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params)}$   
**using**  $iD \ m$   
**proof** (*induction fa*)  
**case** (*Inverse fa*)  
**then have**  $\forall i < DIM('a). \forall_F x \text{ in at } x. \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params)}$   
**by** *auto*  
**moreover**  
**have**  $iD: i < DIM('a) \implies \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params) interpret-floatarith}$   
 $\text{fa (list-of-eucl } x \text{ @ params)} \neq 0$  **for**  $i$   
**using**  $\text{Inverse.prem}(1)[OF]$  **by** *force+*  
**from** *Inverse* **have**  $m: \text{max-Var-floatarith } \text{fa} \leq DIM('a) + \text{length } \text{params}$  **by**  
*simp*  
**from**  $\text{has-derivative-continuous}[OF \text{ interpret-floatarith-FDERIV-floatarith-append,}$   
 $OF \ iD(1) \ m]$   
**have**  $\text{isCont } (\lambda x. \text{interpret-floatarith } \text{fa (list-of-eucl } x \text{ @ params)}) \ x$  **by** *simp*  
**then have**  $\forall_F x \text{ in at } x. \text{interpret-floatarith } \text{fa (list-of-eucl } x \text{ @ params)} \neq 0$   
**using**  $iD(2) \ \text{tendsto-imp-eventually-ne}$   
**by** (*auto simp: isCont-def*)  
**ultimately**  
**show** *?case*  
**by** (*auto elim: eventually-elim2*)  
**next**  
**case** (*Sqrt fa*)  
**then have**  $\forall i < DIM('a). \forall_F x \text{ in at } x. \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params)}$   
**by** *auto*  
**moreover**  
**have**  $iD: i < DIM('a) \implies \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params) interpret-floatarith}$   
 $\text{fa (list-of-eucl } x \text{ @ params)} > 0$  **for**  $i$   
**using**  $\text{Sqrt.prem}(1)[OF]$  **by** *force+*  
**from** *Sqrt* **have**  $m: \text{max-Var-floatarith } \text{fa} \leq DIM('a) + \text{length } \text{params}$  **by** *simp*  
**from**  $\text{has-derivative-continuous}[OF \text{ interpret-floatarith-FDERIV-floatarith-append,}$   
 $OF \ iD(1) \ m]$   
**have**  $\text{isCont } (\lambda x. \text{interpret-floatarith } \text{fa (list-of-eucl } x \text{ @ params)}) \ x$  **by** *simp*  
**then have**  $\forall_F x \text{ in at } x. \text{interpret-floatarith } \text{fa (list-of-eucl } x \text{ @ params)} > 0$   
**using**  $iD(2) \ \text{order-tendstoD}$   
**by** (*auto simp: isCont-def*)  
**ultimately**  
**show** *?case*  
**by** (*auto elim: eventually-elim2*)  
**next**  
**case** (*Powr fa1 fa2*)  
**then have**  $\forall i < DIM('a). \forall_F x \text{ in at } x. \text{isDERIV } i \text{ fa1 (list-of-eucl } x \text{ @ params)}$   
 $\forall i < DIM('a). \forall_F x \text{ in at } x. \text{isDERIV } i \text{ fa2 (list-of-eucl } x \text{ @ params)}$   
**by** *auto*  
**moreover**  
**have**  $iD: i < DIM('a) \implies \text{isDERIV } i \text{ fa1 (list-of-eucl } x \text{ @ params) inter-}$   
 $\text{pret-floatarith } \text{fa1 (list-of-eucl } x \text{ @ params)} > 0$   
**for**  $i$   
**using**  $\text{Powr.prem}(1)$  **by** *force+*

**from** *Power* **have**  $m: \text{max-Var-floatarith } fa1 \leq \text{DIM}('a) + \text{length } \text{params}$  **by** *simp*  
**from** *has-derivative-continuous*[*OF interpret-floatarith-FDERIV-floatarith-append,*  
*OF iD(1) m*]  
**have**  $\text{isCont } (\lambda x. \text{interpret-floatarith } fa1 \text{ (list-of-eucl } x \text{ @ params)}) x$  **by** *simp*  
**then have**  $\forall_F x \text{ in at } x. \text{interpret-floatarith } fa1 \text{ (list-of-eucl } x \text{ @ params)} > 0$   
**using** *iD(2) order-tendstoD*  
**by** (*auto simp: isCont-def*)  
**ultimately**  
**show** *?case*  
**apply** *safe*  
**subgoal for**  $i$   
**apply** (*safe dest!: spec[of - i]*)  
**subgoal premises** *prems*  
**using** *prems(1,3,4)*  
**by** *eventually-elim auto*  
**done**  
**done**  
**next**  
**case** (*Ln fa*)  
**then have**  $\forall i < \text{DIM}('a). \forall_F x \text{ in at } x. \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params)}$   
**by** *auto*  
**moreover**  
**have**  $iD: i < \text{DIM}('a) \implies \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params) interpret-floatarith}$   
 $\text{fa (list-of-eucl } x \text{ @ params)} > 0$  **for**  $i$   
**using** *Ln.prems(1)[OF ] by force+*  
**from** *Ln* **have**  $m: \text{max-Var-floatarith } fa \leq \text{DIM}('a) + \text{length } \text{params}$  **by** *simp*  
**from** *has-derivative-continuous*[*OF interpret-floatarith-FDERIV-floatarith-append,*  
*OF iD(1) m*]  
**have**  $\text{isCont } (\lambda x. \text{interpret-floatarith } fa \text{ (list-of-eucl } x \text{ @ params)}) x$  **by** *simp*  
**then have**  $\forall_F x \text{ in at } x. \text{interpret-floatarith } fa \text{ (list-of-eucl } x \text{ @ params)} > 0$   
**using** *iD(2) order-tendstoD*  
**by** (*auto simp: isCont-def*)  
**ultimately**  
**show** *?case*  
**by** (*auto elim: eventually-elim2*)  
**next**  
**case** (*Power fa m*) **then show** *?case* **by** (*cases m*) *auto*  
**next**  
**case** (*Floor fa*)  
**then have**  $\forall i < \text{DIM}('a). \forall_F x \text{ in at } x. \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params)}$   
**by** *auto*  
**moreover**  
**have**  $iD: i < \text{DIM}('a) \implies \text{isDERIV } i \text{ fa (list-of-eucl } x \text{ @ params)}$   
 $\text{interpret-floatarith } fa \text{ (list-of-eucl } x \text{ @ params)} \notin \mathbb{Z}$  **for**  $i$   
**using** *Floor.prems(1)[OF ] by force+*  
**from** *Floor* **have**  $m: \text{max-Var-floatarith } fa \leq \text{DIM}('a) + \text{length } \text{params}$  **by** *simp*  
**from** *has-derivative-continuous*[*OF interpret-floatarith-FDERIV-floatarith-append,*  
*OF iD(1) m*]  
**have**  $\text{cont: isCont } (\lambda x. \text{interpret-floatarith } fa \text{ (list-of-eucl } x \text{ @ params)}) x$  **by**

*simp*  
**let**  $?i = \lambda x. \text{interpret-floatarith } fa \text{ (list-of-eucl } x \text{ @ params)}$   
**have**  $\forall_F y \text{ in at } x. ?i y > \text{floor } (?i x) \forall_F y \text{ in at } x. ?i y < \text{ceiling } (?i x)$   
**using** *cont*  
**by** (*auto simp: isCont-def eventually-floor-less eventually-less-ceiling iD(2)*)  
**then have**  $\forall_F x \text{ in at } x. ?i x \notin \mathbb{Z}$   
**apply** *eventually-elim*  
**apply** (*auto simp: Ints-def*)  
**by** *linarith*  
**ultimately**  
**show** *?case*  
**by** (*auto elim: eventually-elim2*)  
**qed** (*fastforce intro: DIM-positive elim: eventually-elim2*)+

**lemma** *eventually-isFDERIV*:  
**assumes**  $iD: \text{isFDERIV } DIM('a) [0..<DIM('a)] \text{ fas (list-of-eucl } x \text{ @ params)}$   
**assumes**  $m: \text{max-Var-floatariths } fas \leq DIM('a::\text{executable-euclidean-space}) + \text{length } params$   
**shows**  $\forall_F (x::'a) \text{ in at } x. \text{isFDERIV } DIM('a) [0..<DIM('a)] \text{ fas (list-of-eucl } x \text{ @ params)}$   
**by** (*auto simp: isFDERIV-def all-nat-less-eq eventually-ball-finite-distrib isFDERIV-lengthD[OF iD]*)  
*intro!*: *interpret-floatarith-eventually-isDERIV[OF isFDERIV-uptD[OF iD], rule-format]*  
*max-Var-floatarith-le-max-Var-floatariths[THEN order-trans] m*

**lemma** *isFDERIV-eventually-isFDERIV*:  
**assumes**  $iD: \text{isFDERIV } DIM('a) [0..<DIM('a)] \text{ fas (list-of-eucl } x \text{ @ params)}$   
**assumes**  $m: \text{max-Var-floatariths } fas \leq DIM('a::\text{executable-euclidean-space}) + \text{length } params$   
**shows**  $\forall_F (x::'a) \text{ in at } x. \text{isFDERIV } DIM('a) [0..<DIM('a)] \text{ fas (list-of-eucl } x \text{ @ params)}$   
**by** (*rule eventually-isFDERIV (use assms in <auto simp: isFDERIV-def>)*)

**lemma** *interpret-floatarith-FDERIV-floatariths-eucl-of-env*:  
**assumes**  $iD: \text{isFDERIV } DIM('a) \text{ xs } fas \text{ vs}$   
**assumes** *fresh: freshs-floatariths (fas) ds*  
**assumes** [*simp*]:  $\text{length } ds = DIM('a)$   
 $\bigwedge i. i \in \text{set } xs \implies i < \text{length } vs \text{ distinct } xs$   
 $\bigwedge i. i \in \text{set } ds \implies i < \text{length } vs \text{ distinct } ds$   
**shows**  $((\lambda x::'a::\text{executable-euclidean-space. eucl-of-list (interpret-floatariths } fas \text{ (list-updates } xs \text{ (list-of-eucl } x \text{ vs))::'a) has-derivative } (\lambda d. \text{eucl-of-list (interpret-floatariths (FDERIV-floatariths } fas \text{ xs (map Var } ds) (list-updates } ds \text{ (list-of-eucl } d \text{ vs)))) (at (eucl-of-env } xs \text{ vs))}))$   
**by** (*subst has-derivative-componentwise-within (auto simp add: eucl-of-list-inner isFDERIV-lengthD[OF iD] intro!*: *interpret-floatarith-FDERIV-floatarith-eucl-of-env iD[THEN isFDERIV-isDERIV-D]*)

*fresh-floatariths-fresh-floatarithI fresh)*

**lemma** *interpret-floatarith-FDERIV-floatariths-append:*

**assumes** *iD: isFDERIV DIM('a) [0.. $DIM('a)$ ] fas (list-of-eucl x @ ramsch)*

**assumes** *m: max-Var-floatariths fas  $\leq$  DIM('a) + length ramsch*

**assumes** [*simp*]: *length fas = DIM('a)*

**shows** *(( $\lambda x::'a::executable-euclidean-space.$*

*eucl-of-list*

*(interpret-floatariths fas (list-of-eucl x@ramsch))::'a) has-derivative*

*( $\lambda d.$  eucl-of-list (interpret-floatariths*

*(FDERIV-floatariths fas [0.. $DIM('a)$ ] (map Var [DIM('a)+length ramsch.. $2*DIM('a)$  + length ramsch])))*

*(list-of-eucl x @ ramsch @ list-of-eucl d)))) (at x)*

**proof** –

**have** *m-nth: ia < max-Var-floatariths fas  $\implies$  ia < DIM('a) + length ramsch*

**for** *ia*

**using** *m by simp*

**have** *m-nth': ia < max-Var-floatarith (fas ! j)  $\implies$  ia < DIM('a) + length ramsch*

**if** *j < DIM('a)* **for** *j ia*

**using** *m-nth max-Var-floatariths-lessI that by auto*

**have** *(( $\lambda xa::'a.$  eucl-of-list*

*(interpret-floatariths fas*

*(list-updates [0.. $DIM('a)$ ] (list-of-eucl xa)*

*(list-of-eucl x @ ramsch @ replicate DIM('a) 0))))::'a) has-derivative*

*( $\lambda d.$  eucl-of-list*

*(interpret-floatariths*

*(FDERIV-floatariths fas [0.. $DIM('a)$ ] (map Var [length ramsch + DIM('a).. $length ramsch$  + 2 \* DIM('a)]))*

*(list-updates [length ramsch + DIM('a).. $length ramsch$  + 2 \* DIM('a)]*

*(list-of-eucl d)*

*(list-of-eucl x @ ramsch @ replicate DIM('a) 0))))*

*(at (eucl-of-env [0.. $DIM('a)$ ] (list-of-eucl x @ ramsch @ replicate DIM('a) 0)))*

**by** *(rule interpret-floatarith-FDERIV-floatariths-eucl-of-env[of*

*[0.. $DIM('a)$ ] fas list-of-eucl x@ramsch@replicate DIM('a) 0 [length ramsch+DIM('a).. $length ramsch$ +2\*DIM('a)]])*

*(auto intro!: iD[THEN isFDERIV-uptD] freshs-floatarith-max-Var-floatarithI isFDERIV-max-Var-congI[OF iD]*

*max-Var-floatarith-le-max-Var-floatariths[THEN order-trans] m[THEN order-trans]*

*freshs-floatariths-max-Var-floatarithsI simp: nth-append m add commute less-diff-conv2 m-nth)*

**moreover have** *interpret-floatariths fas (list-updates [0.. $DIM('a)$ ] (list-of-eucl xa) (list-of-eucl x @ ramsch @ replicate DIM('a) 0)) =*

*interpret-floatariths fas (list-of-eucl xa @ ramsch) for xa::'a*

**apply** *(auto intro!: nth-equalityI interpret-floatarith-max-Var-cong simp: )*

**apply** *(auto simp: list-updates-nth nth-append dest: m-nth')*

**done**

**moreover have**

```

    (list-updates [DIM('a) + length ramsch..<length ramsch + 2 * DIM('a)]
      (list-of-eucl d)
      (list-of-eucl x @ ramsch @ replicate DIM('a) 0)) =
    (list-of-eucl x @ ramsch @ list-of-eucl d) for d::'a
  by (auto simp: intro!: nth-equalityI simp: list-updates-nth nth-append)
  moreover have (eucl-of-env [0..<DIM('a)] (list-of-eucl x @ ramsch @ replicate
    DIM('a) 0)) = x
  by (auto intro!: euclidean-eqI[where 'a='a] simp: eucl-of-env-def eucl-of-list-inner
    nth-append)
  ultimately show ?thesis by (simp add: add.commute)
qed

```

```

lemma interpret-floatarith-FDERIV-floatariths:
  assumes iD: isFDERIV DIM('a) [0..<DIM('a)] fas (list-of-eucl x)
  assumes m: max-Var-floatariths fas ≤ DIM('a)
  assumes [simp]: length fas = DIM('a)
  shows ((λx::'a::executable-euclidean-space.
    eucl-of-list
    (interpret-floatariths fas (list-of-eucl x))::'a) has-derivative
    (λd. eucl-of-list (interpret-floatariths
      (FDERIV-floatariths fas [0..<DIM('a)] (map Var [DIM('a)..<2*DIM('a)]))
      (list-of-eucl x @ list-of-eucl d)))) (at x)
  using interpret-floatarith-FDERIV-floatariths-append[where ramsch=Nil, sim-
    plified, OF assms]
  by simp

```

```

lemma continuous-on-min[continuous-intros]:
  fixes f g :: 'a::topological-space ⇒ 'b::linorder-topology
  shows continuous-on A f ⇒ continuous-on A g ⇒ continuous-on A (λx. min
    (f x) (g x))
  by (auto simp: continuous-on-def intro!: tendsto-min)

```

**lemmas** [continuous-intros] = continuous-on-max

```

lemma continuous-on-if-const[continuous-intros]:
  continuous-on s f ⇒ continuous-on s g ⇒ continuous-on s (λx. if p then f x
    else g x)
  by (cases p) auto

```

```

lemma continuous-on-floatarith:
  assumes continuous-on-floatarith fa length xs = DIM('a) distinct xs
  shows continuous-on UNIV (λx. interpret-floatarith fa (list-updates xs (list-of-eucl
    (x::'a::executable-euclidean-space)) vs))
  using assms
  by (induction fa)
  (auto intro!: continuous-intros split: if-splits simp: list-updates-nth list-of-eucl-nth-if)

```

```

fun open-form :: form ⇒ bool where
  open-form (Bound x a b f) = False |
  open-form (Assign x a f) = False |

```

$\text{open-form } (\text{Less } a \ b) \longleftrightarrow \text{continuous-on-floatarith } a \ \wedge \ \text{continuous-on-floatarith } b \mid$   
 $\text{open-form } (\text{LessEqual } a \ b) = \text{False} \mid$   
 $\text{open-form } (\text{AtLeastAtMost } x \ a \ b) = \text{False} \mid$   
 $\text{open-form } (\text{Conj } f \ g) \longleftrightarrow \text{open-form } f \ \wedge \ \text{open-form } g \mid$   
 $\text{open-form } (\text{Disj } f \ g) \longleftrightarrow \text{open-form } f \ \wedge \ \text{open-form } g$

**lemma** *open-form*:

**assumes**  $\text{open-form } f \ \text{length } xs = \text{DIM}('a::\text{executable-euclidean-space}) \ \text{distinct } xs$   
**shows**  $\text{open } (\text{Collect } (\lambda x::'a. \text{interpret-form } f \ (\text{list-updates } xs \ (\text{list-of-eucl } x) \ vs)))$   
**using** *assms*  
**by** (*induction f*) (*auto intro!*: *open-Collect-less continuous-on-floatarith open-Collect-conj open-Collect-disj*)

**primrec** *isnFDERIV* **where**

$\text{isnFDERIV } N \ \text{fas } xs \ ds \ vs \ 0 = \text{True}$   
 $\mid \text{isnFDERIV } N \ \text{fas } xs \ ds \ vs \ (\text{Suc } n) \longleftrightarrow$   
 $\text{isFDERIV } N \ xs \ (\text{FDERIV-n-floatariths } \text{fas } xs \ (\text{map } \text{Var } ds) \ n) \ vs \ \wedge$   
 $\text{isnFDERIV } N \ \text{fas } xs \ ds \ vs \ n$

**lemma** *one-add-square-eq-0*:  $1 + (x)^2 \neq (0::\text{real})$

**by** (*sos* ( $(R < 1 + (([ \sim 1 ] * A = 0) + (R < 1 * (R < 1 * [x] \wedge 2))))$ ))

**lemma** *isDERIV-fold-const-fa*[*intro*]:

**assumes**  $\text{isDERIV } x \ \text{fa } vs$   
**shows**  $\text{isDERIV } x \ (\text{fold-const-fa } \text{fa}) \ vs$   
**using** *assms*  
**apply** (*induction fa*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits option.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits option.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal for**  $\text{fa } n$   
**by** (*cases n*) (*auto simp: fold-const-fa.simps split: floatarith.splits nat.splits*)  
**subgoal**  
**by** (*auto simp: fold-const-fa.simps split: floatarith.splits*) (*subst (asm) fold-const-fa[symmetric], force*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**subgoal by** (*auto simp: fold-const-fa.simps split: floatarith.splits*)  
**done**

**lemma** *isDERIV-fold-const-fa-minus*[intro]:  
**assumes** *isDERIV x (fold-const-fa fa) vs*  
**shows** *isDERIV x (fold-const-fa (Minus fa)) vs*  
**using** *assms*  
**by** (*induction fa*) (*auto simp: fold-const-fa.simps split: floatarith.splits*)

**lemma** *isDERIV-fold-const-fa-plus*[intro]:  
**assumes** *isDERIV x (fold-const-fa fa) vs*  
**assumes** *isDERIV x (fold-const-fa fb) vs*  
**shows** *isDERIV x (fold-const-fa (Add fa fb)) vs*  
**using** *assms*  
**by** (*induction fa*)  
(*auto simp: fold-const-fa.simps*  
*split: floatarith.splits option.splits*)

**lemma** *isDERIV-fold-const-fa-mult*[intro]:  
**assumes** *isDERIV x (fold-const-fa fa) vs*  
**assumes** *isDERIV x (fold-const-fa fb) vs*  
**shows** *isDERIV x (fold-const-fa (Mult fa fb)) vs*  
**using** *assms*  
**by** (*induction fa*)  
(*auto simp: fold-const-fa.simps*  
*split: floatarith.splits option.splits*)

**lemma** *isDERIV-fold-const-fa-power*[intro]:  
**assumes** *isDERIV x (fold-const-fa fa) vs*  
**shows** *isDERIV x (fold-const-fa (fa  $\hat{e}$  n)) vs*  
**apply** (*cases n, simp add: fold-const-fa.simps split: floatarith.splits*)  
**using** *assms*  
**by** (*induction fa*)  
(*auto simp: fold-const-fa.simps split: floatarith.splits option.splits*)

**lemma** *isDERIV-fold-const-fa-inverse*[intro]:  
**assumes** *isDERIV x (fold-const-fa fa) vs*  
**assumes** *interpret-floatarith fa vs  $\neq 0$*   
**shows** *isDERIV x (fold-const-fa (Inverse fa)) vs*  
**using** *assms*  
**by** (*simp add: fold-const-fa.simps*)

**lemma** *add-square-ne-zero*[simp]: (*y::'a::linordered-idom*)  $> 0 \implies y + x^2 \neq 0$   
**by** *auto (metis less-add-same-cancel2 power2-less-0)*

**lemma** *isDERIV-FDERIV-floatarith*:  
**assumes** *isDERIV x fa vs  $\bigwedge i. i < \text{length } ds \implies \text{isDERIV } x (ds ! i) \text{ vs}$*   
**assumes** [*simp*]: *length xs = length ds*  
**shows** *isDERIV x (FDERIV-floatarith fa xs ds) vs*  
**using** *assms*  
**apply** (*induction fa*)



**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal for** *fa n* **by** (*cases n*) (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**subgoal by** (*auto simp: FDERIV-floatarith-def isDERIV-inner-iff*)  
**done**

**lemma** *isDERIV-FDERIV-floatariths*:

**assumes** *isFDERIV N xs fas vs isFDERIV N xs ds vs* **and** [*simp*]: *length fas = length ds*  
**shows** *isFDERIV N xs (FDERIV-floatariths fas xs ds) vs*  
**using** *assms*  
**by** (*auto simp: isFDERIV-def FDERIV-floatariths-def intro!: isDERIV-FDERIV-floatarith*)

**lemma** *isFDERIV-imp-isFDERIV-FDERIV-n*:

**assumes** *length fas = length ds*  
**shows** *isFDERIV N xs fas vs  $\implies$  isFDERIV N xs ds vs  $\implies$  isFDERIV N xs (FDERIV-n-floatariths fas xs ds n) vs*  
**using** *assms*  
**by** (*induction n*) (*auto intro!: isDERIV-FDERIV-floatariths*)

**lemma** *isFDERIV-map-Var*:

**assumes** [*simp*]: *length ds = N length xs = N*  
**shows** *isFDERIV N xs (map Var ds) vs*  
**by** (*auto simp: isFDERIV-def*)

**theorem** *isFDERIV-imp-isnFDERIV*:

**assumes** *isFDERIV N xs fas vs* **and** [*simp*]: *length fas = N length xs = N length ds = N*  
**shows** *isnFDERIV N fas xs ds vs n*  
**using** *assms*  
**by** (*induction n*) (*auto intro!: isFDERIV-imp-isFDERIV-FDERIV-n isFDERIV-map-Var*)

**lemma** *eventually-isnFDERIV*:

**assumes** *iD: isnFDERIV DIM('a) fas [0..*DIM('a)*] [*DIM('a)*..*2\*DIM('a)*] (list-of-eucl *x* @ list-of-eucl (*d::'a*)) *n**

**assumes**  $m$ :  $\text{max-Var-floatariths } fas \leq 2 * \text{DIM}('a::\text{executable-euclidean-space})$   
**shows**  $\forall_F (x::'a)$  in at  $x$ .  $\text{isnFDERIV } \text{DIM}('a) \text{ fas } [0..<\text{DIM}('a)] [\text{DIM}('a)..<2*\text{DIM}('a)]$   
 $(\text{list-of-eucl } x @ \text{list-of-eucl } d) \ n$   
**using**  $iD$   
**proof** ( $\text{induction } n$ )  
**case** ( $\text{Suc } n$ )  
**then have**  $1$ :  $\forall_F x$  in at  $x$ .  $\text{isnFDERIV } \text{DIM}('a) \text{ fas } [0..<\text{DIM}('a)] [\text{DIM}('a)..<2$   
 $* \text{DIM}('a)] (\text{list-of-eucl } x @ \text{list-of-eucl } d) \ n$   
**and**  $2$ :  $\text{isFDERIV } \text{DIM}('a) [0..<\text{DIM}('a)] (\text{FDERIV-n-floatariths } fas [0..<\text{DIM}('a)]$   
 $(\text{map } \text{Var } [\text{DIM}('a)..<2 * \text{DIM}('a)]) \ n$   
 $(\text{list-of-eucl } x @ \text{list-of-eucl } d)$   
**by**  $\text{simp-all}$   
**have**  $\text{max-Var-floatariths } (\text{FDERIV-n-floatariths } fas [0..<\text{DIM}('a)] (\text{map } \text{Var}$   
 $[\text{DIM}('a)..<2 * \text{DIM}('a)]) \ n) \leq$   
 $\text{DIM}('a) + \text{length } (\text{list-of-eucl } d)$   
**by** ( $\text{auto intro!}$ :  $\text{max-Var-floatarith-FDERIV-n-floatariths}[\text{THEN } \text{order-trans}]$   
 $m[\text{THEN } \text{order-trans}]$ )  
**from**  $\text{eventually-isFDERIV}[\text{OF } 2 \text{ this}] \ 1$   
**show**  $?case$   
**by**  $\text{eventually-elim simp}$   
**qed**  $\text{simp}$

**lemma**  $\text{isFDERIV-open}$ :

**assumes**  $\text{max-Var-floatariths } fas \leq \text{DIM}('a)$   
**shows**  $\text{open } \{x::'a. \text{isFDERIV } \text{DIM}('a::\text{executable-euclidean-space}) [0..<\text{DIM}('a)]$   
 $fas (\text{list-of-eucl } x)\}$   
 $(\text{is } \text{open } (\text{Collect } ?s))$   
**proof** ( $\text{safe intro!}$ :  $\text{topological-space-class.openI}$ )  
**fix**  $x::'a$  **assume**  $x$ :  $?s \ x$   
**with**  $\text{eventually-isFDERIV}[\text{where } 'a='a, \text{ of } fas \ x \ \text{Nil}]$   
**have**  $\forall_F x$  in at  $x$ .  $x \in \text{Collect } ?s$   
**by** ( $\text{auto simp}$ :  $\text{assms}$ )  
**then obtain**  $S$  **where**  $\text{open } S \ x \in S$   
 $(\forall xa \in S. xa \neq x \longrightarrow ?s \ xa)$   
**unfolding**  $\text{eventually-at-topological}$   
**by**  $\text{auto}$   
**with**  $x$  **show**  $\exists T. \text{open } T \wedge x \in T \wedge T \subseteq \text{Collect } ?s$   
**by** ( $\text{auto intro!}$ :  $\text{exI}[\text{where } x=S]$ )  
**qed**

**lemma**  $\text{interpret-floatarith-FDERIV-floatarith-eq}$ :

**assumes**  $[\text{simp}]$ :  $\text{length } xs = \text{DIM}('a::\text{executable-euclidean-space})$   $\text{length } ds =$   
 $\text{DIM}('a)$   
**shows**  $\text{interpret-floatarith } (\text{FDERIV-floatarith } fa \ xs \ ds) \ vs =$   
 $\text{einterpret } (\text{map } (\lambda x. \text{DERIV-floatarith } x \ fa) \ xs) \ vs \cdot (\text{einterpret } ds \ vs::'a)$   
**by** ( $\text{auto simp}$ :  $\text{FDERIV-floatarith-def interpret-floatarith-inner-floatariths}$ )

**lemma**

$\text{interpret-floatariths-FDERIV-floatariths-cong}$ :

**assumes** [*simp*]:  $\text{length } d1s = \text{DIM}('a::\text{executable-euclidean-space})$   $\text{length } d2s = \text{DIM}('a)$   $\text{length } fas1 = \text{length } fas2$   
**assumes** *fresh1*: *freshs-floatariths* *fas1* *d1s*  
**assumes** *fresh2*: *freshs-floatariths* *fas2* *d2s*  
**assumes** *eq1*:  $\bigwedge i. i < \text{length } fas1 \implies \text{interpret-floatariths } (\text{map } (\lambda x. \text{DERIV-floatarith } x (fas1 ! i)) [0..\text{DIM}('a)]) xs1 = \text{interpret-floatariths } (\text{map } (\lambda x. \text{DERIV-floatarith } x (fas2 ! i)) [0..\text{DIM}('a)]) xs2$   
**assumes** *eq2*:  $\bigwedge i. i < \text{DIM}('a) \implies xs1 ! (d1s ! i) = xs2 ! (d2s ! i)$   
**shows** *interpret-floatariths* (*FDERIV-floatariths* *fas1* [ $0..\text{DIM}('a)$ ]) (*map floatarith. Var* *d1s*) *xs1* = *interpret-floatariths* (*FDERIV-floatariths* *fas2* [ $0..\text{DIM}('a)$ ]) (*map floatarith. Var* *d2s*) *xs2*  
**proof** –  
**note** *eq1*  
**moreover have** *interpret-floatariths* (*map Var* *d1s*) (*xs1*) = *interpret-floatariths* (*map Var* *d2s*) (*xs2*)  
**by** (*auto intro!*: *nth-equalityI* *eq2*)  
**ultimately**  
**show** *?thesis*  
**by** (*auto intro!*: *nth-equalityI simp: interpret-floatarith-FDERIV-floatarith-eq*)  
**qed**

**lemma** *subst-floatarith-Var-DERIV-floatarith*:  
**assumes**  $\bigwedge x. x = n \longleftrightarrow s x = n$   
**shows** *subst-floatarith* ( $\lambda x. \text{Var } (s x)$ ) (*DERIV-floatarith* *n* *fa*) = *DERIV-floatarith* *n* (*subst-floatarith* ( $\lambda x. \text{Var } (s x)$ ) *fa*)  
**using** *assms*  
**proof** (*induction* *fa*)  
**case** (*Power* *fa* *n*)  
**then show** *?case* **by** (*cases* *n*) *auto*  
**qed** *force+*

**lemma** *subst-floatarith-inner-floatariths*[*simp*]:  
**assumes**  $\text{length } fs = \text{length } gs$   
**shows** *subst-floatarith* *s* (*inner-floatariths* *fs* *gs*) = *inner-floatariths* (*map* (*subst-floatarith* *s*) *fs*) (*map* (*subst-floatarith* *s*) *gs*)  
**using** *assms*  
**by** (*induction rule: list-induct2*) *auto*

**fun-cases** *subst-floatarith-Num*: *subst-floatarith* *s* *fa* = *Num* *y*  
**and** *subst-floatarith-Add*: *subst-floatarith* *s* *fa* = *Add* *x* *y*  
**and** *subst-floatarith-Minus*: *subst-floatarith* *s* *fa* = *Minus* *y*

**lemma** *Num-eq-subst-Var*[*simp*]: *Num* *x* = *subst-floatarith* ( $\lambda x. \text{Var } (s x)$ ) *fa*  $\longleftrightarrow$  *fa* = *Num* *x*  
**by** (*cases* *fa*) *auto*

**lemma** *Add-eq-subst-VarE*:

```

assumes Add fa1 fa2 = subst-floatarith ( $\lambda x. \text{Var } (s\ x)$ ) fa
obtains a1 a2 where fa = Add a1 a2 fa1 = subst-floatarith ( $\lambda x. \text{Var } (s\ x)$ ) a1
  fa2 = subst-floatarith ( $\lambda x. \text{Var } (s\ x)$ ) a2
using assms
by (cases fa) auto

lemma subst-floatarith-eq-self[simp]: subst-floatarith s f = f if max-Var-floatarith
f = 0
using that by (induction f) auto

lemma fold-const-fa-unique: False if ( $\bigwedge x. f = \text{Num } x$ )
using that[of 0] that[of 1]
by auto

lemma zero-unique: False if ( $\bigwedge x::\text{float}. x = 0$ )
using that[of 0] that[of 1]
by auto

lemma fold-const-fa-Mult-eq-NumE:
assumes fold-const-fa (Mult a b) = Num x
obtains y z where fold-const-fa a = Num y fold-const-fa b = Num z x = y * z
  | y where fold-const-fa a = Num 0 x = 0
  | y where fold-const-fa b = Num 0 x = 0
using assms
by atomize-elim (auto simp: fold-const-fa.simps split!: option.splits if-splits
  elim!: dest-Num-fa-Some dest-Num-fa-None)

lemma fold-const-fa-Add-eq-NumE:
assumes fold-const-fa (Add a b) = Num x
obtains y z where fold-const-fa a = Num y fold-const-fa b = Num z x = y + z
using assms
by atomize-elim (auto simp: fold-const-fa.simps split!: option.splits if-splits
  elim!: dest-Num-fa-Some dest-Num-fa-None)

lemma subst-floatarith-Var-fold-const-fa[symmetric]:
  fold-const-fa (subst-floatarith ( $\lambda x. \text{Var } (s\ x)$ ) fa) =
  subst-floatarith ( $\lambda x. \text{Var } (s\ x)$ ) (fold-const-fa fa)
proof (induction fa)
case (Add fa1 fa2)
then show ?case
  apply (auto simp: fold-const-fa.simps
    split!: floatarith.splits option.splits if-splits
    elim!: dest-Num-fa-Some)
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))

```

```

    apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
    apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
    apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
    apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  done
next
case (Mult fa1 fa2)
then show ?case
  apply (auto simp: fold-const-fa.simps
    split!: floatarith.splits option.splits if-splits
    elim!: dest-Num-fa-Some)
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  apply (metis Num-eq-subst-Var dest-Num-fa.simps(1) option.simps(3))
  done
next
case (Min)
then show ?case
  by (auto simp: fold-const-fa.simps split: floatarith.splits)
next
case (Max)
then show ?case
  by (auto simp: fold-const-fa.simps split: floatarith.splits)
qed (auto simp: fold-const-fa.simps
  split!: floatarith.splits option.splits if-splits
  elim!: dest-Num-fa-Some)

lemma subst-floatarith-eq-Num[simp]: (subst-floatarith ( $\lambda x. \text{Var } (s \ x)$ ) fa = Num x)  $\longleftrightarrow$  fa = Num x
  by (induction fa) auto

lemma fold-const-fa-subst-eq-Num0-iff[simp]:
  fold-const-fa (subst-floatarith ( $\lambda x. \text{Var } (s \ x)$ ) fa) = Num x  $\longleftrightarrow$  fold-const-fa fa = Num x
  unfolding subst-floatarith-Var-fold-const-fa[symmetric]
  by simp

```

**lemma** *subst-floatarith-Var-FDERIV-floatarith*:  
**assumes** *len*:  $\text{length } xs = \text{DIM}('a::\text{executable-euclidean-space})$  **and** [*simp*]:  $\text{length } ds = \text{DIM}('a)$   
**assumes** *eq*:  $\bigwedge x y. x \in \text{set } xs \implies (y = x) = (s y = x)$   
**shows**  $\text{subst-floatarith } (\lambda x. \text{Var } (s x)) (\text{FDERIV-floatarith } fa \text{ } xs \text{ } ds) =$   
 $(\text{FDERIV-floatarith } (\text{subst-floatarith } (\lambda x. \text{Var } (s x))) fa \text{ } xs \text{ } (\text{map } (\text{subst-floatarith } (\lambda x. \text{Var } (s x)))) ds))$   
**proof** –  
**have** [*simp*]:  $\bigwedge x. x \in \text{set } xs \implies \text{subst-floatarith } (\lambda x. \text{Var } (s x)) (\text{DERIV-floatarith } x \text{ } fa1) =$   
 $(\text{DERIV-floatarith } x \text{ } (\text{subst-floatarith } (\lambda x. \text{Var } (s x)) fa1))$   
**for** *fa1*  
**by** (*rule subst-floatarith-Var-DERIV-floatarith*) (*rule eq*)  
**have** *map-eq*:  $(\text{map } (\lambda xa. \text{if } xa = s x \text{ then Num } 1 \text{ else Num } 0) xs) =$   
 $(\text{map } (\lambda xa. \text{if } xa = x \text{ then Num } 1 \text{ else Num } 0) xs)$   
**for** *x*  
**apply** (*subst map-eq-conv*)  
**using** *eq[of x x]* *eq[of s x]*  
**by** *auto*  
**show** *?thesis*  
**using** *len*  
**by** (*induction fa*)  
 $(\text{auto simp: FDERIV-floatarith-def o-def if-distrib}$   
 $\text{subst-floatarith-Var-fold-const-fa fold-const-fa.simps(18) map-eq}$   
 $\text{cong: map-cong if-cong})$   
**qed**

**lemma** *subst-floatarith-Var-FDERIV-n-nth*:  
**assumes** *len*:  $\text{length } xs = \text{DIM}('a::\text{executable-euclidean-space})$  **and** [*simp*]:  $\text{length } ds = \text{DIM}('a)$   
**assumes** *eq*:  $\bigwedge x y. x \in \text{set } xs \implies (y = x) = (s y = x)$   
**assumes** [*simp*]:  $i < \text{length } fas$   
**shows**  $\text{subst-floatarith } (\lambda x. \text{Var } (s x)) (\text{FDERIV-n-floatariths } fas \text{ } xs \text{ } ds \text{ } n ! i) =$   
 $(\text{FDERIV-n-floatariths } (\text{map } (\text{subst-floatarith } (\lambda x. \text{Var } (s x)))) fas \text{ } xs \text{ } (\text{map } (\text{subst-floatarith } (\lambda x. \text{Var } (s x)))) ds) \text{ } n ! i)$   
**proof** (*induction n*)  
**case** (*Suc n*)  
**show** *?case*  
**by** (*simp add: subst-floatarith-Var-FDERIV-floatarith[OF len - eq] Suc.IH[symmetric]*)  
**qed** *simp*

**lemma** *subst-floatarith-Var-max-Var-floatarith*:  
**assumes**  $\bigwedge i. i < \text{max-Var-floatarith } fa \implies s i = i$   
**shows**  $\text{subst-floatarith } (\lambda i. \text{Var } (s i)) fa = fa$   
**using** *assms*  
**by** (*induction fa*) *auto*

**lemma** *interpret-floatarith-subst-floatarith-idem*:  
**assumes** *mv*:  $\text{max-Var-floatarith } fa \leq \text{length } vs$

**assumes** *idem*:  $\bigwedge j. j < \text{max-Var-floatarith } fa \implies vs ! s j = vs ! j$   
**shows** *interpret-floatarith* (*subst-floatarith* ( $\lambda i. \text{Var } (s i)$ ) *fa*) *vs* = *interpret-floatarith* *fa* *vs*  
**using** *assms*  
**by** (*induction fa*) *auto*

**lemma** *isDERIV-subst-Var-floatarith*:  
**assumes** *mv*:  $\text{max-Var-floatarith } fa \leq \text{length } vs$   
**assumes** *idem*:  $\bigwedge j. j < \text{max-Var-floatarith } fa \implies vs ! s j = vs ! j$   
**assumes**  $\bigwedge j. s j = i \longleftrightarrow j = i$   
**shows** *isDERIV* *i* (*subst-floatarith* ( $\lambda i. \text{Var } (s i)$ ) *fa*) *vs* = *isDERIV* *i* *fa* *vs*  
**using** *mv idem*  
**proof** (*induction fa*)  
**case** (*Power fa n*)  
**then show** ?*case* **by** (*cases n*) *auto*  
**qed** (*auto simp: interpret-floatarith-subst-floatarith-idem*)

**lemma** *isFDERIV-subst-Var-floatarith*:  
**assumes** *mv*:  $\text{max-Var-floatariths } fas \leq \text{length } vs$   
**assumes** *idem*:  $\bigwedge j. j < \text{max-Var-floatariths } fas \implies vs ! (s j) = vs ! j$   
**assumes**  $\bigwedge i j. i \in \text{set } xs \implies s j = i \longleftrightarrow j = i$   
**shows** *isFDERIV* *n* *xs* (*map* (*subst-floatarith* ( $\lambda i. \text{Var } (s i)$ )) *fas*) *vs* = *isFDERIV* *n* *xs* *fas* *vs*  
**proof** –  
**have** *mv*:  $\bigwedge i. i < \text{length } fas \implies \text{max-Var-floatarith } (fas ! i) \leq \text{length } vs$   
**apply** (*rule order-trans[OF - mv]*)  
**by** (*intro max-Var-floatarith-le-max-Var-floatariths-nth*)  
**have** *idem*:  $\bigwedge i j. i < \text{length } fas \implies j < \text{max-Var-floatarith } (fas ! i) \implies vs ! s j = vs ! j$   
**using** *idem*  
**by** (*auto simp: dest!: max-Var-floatariths-lessI*)  
**show** ?*thesis*  
**unfolding** *isFDERIV-def*  
**using** *mv idem assms(3)*  
**by** (*auto simp: isDERIV-subst-Var-floatarith*)  
**qed**

**lemma** *interpret-floatariths-append[simp]*:  
*interpret-floatariths* (*xs* @ *ys*) *vs* = *interpret-floatariths* *xs* *vs* @ *interpret-floatariths* *ys* *vs*  
**by** (*induction xs*) *auto*

**lemma** *not-fresh-inner-floatariths*:  
**assumes**  $\text{length } xs = \text{length } ys$   
**shows**  $\neg \text{fresh-floatarith } (\text{inner-floatariths } xs \ ys) \ i \longleftrightarrow \neg \text{fresh-floatariths } xs \ i \vee \neg \text{fresh-floatariths } ys \ i$   
**using** *assms*  
**by** (*induction xs ys rule: list-induct2*) *auto*

**lemma** *fresh-inner-floatariths*:  
**assumes**  $\text{length } xs = \text{length } ys$   
**shows**  $\text{fresh-floatarith } (\text{inner-floatariths } xs \ ys) \ i \longleftrightarrow \text{fresh-floatariths } xs \ i \wedge \text{fresh-floatariths } ys \ i$   
**using** *not-fresh-inner-floatariths assms by auto*

**lemma** *not-fresh-floatariths-map*:  
 $\neg \text{fresh-floatariths } (\text{map } f \ xs) \ i \longleftrightarrow (\exists x \in \text{set } xs. \neg \text{fresh-floatarith } (f \ x) \ i)$   
**by** (*induction xs*) *auto*

**lemma** *fresh-floatariths-map*:  
 $\text{fresh-floatariths } (\text{map } f \ xs) \ i \longleftrightarrow (\forall x \in \text{set } xs. \text{fresh-floatarith } (f \ x) \ i)$   
**by** (*induction xs*) *auto*

**lemma** *fresh-floatarith-fold-const-fa*:  $\text{fresh-floatarith } fa \ i \implies \text{fresh-floatarith } (\text{fold-const-fa } fa) \ i$   
**by** (*induction fa*) (*auto simp: fold-const-fa.simps split: floatarith.splits option.splits*)

**lemma** *fresh-floatarith-fold-const-fa-Add[intro!]*:  
**assumes**  $\text{fresh-floatarith } (\text{fold-const-fa } a) \ i \ \text{fresh-floatarith } (\text{fold-const-fa } b) \ i$   
**shows**  $\text{fresh-floatarith } (\text{fold-const-fa } (\text{Add } a \ b)) \ i$   
**using** *assms*  
**by** (*auto simp: fold-const-fa.simps split!: floatarith.splits option.splits*)

**lemma** *fresh-floatarith-fold-const-fa-Mult[intro!]*:  
**assumes**  $\text{fresh-floatarith } (\text{fold-const-fa } a) \ i \ \text{fresh-floatarith } (\text{fold-const-fa } b) \ i$   
**shows**  $\text{fresh-floatarith } (\text{fold-const-fa } (\text{Mult } a \ b)) \ i$   
**using** *assms*  
**by** (*auto simp: fold-const-fa.simps split!: floatarith.splits option.splits*)

**lemma** *fresh-floatarith-fold-const-fa-Minus[intro!]*:  
**assumes**  $\text{fresh-floatarith } (\text{fold-const-fa } b) \ i$   
**shows**  $\text{fresh-floatarith } (\text{fold-const-fa } (\text{Minus } b)) \ i$   
**using** *assms*  
**by** (*auto simp: fold-const-fa.simps split!: floatarith.splits*)

**lemma** *fresh-FDERIV-floatarith*:  
 $\text{fresh-floatarith } \text{ode-e } i \implies \text{fresh-floatariths } ds \ i$   
 $\implies \text{length } ds = \text{DIM}('a)$   
 $\implies \text{fresh-floatarith } (\text{FDERIV-floatarith } \text{ode-e } [0..<\text{DIM}('a)::\text{executable-euclidean-space}]) \ ds) \ i$   
**proof** (*induction ode-e*)  
**case** (*Power ode-e n*)  
**then show** *?case by (cases n) (auto simp: FDERIV-floatarith-def fresh-inner-floatariths fresh-floatariths-map fresh-floatarith-fold-const-fa)*  
**qed** (*auto simp: FDERIV-floatarith-def fresh-inner-floatariths fresh-floatariths-map fresh-floatarith-fold-const-fa*)

**lemma** *not-fresh-FDERIV-floatarith*:



$\neg$  fresh-floatarith (FDERIV-floatarith ode-e [0.. $DIM('a)::executable-euclidean-space$ ])  
 ds) i  
 $\implies$  length ds =  $DIM('a)$   
 $\implies \neg$ fresh-floatarith ode-e i  $\vee$   $\neg$ fresh-floatariths ds i  
 using fresh-FDERIV-floatarith by auto

**lemma** not-fresh-FDERIV-floatariths:

$\neg$  fresh-floatariths (FDERIV-floatariths ode-e [0.. $DIM('a)::executable-euclidean-space$ ])  
 ds) i  $\implies$   
 length ds =  $DIM('a)$   $\implies \neg$ fresh-floatariths ode-e i  $\vee$   $\neg$ fresh-floatariths ds i  
 by (induction ode-e) (auto simp: FDERIV-floatariths-def dest!: not-fresh-FDERIV-floatarith)

**lemma** isDERIV-FDERIV-floatarith-linear:

fixes x h::' $a::executable-euclidean-space$   
 assumes  $\bigwedge k. k < DIM('a) \implies isDERIV i (DERIV-floatarith k fa) xs$   
 assumes max-Var-floatarith fa  $\leq DIM('a)$   
 assumes [simp]: length xs =  $DIM('a)$  length hs =  $DIM('a)$   
 shows isDERIV i (FDERIV-floatarith fa [0.. $DIM('a)$ ] (map Var [ $DIM('a)..<2$   
 \*  $DIM('a)$ ]))  
 (xs @ hs)  
 using assms  
 apply (auto simp: FDERIV-floatarith-def isDERIV-inner-iff)  
 apply (rule isDERIV-max-Var-floatarithI) apply force  
 apply (auto simp: nth-append)  
 by (metis add-diff-inverse-nat leD max-Var-floatarith-DERIV-floatarith  
 max-Var-floatarith-fold-const-fa trans-le-add1)

**lemma**

isFDERIV-FDERIV-floatariths-linear:  
 fixes x h::' $a::executable-euclidean-space$   
 assumes  $\bigwedge i j k.$   
 $i < DIM('a) \implies$   
 $j < DIM('a) \implies k < DIM('a) \implies isDERIV i (DERIV-floatarith k (fas !$   
 j)) (xs)  
 assumes [simp]: length fas =  $DIM('a)::executable-euclidean-space$   
 assumes [simp]: length xs =  $DIM('a)$  length hs =  $DIM('a)$   
 assumes max-Var-floatariths fas  $\leq DIM('a)$   
 shows isFDERIV  $DIM('a)$  [0.. $DIM('a)::executable-euclidean-space$ ]  
 (FDERIV-floatariths fas [0.. $DIM('a)$ ] (map floatarith.Var [ $DIM('a)..<2$  \*  
 $DIM('a)$ ]))  
 (xs @ hs)  
 apply (auto simp: isFDERIV-def intro!: isDERIV-FDERIV-floatarith-linear assms)  
 using assms(5) max-Var-floatariths-lessI not-le-imp-less by fastforce

**definition** isFDERIV-approx where

isFDERIV-approx p n xs fas vs =  
 $((\forall i < n. \forall j < n. isDERIV-approx p (xs ! i) (fas ! j) vs) \wedge$  length fas = n  $\wedge$  length  
 xs = n)

**lemma** *isFDERIV-approx*:  
*bounded-by vs VS*  $\implies$  *isFDERIV-approx prec n xs fas VS*  $\implies$  *isFDERIV n xs fas vs*  
**by** (*auto simp: isFDERIV-approx-def isFDERIV-def intro!: isDERIV-approx*)

**primrec** *isnFDERIV-approx where*  
*isnFDERIV-approx p N fas xs ds vs 0 = True*  
| *isnFDERIV-approx p N fas xs ds vs (Suc n)*  $\longleftrightarrow$   
*isFDERIV-approx p N xs (FDERIV-n-floatariths fas xs (map Var ds) n) vs*  $\wedge$   
*isnFDERIV-approx p N fas xs ds vs n*

**lemma** *isnFDERIV-approx*:  
*bounded-by vs VS*  $\implies$  *isnFDERIV-approx prec N fas xs ds VS n*  $\implies$  *isnFDERIV N fas xs ds vs n*  
**by** (*induction n (auto intro!: isFDERIV-approx)*)

**fun** *plain-floatarith::nat  $\Rightarrow$  floatarith  $\Rightarrow$  bool where*  
*plain-floatarith N (floatarith.Add a b)*  $\longleftrightarrow$  *plain-floatarith N a*  $\wedge$  *plain-floatarith N b*  
| *plain-floatarith N (floatarith.Mult a b)*  $\longleftrightarrow$  *plain-floatarith N a*  $\wedge$  *plain-floatarith N b*  
| *plain-floatarith N (floatarith.Minus a)*  $\longleftrightarrow$  *plain-floatarith N a*  
| *plain-floatarith N (floatarith.Pi)*  $\longleftrightarrow$  *True*  
| *plain-floatarith N (floatarith.Num n)*  $\longleftrightarrow$  *True*  
| *plain-floatarith N (floatarith.Var i)*  $\longleftrightarrow$  *i < N*  
| *plain-floatarith N (floatarith.Max a b)*  $\longleftrightarrow$  *plain-floatarith N a*  $\wedge$  *plain-floatarith N b*  
| *plain-floatarith N (floatarith.Min a b)*  $\longleftrightarrow$  *plain-floatarith N a*  $\wedge$  *plain-floatarith N b*  
| *plain-floatarith N (floatarith.Power a n)*  $\longleftrightarrow$  *plain-floatarith N a*  
| *plain-floatarith N (floatarith.Cos a)*  $\longleftrightarrow$  *False* — *TODO: should be plain!*  
| *plain-floatarith N (floatarith.Arctan a)*  $\longleftrightarrow$  *False* — *TODO: should be plain!*  
| *plain-floatarith N (floatarith.Abs a)*  $\longleftrightarrow$  *plain-floatarith N a*  
| *plain-floatarith N (floatarith.Exp a)*  $\longleftrightarrow$  *False* — *TODO: should be plain!*  
| *plain-floatarith N (floatarith.Sqrt a)*  $\longleftrightarrow$  *False* — *TODO: should be plain!*  
| *plain-floatarith N (floatarith.Floor a)*  $\longleftrightarrow$  *plain-floatarith N a*  
  
| *plain-floatarith N (floatarith.Powr a b)*  $\longleftrightarrow$  *False*  
| *plain-floatarith N (floatarith.Inverse a)*  $\longleftrightarrow$  *False*  
| *plain-floatarith N (floatarith.Ln a)*  $\longleftrightarrow$  *False*

**lemma** *plain-floatarith-approx-not-None*:  
**assumes** *plain-floatarith N fa N  $\leq$  length XS  $\wedge$   $\bigwedge i. i < N \implies XS ! i \neq None$*   
**shows** *approx p fa XS  $\neq$  None*  
**using** *assms*  
**by** (*induction fa*)  
(*auto simp: Let-def split-beta' prod-eq-iff approx.simps*)

**definition**  $Rad\text{-of } w = w * (Pi / Num\ 180)$   
**lemma**  $interpret\text{-Rad-of}[simp]: interpret\text{-floatarith } (Rad\text{-of } w) xs = rad\text{-of } (interpret\text{-floatarith } w xs)$   
**by** (*auto simp: Rad-of-def rad-of-def*)

**definition**  $Deg\text{-of } w = Num\ 180 * w / Pi$   
**lemma**  $interpret\text{-Deg-of}[simp]: interpret\text{-floatarith } (Deg\text{-of } w) xs = deg\text{-of } (interpret\text{-floatarith } w xs)$   
**by** (*auto simp: Deg-of-def deg-of-def inverse-eq-divide*)

**unbundle** *no floatarith-syntax*

**end**

## 4 Straight Line Programs

**theory** *Straight-Line-Program*

**imports**

*Floatarith-Expression*

*Deriving.Derive*

*HOL-Library.Monad-Syntax*

*HOL-Library.RBT-Mapping*

**begin**

**unbundle** *floatarith-syntax*

**derive** (*linorder*) *compare-order float*

**derive** *linorder floatarith*

### 4.1 Definition

**type-synonym**  $slp = floatarith\ list$

**primrec**  $interpret\text{-slp}::slp \Rightarrow real\ list \Rightarrow real\ list$  **where**

$interpret\text{-slp } [] = (\lambda xs. xs)$

$| interpret\text{-slp } (ea \# eas) = (\lambda xs. interpret\text{-slp } eas (interpret\text{-floatarith } ea xs\#xs))$

### 4.2 Reification as straight line program (with common subexpression elimination)

**definition**  $slp\text{-index } vs\ i = (length\ vs - Suc\ i)$

**definition**  $slp\text{-index-lookup } vs\ M\ a = slp\text{-index } vs\ (the\ (Mapping.lookup\ M\ a))$

**definition**

$slp\text{-of-fa-bin } Binop\ a\ b\ M\ slp\ M2\ slp2 =$

$(case\ Mapping.lookup\ M\ (Binop\ a\ b)\ of$

$\text{Some } i \Rightarrow (\text{Mapping.update } (\text{Binop } a \ b) \ (\text{length } \text{slp}) \ M, \ \text{slp}@[\text{Var } (\text{slp-index } \text{slp } i)])$   
 $| \text{None} \Rightarrow (\text{Mapping.update } (\text{Binop } a \ b) \ (\text{length } \text{slp2}) \ M2,$   
 $\quad \text{slp2}@[\text{Binop } (\text{Var } (\text{slp-index-lookup } \text{slp2 } M2 \ a)) \ (\text{Var } (\text{slp-index-lookup } \text{slp2 } M2 \ b))])$ )

**definition**

$\text{slp-of-fa-un } \text{Unop } a \ M \ \text{slp} \ M1 \ \text{slp1} =$   
 $(\text{case } \text{Mapping.lookup } M \ (\text{Unop } a) \ \text{of}$   
 $\quad \text{Some } i \Rightarrow (\text{Mapping.update } (\text{Unop } a) \ (\text{length } \text{slp}) \ M, \ \text{slp}@[\text{Var } (\text{slp-index } \text{slp } i)])$   
 $| \text{None} \Rightarrow (\text{Mapping.update } (\text{Unop } a) \ (\text{length } \text{slp1}) \ M1,$   
 $\quad \text{slp1}@[\text{Unop } (\text{Var } (\text{slp-index-lookup } \text{slp1 } M1 \ a))])$ )

**definition**

$\text{slp-of-fa-cnst } \text{Const } \text{Const}' \ M \ \text{vs} =$   
 $(\text{Mapping.update } \text{Const} \ (\text{length } \text{vs}) \ M,$   
 $\quad \text{vs} \ @ \ [\text{case } \text{Mapping.lookup } M \ \text{Const} \ \text{of } \text{Some } i \Rightarrow \text{Var } (\text{slp-index } \text{vs } i) \ | \ \text{None}$   
 $\Rightarrow \text{Const}'])$

**fun**  $\text{slp-of-fa} :: \text{floatarith} \Rightarrow (\text{floatarith}, \text{nat}) \ \text{mapping} \Rightarrow \text{floatarith list} \Rightarrow$   
 $((\text{floatarith}, \text{nat}) \ \text{mapping} \times \text{floatarith list}) \ \text{where}$   
 $\text{slp-of-fa } (\text{Add } a \ b) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp}; \ (M2, \ \text{slp2}) = \text{slp-of-fa } b \ M1 \ \text{slp1} \ \text{in}$   
 $\quad \text{slp-of-fa-bin } \text{Add } a \ b \ M \ \text{slp} \ M2 \ \text{slp2})$   
 $| \text{slp-of-fa } (\text{Mult } a \ b) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp}; \ (M2, \ \text{slp2}) = \text{slp-of-fa } b \ M1 \ \text{slp1} \ \text{in}$   
 $\quad \text{slp-of-fa-bin } \text{Mult } a \ b \ M \ \text{slp} \ M2 \ \text{slp2})$   
 $| \text{slp-of-fa } (\text{Min } a \ b) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp}; \ (M2, \ \text{slp2}) = \text{slp-of-fa } b \ M1 \ \text{slp1} \ \text{in}$   
 $\quad \text{slp-of-fa-bin } \text{Min } a \ b \ M \ \text{slp} \ M2 \ \text{slp2})$   
 $| \text{slp-of-fa } (\text{Max } a \ b) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp}; \ (M2, \ \text{slp2}) = \text{slp-of-fa } b \ M1 \ \text{slp1} \ \text{in}$   
 $\quad \text{slp-of-fa-bin } \text{Max } a \ b \ M \ \text{slp} \ M2 \ \text{slp2})$   
 $| \text{slp-of-fa } (\text{Powr } a \ b) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp}; \ (M2, \ \text{slp2}) = \text{slp-of-fa } b \ M1 \ \text{slp1} \ \text{in}$   
 $\quad \text{slp-of-fa-bin } \text{Powr } a \ b \ M \ \text{slp} \ M2 \ \text{slp2})$   
 $| \text{slp-of-fa } (\text{Inverse } a) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in } \text{slp-of-fa-un } \text{Inverse } a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (\text{Cos } a) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in } \text{slp-of-fa-un } \text{Cos } a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (\text{Arctan } a) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in } \text{slp-of-fa-un } \text{Arctan } a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (\text{Abs } a) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in } \text{slp-of-fa-un } \text{Abs } a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (\text{Sqrt } a) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in } \text{slp-of-fa-un } \text{Sqrt } a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (\text{Exp } a) \ M \ \text{slp} =$   
 $\quad (\text{let } (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in } \text{slp-of-fa-un } \text{Exp } a \ M \ \text{slp} \ M1 \ \text{slp1})$

$| \text{slp-of-fa } (Ln \ a) \ M \ \text{slp} =$   
 $(let \ (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in} \ \text{slp-of-fa-un } Ln \ a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (Minus \ a) \ M \ \text{slp} =$   
 $(let \ (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in} \ \text{slp-of-fa-un } Minus \ a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (Floor \ a) \ M \ \text{slp} =$   
 $(let \ (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in} \ \text{slp-of-fa-un } Floor \ a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } (Power \ a \ n) \ M \ \text{slp} =$   
 $(let \ (M1, \ \text{slp1}) = \text{slp-of-fa } a \ M \ \text{slp} \ \text{in} \ \text{slp-of-fa-un } (\lambda a. \ \text{Power } a \ n) \ a \ M \ \text{slp} \ M1 \ \text{slp1})$   
 $| \text{slp-of-fa } Pi \ M \ \text{slp} = \text{slp-of-fa-cnst } Pi \ Pi \ M \ \text{slp}$   
 $| \text{slp-of-fa } (Var \ v) \ M \ \text{slp} = \text{slp-of-fa-cnst } (Var \ v) \ (Var \ (v + \text{length } \text{slp})) \ M \ \text{slp}$   
 $| \text{slp-of-fa } (Num \ n) \ M \ \text{slp} = \text{slp-of-fa-cnst } (Num \ n) \ (Num \ n) \ M \ \text{slp}$

**lemma** *interpret-slp-snoc*[simp]:

$\text{interpret-slp } (\text{slp} \ @ \ [fa]) \ xs = \text{interpret-floatarith } fa \ (\text{interpret-slp } \text{slp} \ xs) \# \text{interpret-slp } \text{slp} \ xs$

**by** (*induction slp arbitrary: fa xs*) *auto*

**lemma**

*binop-slp-of-fa-induction-step:*

**assumes**

*Binop-IH1:*

$\bigwedge M \ \text{slp} \ M' \ \text{slp}'. \ \text{slp-of-fa } fa1 \ M \ \text{slp} = (M', \ \text{slp}') \implies$

$(\bigwedge f. \ f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M) \implies$

$(\bigwedge f. \ f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M \ f) < \text{length } \text{slp}) \implies$

$(\bigwedge f. \ f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp} \ xs \ ! \ \text{slp-index-lookup } \text{slp} \ M \ f = \text{interpret-floatarith } f \ xs) \implies$

$\text{subterms } fa1 \subseteq \text{Mapping.keys } M' \wedge$

$\text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$

$(\forall f \in \text{Mapping.keys } M'. \ \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$

$\text{the } (\text{Mapping.lookup } M' \ f) < \text{length } \text{slp}' \wedge$

$\text{interpret-slp } \text{slp}' \ xs \ ! \ \text{slp-index-lookup } \text{slp}' \ M' \ f = \text{interpret-floatarith } f \ xs)$

**and**

*Binop-IH2:*

$\bigwedge M \ \text{slp} \ M' \ \text{slp}'. \ \text{slp-of-fa } fa2 \ M \ \text{slp} = (M', \ \text{slp}') \implies$

$(\bigwedge f. \ f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M) \implies$

$(\bigwedge f. \ f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M \ f) < \text{length } \text{slp}) \implies$

$(\bigwedge f. \ f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp} \ xs \ ! \ \text{slp-index-lookup } \text{slp} \ M \ f = \text{interpret-floatarith } f \ xs) \implies$

$\text{subterms } fa2 \subseteq \text{Mapping.keys } M' \wedge$

$\text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$

$(\forall f \in \text{Mapping.keys } M'. \ \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$

$\text{the } (\text{Mapping.lookup } M' \ f) < \text{length } \text{slp}' \wedge$

$\text{interpret-slp } \text{slp}' \ xs \ ! \ \text{slp-index-lookup } \text{slp}' \ M' \ f = \text{interpret-floatarith } f \ xs)$

**and** *Binop-prems:*

$(\text{case } \text{slp-of-fa } fa1 \ M \ \text{slp} \ \text{of}$

$(M1, \ \text{slp1}) \Rightarrow$

$\text{case } \text{slp-of-fa } fa2 \ M1 \ \text{slp1} \ \text{of } (x, \ xa) \Rightarrow \text{slp-of-fa-bin } Binop \ fa1 \ fa2 \ M \ \text{slp} \ x \ xa) = (M', \ \text{slp}')$

$\wedge f. f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M$   
 $\wedge f. f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M f) < \text{length } \text{slp}$   
 $\wedge f. f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp } xs \ ! \ \text{slp-index-lookup } \text{slp } M f =$   
 $\text{interpret-floatarith } f \ xs$   
**assumes**  $\text{subterms-Binop}[simp]$ :  
 $\wedge a \ b. \text{subterms } (\text{Binop } a \ b) = \text{insert } (\text{Binop } a \ b) (\text{subterms } a \cup \text{subterms } b)$   
**assumes**  $\text{interpret-Binop}[simp]$ :  
 $\wedge a \ b \ xs. \text{interpret-floatarith } (\text{Binop } a \ b) \ xs = \text{binop } (\text{interpret-floatarith } a \ xs)$   
 $(\text{interpret-floatarith } b \ xs)$   
**shows**  $\text{insert } (\text{Binop } fa1 \ fa2) (\text{subterms } fa1 \cup \text{subterms } fa2) \subseteq \text{Mapping.keys } M'$   
 $\wedge$   
 $\text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$   
 $(\forall f \in \text{Mapping.keys } M'. \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$   
 $\text{the } (\text{Mapping.lookup } M' f) < \text{length } \text{slp}' \wedge$   
 $\text{interpret-slp } \text{slp}' \ xs \ ! \ \text{slp-index-lookup } \text{slp}' M' f = \text{interpret-floatarith } f \ xs)$   
**proof** –  
**from**  $\text{Binop-prems}$   
**obtain**  $M1 \ \text{slp1} \ M2 \ \text{slp2}$  **where** \*:  
 $\text{slp-of-fa } fa1 \ M \ \text{slp} = (M1, \ \text{slp1})$   
 $\text{slp-of-fa } fa2 \ M1 \ \text{slp1} = (M2, \ \text{slp2})$   
 $\text{slp-of-fa-bin } \text{Binop } fa1 \ fa2 \ M \ \text{slp} \ M2 \ \text{slp2} = (M', \ \text{slp}')$   
**by**  $(\text{auto split: prod.splits})$   
**from**  $\text{Binop-IH1}[OF \ *(1) \ \text{Binop-prems}(2) \ \text{Binop-prems}(3) \ \text{Binop-prems}(4), \ \text{sim-}$   
 $\text{plified}]$   
**have**  $IH1$ :  
 $f \in \text{subterms } fa1 \implies f \in \text{Mapping.keys } M1$   
 $f \in \text{Mapping.keys } M \implies f \in \text{Mapping.keys } M1$   
 $f \in \text{Mapping.keys } M1 \implies \text{subterms } f \subseteq \text{Mapping.keys } M1$   
 $f \in \text{Mapping.keys } M1 \implies \text{the } (\text{Mapping.lookup } M1 f) < \text{length } \text{slp1}$   
 $f \in \text{Mapping.keys } M1 \implies \text{interpret-slp } \text{slp1 } xs \ ! \ \text{slp-index-lookup } \text{slp1 } M1 f =$   
 $\text{interpret-floatarith } f \ xs$   
**for**  $f$   
**by**  $(\text{auto simp: subset-iff})$   
**from**  $\text{Binop-IH2}[OF \ *(2) \ IH1(3) \ IH1(4) \ IH1(5)]$   
**have**  $IH2$ :  
 $f \in \text{subterms } fa2 \implies f \in \text{Mapping.keys } M2$   
 $f \in \text{Mapping.keys } M1 \implies f \in \text{Mapping.keys } M2$   
 $f \in \text{Mapping.keys } M2 \implies \text{subterms } f \subseteq \text{Mapping.keys } M2$   
 $f \in \text{Mapping.keys } M2 \implies \text{the } (\text{Mapping.lookup } M2 f) < \text{length } \text{slp2}$   
 $f \in \text{Mapping.keys } M2 \implies \text{interpret-slp } \text{slp2 } xs \ ! \ \text{slp-index-lookup } \text{slp2 } M2 f =$   
 $\text{interpret-floatarith } f \ xs$   
**for**  $f$   
**by**  $(\text{auto simp: subset-iff})$   
**show**  $?thesis$   
**proof**  $(\text{cases } \text{Mapping.lookup } M \ (\text{Binop } fa1 \ fa2))$   
**case**  $\text{None}$   
**then have**  $M'$ :  $M' = \text{Mapping.update } (\text{Binop } fa1 \ fa2) (\text{length } \text{slp2}) \ M2$   
**and**  $\text{slp}'$ :  $\text{slp}' = \text{slp2} \ @ \ [\text{Binop } (\text{Var } (\text{slp-index-lookup } \text{slp2 } M2 \ fa1)) \ (\text{Var } (\text{slp-index-lookup } \text{slp2 } M2 \ fa2))]$

```

using *
by (auto simp: slp-of-fa-bin-def)
have Mapping.keys M  $\subseteq$  Mapping.keys M'
using IH1 IH2
by (auto simp: M')
have Binop fa1 fa2  $\in$  Mapping.keys M'
using M' by auto
have M'-0: Mapping.lookup M' (Binop fa1 fa2) = Some (length slp2)
by (auto simp: M' lookup-update)
have fa1: fa1  $\in$  Mapping.keys M2 and fa2: fa2  $\in$  Mapping.keys M2
by (force intro: IH2 IH1)+
have rew: binop (interpret-slp slp2 xs ! slp-index-lookup slp2 M2 fa1) (interpret-slp
slp2 xs ! slp-index-lookup slp2 M2 fa2) =
binop (interpret-floatarith fa1 xs) (interpret-floatarith fa2 xs)
by (auto simp: IH2 fa1)
show ?thesis
apply (auto )
subgoal by fact
subgoal
unfolding M'
apply simp
apply (rule disjI2)
apply (rule IH2(2))
apply (rule IH1) apply simp
done
subgoal
unfolding M'
apply simp
apply (rule disjI2)
apply (rule IH2)
by simp
subgoal
unfolding M'
apply simp
apply (rule disjI2)
apply (rule IH2(2))
apply (rule IH1(2))
by simp
subgoal
unfolding M'
apply auto
apply (simp add: IH1(1) IH2(2))
apply (simp add: IH1(2) IH2(1))
using IH2(3)
by auto
subgoal for f
unfolding M' slp'
apply simp
apply (auto simp add: lookup-update' rew lookup-map-values slp-index-lookup-def

```

```

slp-index-def)
  by (simp add: IH2(4) less-Suc-eq)
subgoal for f
  unfolding M' slp'
  apply simp
  apply (subst rew)
  apply (auto simp add: fa1 lookup-update' rew lookup-map-values slp-index-lookup-def
slp-index-def)
  apply (auto simp add: nth-Cons fa1 lookup-update' rew lookup-map-values
slp-index-lookup-def slp-index-def
split: nat.splits)
  using IH2(4) apply fastforce
  by (metis IH2(4) IH2(5) Suc-diff-Suc Suc-inject slp-index-def slp-index-lookup-def)
done
next
case (Some C)
then have M': M' = Mapping.update (Binop fa1 fa2) (length slp) M
  and slp': slp' = slp @ [Var (slp-index slp C)]
  and Binop-keys: (Binop fa1 fa2) ∈ Mapping.keys M
  using *
  by (auto simp: slp-of-fa-bin-def keys-dom-lookup)
have subterms (Binop fa1 fa2) ⊆ Mapping.keys M'
  using Binop-keys assms(4)
  by (force simp: M')
moreover
have Mapping.keys M ⊆ Mapping.keys M'
  using Binop-keys
  by (auto simp add: M')
moreover have f ∈ Mapping.keys M' ⇒ interpret-slp slp' xs ! slp-index-lookup
slp' M' f =
  interpret-floatarith f xs for f
  apply (auto simp add: M' lookup-map-values lookup-update' slp' Binop-prems
slp-index-def
slp-index-lookup-def)
  apply (metis Binop-keys Some assms(6) interpret-Binop option.sel slp-index-def
slp-index-lookup-def)
  apply (metis Binop-keys Some assms(6) interpret-Binop option.sel slp-index-def
slp-index-lookup-def)
  apply (metis assms(6) slp-index-def slp-index-lookup-def)
  done
moreover have f ∈ Mapping.keys M' ⇒ subterms f ⊆ Mapping.keys M' for f
  using Binop-keys Some assms(4,6)
  by (auto simp add: M' lookup-map-values)
moreover have f ∈ Mapping.keys M' ⇒ the (Mapping.lookup M' f) < length
slp' for f
  using Binop-keys Some assms(5,7) IH1 IH2
  by (auto simp add: M' lookup-map-values lookup-update' Binop-prems slp'
less-SucI)
ultimately

```



**show** *?thesis*  
**by** *auto*  
**qed**  
**qed**

**lemma**  
*unop-slp-of-fa-induction-step:*  
**assumes**  
*Unop-IH1:*  
 $\bigwedge M \text{ slp } M' \text{ slp}'. \text{ slp-of-fa } fa1 \text{ } M \text{ slp} = (M', \text{ slp}') \implies$   
 $(\bigwedge f. f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M) \implies$   
 $(\bigwedge f. f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M \text{ } f) < \text{length } \text{slp}) \implies$   
 $(\bigwedge f. f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp } xs \text{ ! slp-index-lookup } \text{slp } M \text{ } f =$   
*interpret-floatarith*  $f \text{ } xs) \implies$   
 $\text{subterms } fa1 \subseteq \text{Mapping.keys } M' \wedge$   
 $\text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$   
 $(\forall f \in \text{Mapping.keys } M'. \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$   
 $\text{the } (\text{Mapping.lookup } M' \text{ } f) < \text{length } \text{slp}' \wedge$   
 $\text{interpret-slp } \text{slp}' \text{ } xs \text{ ! slp-index-lookup } \text{slp}' \text{ } M' \text{ } f = \text{interpret-floatarith } f \text{ } xs)$   
**and** *Unop-prems:*  
 $(\text{case } \text{slp-of-fa } fa1 \text{ } M \text{ slp } \text{ of } (M1, \text{ slp1}) \Rightarrow \text{slp-of-fa-un } \text{Unop } fa1 \text{ } M \text{ slp } M1 \text{ slp1})$   
 $= (M', \text{ slp}')$   
 $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M$   
 $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M \text{ } f) < \text{length } \text{slp}$   
 $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp } xs \text{ ! slp-index-lookup } \text{slp } M \text{ } f =$   
*interpret-floatarith*  $f \text{ } xs$   
**assumes** *subterms-Unop[simp]:*  
 $\bigwedge a \text{ } b. \text{subterms } (\text{Unop } a) = \text{insert } (\text{Unop } a) (\text{subterms } a)$   
**assumes** *interpret-Unop[simp]:*  
 $\bigwedge a \text{ } b \text{ } xs. \text{interpret-floatarith } (\text{Unop } a) \text{ } xs = \text{unop } (\text{interpret-floatarith } a \text{ } xs)$   
**shows**  $\text{insert } (\text{Unop } fa1) (\text{subterms } fa1) \subseteq \text{Mapping.keys } M' \wedge$   
 $\text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$   
 $(\forall f \in \text{Mapping.keys } M'. \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$   
 $\text{the } (\text{Mapping.lookup } M' \text{ } f) < \text{length } \text{slp}' \wedge$   
 $\text{interpret-slp } \text{slp}' \text{ } xs \text{ ! slp-index-lookup } \text{slp}' \text{ } M' \text{ } f = \text{interpret-floatarith } f \text{ } xs)$

**proof** –  
**from** *Unop-prems*  
**obtain**  $M1 \text{ slp1}$  **where**  $*$ :  
 $\text{slp-of-fa } fa1 \text{ } M \text{ slp} = (M1, \text{ slp1})$   
 $\text{slp-of-fa-un } \text{Unop } fa1 \text{ } M \text{ slp } M1 \text{ slp1} = (M', \text{ slp}')$   
**by** (*auto split: prod.splits*)  
**from** *Unop-IH1* [*OF*  $*(1)$  *Unop-prems*(2) *Unop-prems*(3) *Unop-prems*(4), *simplified*]  
**have** *IH1:*  
 $f \in \text{subterms } fa1 \implies f \in \text{Mapping.keys } M1$   
 $f \in \text{Mapping.keys } M \implies f \in \text{Mapping.keys } M1$   
 $f \in \text{Mapping.keys } M1 \implies \text{subterms } f \subseteq \text{Mapping.keys } M1$   
 $f \in \text{Mapping.keys } M1 \implies \text{the } (\text{Mapping.lookup } M1 \text{ } f) < \text{length } \text{slp1}$   
 $f \in \text{Mapping.keys } M1 \implies \text{interpret-slp } \text{slp1 } xs \text{ ! slp-index-lookup } \text{slp1 } M1 \text{ } f =$

```

interpret-floatarith f xs
  for f
  by (auto simp: subset-iff)
show ?thesis
proof (cases Mapping.lookup M (Unop fa1))
case None
then have M': M' = Mapping.update (Unop fa1) (length slp1) M1
  and slp': slp' = slp1 @ [Unop (Var (slp-index-lookup slp1 M1 fa1))]
  using *
  by (auto simp: slp-of-fa-un-def)
have Mapping.keys M  $\subseteq$  Mapping.keys M'
  using IH1
  by (auto simp: M')
have Unop fa1  $\in$  Mapping.keys M'
  using M' by auto
have fa1: fa1  $\in$  Mapping.keys M1
  by (force intro: IH1)+
have rew: interpret-slp slp1 xs ! slp-index-lookup slp1 M1 fa1 = interpret-floatarith
fa1 xs
  by (auto simp: IH1 fa1)
show ?thesis
  apply (auto)
  subgoal by fact
  subgoal
    unfolding M'
    apply simp
    apply (rule disjI2)
    apply (rule IH1) apply simp
    done
  subgoal
    unfolding M'
    apply simp
    apply (rule disjI2)
    by (rule IH1) simp
  subgoal
    using IH1(3) M'  $\langle \wedge x. x \in \text{subterms } fa1 \implies x \in \text{Mapping.keys } M' \rangle$  by
fastforce
  subgoal for f
    unfolding M' slp'
    apply simp
    apply (auto simp add: lookup-update' rew lookup-map-values)
    by (simp add: IH1(4) less-SucI)
  subgoal for f
    unfolding M' slp'
    apply simp
    apply (subst rew)
    apply (auto simp add: fa1 lookup-update' rew lookup-map-values slp-index-lookup-def
slp-index-def)
    apply (auto simp add: nth-Cons fa1 lookup-update' rew lookup-map-values

```

```

slp-index-lookup-def slp-index-def
  split: nat.splits)
  using IH1(4) apply fastforce
  by (metis IH1(4) IH1(5) Suc-diff-Suc Suc-inject slp-index-def slp-index-lookup-def)
  done
next
case (Some C)
then have M': M' = Mapping.update (Unop fa1) (length slp) M
  and slp': slp' = slp @ [Var (slp-index slp C)]
  and Unop-keys: (Unop fa1) ∈ Mapping.keys M
  using *
  by (auto simp: slp-of-fa-un-def keys-dom-lookup)
have subterms (Unop fa1) ⊆ Mapping.keys M'
  using Unop-keys assms(3)
  by (force simp: M')
moreover
have Mapping.keys M ⊆ Mapping.keys M'
  using Unop-keys assms(5)
  by (force simp: M' IH1)
moreover have f ∈ Mapping.keys M' ⇒ interpret-slp slp' xs ! slp-index-lookup
slp' M' f =
  interpret-floatarith f xs for f
  apply (auto simp add: M' lookup-map-values lookup-update' slp' Unop-prems
slp-index-def slp-index-lookup-def)
  apply (metis Unop-keys Some assms(5) interpret-Unop option.sel slp-index-def
slp-index-lookup-def)
  apply (metis Unop-keys Some assms(5) interpret-Unop option.sel slp-index-def
slp-index-lookup-def)
  apply (metis assms(5) slp-index-def slp-index-lookup-def)
  done
moreover have f ∈ Mapping.keys M' ⇒ subterms f ⊆ Mapping.keys M' for f
  using Unop-keys Some assms(3,5)
  by (auto simp add: M' lookup-map-values)
moreover have f ∈ Mapping.keys M' ⇒ the (Mapping.lookup M' f) < length
slp' for f
  by (auto simp add: M' lookup-map-values lookup-update' slp' Unop-prems IH1
less-SucI)
ultimately
show ?thesis
  by auto
qed
qed

```

**lemma**

*cnst-slp-of-fa-induction-step:*

**assumes** \*:

*slp-of-fa-cnst Unop Unop' M slp = (M', slp')*

$\bigwedge f. f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M$

$\bigwedge f. f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M f) < \text{length } \text{slp}$

```

   $\wedge f. f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp } xs \ ! \ \text{slp-index-lookup } \text{slp } M \ f =$ 
   $\text{interpret-floatarith } f \ xs$ 
assumes subterms-Unop[simp]:
   $\wedge a \ b. \text{subterms } (Unop) = \{Unop\}$ 
assumes interpret-Unop[simp]:
   $\text{interpret-floatarith } Unop \ xs = unop \ xs$ 
   $\text{interpret-floatarith } Unop' \ (\text{interpret-slp } \text{slp } xs) = unop \ xs$ 
assumes ui:  $unop \ (\text{interpret-slp } \text{slp } xs) = unop \ xs$ 
shows  $\{Unop\} \subseteq \text{Mapping.keys } M' \wedge$ 
   $\text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$ 
   $(\forall f \in \text{Mapping.keys } M'. \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$ 
   $\text{the } (\text{Mapping.lookup } M' \ f) < \text{length } \text{slp}' \wedge$ 
   $\text{interpret-slp } \text{slp}' \ xs \ ! \ \text{slp-index-lookup } \text{slp}' \ M' \ f = \text{interpret-floatarith } f \ xs)$ 
proof –
show ?thesis
proof (cases Mapping.lookup M Unop)
  case None
  then have  $M': M' = \text{Mapping.update } Unop \ (\text{length } \text{slp}) \ M$ 
  and  $\text{slp}': \text{slp}' = \text{slp} \ @ \ [Unop]$ 
  using *
  by (auto simp: slp-of-fa-cnst-def)
have  $\text{Mapping.keys } M \subseteq \text{Mapping.keys } M'$ 
  by (auto simp: M')
have  $Unop \in \text{Mapping.keys } M'$ 
  using  $M'$  by auto
show ?thesis
apply (auto)
subgoal by fact
subgoal
  unfolding  $M'$ 
  apply simp
  done
subgoal
  unfolding  $M'$ 
  apply simp
  using assms by auto
subgoal
  unfolding  $M' \ \text{slp}'$ 
  apply simp
  apply (auto simp add: lookup-update' ui lookup-map-values)
  using interpret-Unop apply auto[1]
  by (simp add: assms(3) less-Suc-eq)
subgoal for  $f$ 
  unfolding  $M' \ \text{slp}'$ 
  apply simp
  apply (auto simp add: lookup-update' ui lookup-map-values slp-index-lookup-def
  slp-index-def)
  using interpret-Unop apply auto[1]
  apply (auto simp: nth-Cons split: nat.splits)

```

```

    using assms(3) leD apply blast
  by (metis Suc-diff-Suc Suc-inject assms(3) assms(4) slp-index-def slp-index-lookup-def)
  done
next
case (Some C)
then have  $M' : M' = \text{Mapping.update } \text{Unop } (\text{length } \text{slp}) M$ 
  and  $\text{slp}' : \text{slp}' = \text{slp } @ [ \text{Var } (\text{slp-index } \text{slp } C) ]$ 
  and  $\text{Unop-keys} : (\text{Unop}) \in \text{Mapping.keys } M$ 
  using *
  by (auto simp: slp-of-fa-cnst-def keys-dom-lookup)
have subterms  $(\text{Unop}) \subseteq \text{Mapping.keys } M'$ 
  using Unop-keys
  by (fastforce simp: M')
moreover
have  $\text{Mapping.keys } M \subseteq \text{Mapping.keys } M'$ 
  using Unop-keys assms(5)
  by (force simp: M')
moreover have  $f \in \text{Mapping.keys } M' \implies \text{interpret-slp } \text{slp}' \text{ } xs ! \text{slp-index-lookup}$ 
 $\text{slp}' M' f = \text{interpret-floatarith } f \text{ } xs$  for  $f$ 
  apply (auto simp add: M' lookup-map-values lookup-update' slp' slp-index-lookup-def)
  apply (metis Some Unop-keys assms(4) interpret-Unop option.sel slp-index-def)
  apply (metis Some Unop-keys assms(4) interpret-Unop option.sel slp-index-def)
  by (metis Suc-diff-Suc assms(3) assms(4) nth-Cons-Suc slp-index-def slp-index-lookup-def)
moreover have  $f \in \text{Mapping.keys } M' \implies \text{subterms } f \subseteq \text{Mapping.keys } M'$  for  $f$ 
  using assms by (auto simp add: M' lookup-map-values lookup-update' slp')
moreover have  $f \in \text{Mapping.keys } M' \implies \text{the } (\text{Mapping.lookup } M' f) < \text{length}$ 
 $\text{slp}'$  for  $f$ 
  using assms
  by (auto simp add: M' lookup-map-values lookup-update' slp' less-SucI)
ultimately
show ?thesis
  by auto
qed
qed

```

**lemma** *interpret-slp-nth*:

```

 $n \geq \text{length } \text{slp} \implies \text{interpret-slp } \text{slp } xs ! n = xs ! (n - \text{length } \text{slp})$ 
by (induction slp arbitrary: xs n) auto

```

**theorem**

```

interpret-slp-of-fa:
assumes  $\text{slp-of-fa } fa \text{ } M \text{ } \text{slp} = (M', \text{slp}')$ 
assumes  $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M$ 
assumes  $\bigwedge f. f \in \text{Mapping.keys } M \implies (\text{the } (\text{Mapping.lookup } M f)) < \text{length } \text{slp}$ 
assumes  $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp } xs ! \text{slp-index-lookup } \text{slp}$ 
 $M f = \text{interpret-floatarith } f \text{ } xs$ 

```

```

shows subterms fa  $\subseteq$  Mapping.keys M'  $\wedge$  Mapping.keys M  $\subseteq$  Mapping.keys M'
 $\wedge$ 
  ( $\forall f \in$  Mapping.keys M'.
    subterms f  $\subseteq$  Mapping.keys M'  $\wedge$ 
    the (Mapping.lookup M' f) < length slp'  $\wedge$ 
    (interpret-slp slp' xs ! slp-index-lookup slp' M' f = interpret-floatarith f xs))
using assms
proof (induction fa arbitrary: M' slp' M slp)
case *: (Add fa1 fa2)
show ?case
  unfolding subterms.simps
  by (rule binop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Mult fa1 fa2)
show ?case
  unfolding subterms.simps
  by (rule binop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Min fa1 fa2)
show ?case
  unfolding subterms.simps
  by (rule binop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Max fa1 fa2)
show ?case
  unfolding subterms.simps
  by (rule binop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Pour fa1 fa2)
show ?case
  unfolding subterms.simps
  by (rule binop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Minus fa1)
show ?case
  unfolding subterms.simps
  by (rule unop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Inverse fa1)
show ?case
  unfolding subterms.simps
  by (rule unop-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto

```

```

next
  case *: (Arctan fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Floor fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Cos fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Ln fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Power fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Abs fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Sqrt fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
  case *: (Exp fa1)
  show ?case
    unfolding subterms.simps
    by (rule unop-slp-of-fa-induction-step[OF
      *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next

```

```

case *: Pi
show ?case
  unfolding subterms.simps
  by (rule cnst-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: Num
show ?case
  unfolding subterms.simps
  by (rule cnst-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]]) auto
next
case *: (Var n)
show ?case
  unfolding subterms.simps
  by (rule cnst-slp-of-fa-induction-step[OF
    *[unfolded subterms.simps slp-of-fa.simps Let-def]])
    (auto simp: interpret-slp-nth)
qed

primrec slp-of-fas' where
  slp-of-fas' [] M slp = (M, slp)
| slp-of-fas' (fa#fas) M slp = (let (M, slp) = slp-of-fa fa M slp in slp-of-fas' fas M slp)

theorem
  interpret-slp-of-fas':
  assumes slp-of-fas' fas M slp = (M', slp')
  assumes  $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{subterms } f \subseteq \text{Mapping.keys } M$ 
  assumes  $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{the } (\text{Mapping.lookup } M f) < \text{length } \text{slp}$ 
  assumes  $\bigwedge f. f \in \text{Mapping.keys } M \implies \text{interpret-slp } \text{slp } xs \neq \text{slp-index-lookup } \text{slp}$ 
  M f = interpret-floatarith f xs
  shows  $\bigcup (\text{subterms ' set } \text{fas}) \subseteq \text{Mapping.keys } M' \wedge \text{Mapping.keys } M \subseteq \text{Mapping.keys } M' \wedge$ 
    ( $\forall f \in \text{Mapping.keys } M'. \text{subterms } f \subseteq \text{Mapping.keys } M' \wedge$ 
    (the (Mapping.lookup M' f) < length slp')  $\wedge$ 
    (interpret-slp slp' xs  $\neq$  slp-index-lookup slp' M' f = interpret-floatarith f xs))
  using assms
proof (induction fas arbitrary: M slp)
case Nil then show ?case
  by auto
next
case (Cons fa fas)
from  $\langle \text{slp-of-fas'} (fa \# fas) M \text{slp} = (M', \text{slp}') \rangle$ 
obtain M1 slp1 where
  fa: slp-of-fa fa M slp = (M1, slp1)
  and fas: slp-of-fas' fas M1 slp1 = (M', slp')
  by (auto split: prod.splits)
have subterms fa  $\subseteq \text{Mapping.keys } M1 \wedge$ 

```



$Mapping.keys\ M \subseteq Mapping.keys\ M1 \wedge$   
 $(\forall f \in Mapping.keys\ M1. subterms\ f \subseteq Mapping.keys\ M1 \wedge$   
 $the\ (Mapping.lookup\ M1\ f) < length\ slp1 \wedge$   
 $interpret-slp\ slp1\ xs \ !\ slp-index-lookup\ slp1\ M1\ f = interpret-floatarith\ f\ xs)$   
**apply** (rule interpret-slp-of-fa[OF fa, of xs])  
**using** Cons.premis  
**by** (auto split: prod.splits simp: trans-less-add2)

**moreover**  
**then have**  $(\bigcup a \in set\ fas. subterms\ a) \subseteq Mapping.keys\ M' \wedge$   
 $Mapping.keys\ M1 \subseteq Mapping.keys\ M' \wedge$   
 $(\forall f \in Mapping.keys\ M'. subterms\ f \subseteq Mapping.keys\ M' \wedge$   
 $the\ (Mapping.lookup\ M'\ f) < length\ slp'\ \wedge$   
 $interpret-slp\ slp'\ xs \ !\ slp-index-lookup\ slp'\ M'\ f = interpret-floatarith\ f\ xs)$   
**using** Cons.premis  
**by** (intro Cons.IH[OF fas])  
 (auto split: prod.splits simp: trans-less-add2)

**ultimately**  
**show** ?case  
**by** auto

**qed**

**definition** slp-of-fas fas =  
 (let  
 (M, slp) = slp-of-fas' fas Mapping.empty [];  
 fasi = map (the o Mapping.lookup M) fas;  
 fasi' = map  $(\lambda(a, b). Var\ (length\ slp + a - Suc\ b))\ (zip\ [0..<length\ fasi]\ (rev\ fasi))$   
 in slp @ fasi')

**lemma** length-interpret-slp[simp]:  
 $length\ (interpret-slp\ slp\ xs) = length\ slp + length\ xs$   
**by** (induct slp arbitrary: xs) auto

**lemma** length-interpret-floatariths[simp]:  
 $length\ (interpret-floatariths\ slp\ xs) = length\ slp$   
**by** (induct slp arbitrary: xs) auto

**lemma** interpret-slp-append[simp]:  
 $interpret-slp\ (slp1\ @\ slp2)\ xs =$   
 $interpret-slp\ slp2\ (interpret-slp\ slp1\ xs)$   
**by** (induction slp1 arbitrary: slp2 xs) auto

**lemma** interpret-slp (map Var [a + 0, b + 1, c + 2, d + 3]) xs =  
 $(rev\ (map\ (\lambda(i, e). xs\ !\ (e - i))\ (zip\ [0..<4]\ [a + 0, b + 1, c + 2, d + 3])))@xs$   
**by** (auto simp: numeral-eq-Suc)

**lemma** aC-eq-aa: xs @ y # zs = (xs @ [y]) @ zs  
**by** simp

```

lemma
  interpret-slp-map-Var:
  assumes  $\bigwedge i. i < \text{length } is \implies is ! i \geq i$ 
  assumes  $\bigwedge i. i < \text{length } is \implies (is ! i - i) < \text{length } xs$ 
  shows interpret-slp (map Var is) xs =
    (rev (map ( $\lambda(i, e). xs ! (e - i)$ ) (zip [0.. $\text{length } is$ ] is)))
  @
  xs
  using assms
proof (induction is arbitrary: xs)
  case Nil
  then show ?case by simp
next
  case (Cons a is)
  show ?case
    unfolding interpret-slp.simps list.map
    apply (subst Cons.IH)
    subgoal using Cons.prems by force
    subgoal using Cons.prems by force
    subgoal
      apply (subst aC-eq-aa)
      apply (subst rev.simps(2)[symmetric])
      apply (rule arg-cong[where f= $\lambda a. a @ xs$ ])
      apply (rule arg-cong[where f=rev])
      unfolding interpret-floatarith.simps
      apply auto
      apply (rule nth-equalityI)
      apply force
      apply auto
      using Cons.prems
      apply (auto simp: nth-append nth-Cons split: nat.splits)
      subgoal
        by (metis Suc-leI le-imp-less-Suc not-le old.nat.simps(5))
      subgoal
        by (simp add: minus-nat.simps(2))
      subgoal
        by (metis Suc-lessI minus-nat.simps(2) old.nat.simps(5))
      done
    done
  qed

theorem slp-of-fas:
  take (length fas) (interpret-slp (slp-of-fas fas) xs) = interpret-floatariths fas xs
proof –
  obtain M slp where Mslp:
    slp-of-fas' fas Mapping.empty [] = (M, slp)
  using old.prod.exhaust by blast
  have M:  $\bigcup (\text{subterms } (set\ fas)) \subseteq \text{Mapping.keys } M \wedge$ 
     $\text{Mapping.keys } (\text{Mapping.empty}::(\text{floatarith}, \text{nat}) \text{ mapping}) \subseteq \text{Mapping.keys } M$ 

```

$\wedge$   
 $(\forall f \in \text{Mapping.keys } M.$   
 $\text{subterms } f \subseteq \text{Mapping.keys } M \wedge$   
 $\text{the } ( \text{Mapping.lookup } M f ) < \text{length } \text{slp} \wedge$   
 $\text{interpret-slp } \text{slp } \text{xs} ! \text{slp-index-lookup } \text{slp } M f =$   
 $\text{interpret-floatarith } f \text{xs})$   
**by** (rule interpret-slp-of-fas'[OF Mslp]) *auto*  
**have** map-eq:  
 $\text{map } (\lambda(a, b). \text{Var } (\text{length } \text{slp} + a - \text{Suc } b)) (\text{zip } [0..<\text{length } \text{fas}] (\text{rev } (\text{map}$   
 $((\lambda x. \text{the } o (\text{Mapping.lookup } x)) M) \text{fas})))$   
 $= \text{map } \text{Var } (\text{map } (\lambda(a, b). (\text{length } \text{slp} + a - \text{Suc } b)) (\text{zip } [0..<\text{length } \text{fas}] (\text{rev}$   
 $(\text{map } (\text{the } o \text{Mapping.lookup } M) \text{fas}))))$   
**unfolding** split-beta'  
**by** (simp add: split-beta')  
**have** take (length fas)  
 $(\text{interpret-slp}$   
 $(\text{slp } @$   
 $\text{map } (\lambda(a, b). \text{Var } (\text{length } \text{slp} + a - \text{Suc } b)) (\text{zip } [0..<\text{length } \text{fas}] (\text{rev } (\text{map}$   
 $((\lambda x. \text{the } o (\text{Mapping.lookup } x)) M) \text{fas}))))$   
 $\text{xs}) =$   
 $\text{interpret-floatariths } \text{fas } \text{xs}$   
**apply** simp  
**unfolding** map-eq  
**apply** (subst interpret-slp-map-Var)  
**apply** (auto simp: rev-nth)  
**subgoal** premises *prems* for *i*  
**proof** –  
**from** *prems* **have** (length fas – Suc *i*) < length fas **using** *prems* **by** *auto*  
**then** **have** fas ! (length fas – Suc *i*) ∈ set fas  
**by** simp  
**also** **have** ... ⊆ Mapping.keys *M*  
**using** *M* **by** force  
**finally** **have** fas ! (length fas – Suc *i*) ∈ Mapping.keys *M* .  
**with** *M*  
**show** ?thesis  
**by** *auto*  
**qed**  
**subgoal** premises *prems* for *i*  
**proof** –  
**from** *prems* **have** (length fas – Suc *i*) < length fas **using** *prems* **by** *auto*  
**then** **have** fas ! (length fas – Suc *i*) ∈ set fas  
**by** simp  
**also** **have** ... ⊆ Mapping.keys *M*  
**using** *M* **by** force  
**finally** **have** fas ! (length fas – Suc *i*) ∈ Mapping.keys *M* .  
**with** *M*  
**show** ?thesis  
**by** *auto*  
**qed**

```

subgoal
  apply (rule nth-equalityI, auto)
  subgoal premises prems for i
  proof –
    from prems have fas ! i ∈ set fas
      by simp
    also have ... ⊆ Mapping.keys M
      using M by force
    finally have fas ! i ∈ Mapping.keys M .
    from M[THEN conjunct2, THEN conjunct2, rule-format, OF this]
    show ?thesis
      using prems
    by (auto simp: rev-nth interpret-floatariths-nth slp-index-lookup-def slp-index-def)
  qed
done
done
then show ?thesis
  by (auto simp: slp-of-fas-def Let-def Mslp)
qed

```

### 4.3 better code equations for construction of large programs

**definition** *slp-indexl slpl i = slpl – Suc i*

**definition** *slp-indexl-lookup vsl M a = slp-indexl vsl (the (Mapping.lookup M a))*

**definition**

*slp-of-fa-rev-bin Binop a b M slp slpl M2 slp2 slpl2 =*  
*(case Mapping.lookup M (Binop a b) of*  
*Some i ⇒ (Mapping.update (Binop a b) (slpl) M, Var (slp-indexl slpl i)#slp,*  
*Suc slpl)*  
*| None ⇒ (Mapping.update (Binop a b) (slpl2) M2,*  
*Binop (Var (slp-indexl-lookup slpl2 M2 a)) (Var (slp-indexl-lookup*  
*slpl2 M2 b))#slp2,*  
*Suc slpl2))*

**definition**

*slp-of-fa-rev-un Unop a M slp slpl M1 slp1 slpl1 =*  
*(case Mapping.lookup M (Unop a) of*  
*Some i ⇒ (Mapping.update (Unop a) (slpl) M, Var (slp-indexl slpl i)#slp,*  
*Suc slpl)*  
*| None ⇒ (Mapping.update (Unop a) (slpl1) M1,*  
*Unop (Var (slp-indexl-lookup slpl1 M1 a))#slp1, Suc slpl1))*

**definition**

*slp-of-fa-rev-cnst Const Const' M vs vsl =*  
*(Mapping.update Const vsl M,*  
*(case Mapping.lookup M Const of Some i ⇒ Var (slp-indexl vsl i) | None ⇒*  
*Const')#vs, Suc vsl)*

```

fun slp-of-fa-rev :: floatarith  $\Rightarrow$  (floatarith, nat) mapping  $\Rightarrow$  floatarith list  $\Rightarrow$  nat
 $\Rightarrow$ 
  ((floatarith, nat) mapping  $\times$  floatarith list  $\times$  nat) where
  slp-of-fa-rev (Add a b) M slp slpl =
    (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl; (M2, slp2, slpl2) = slp-of-fa-rev
  b M1 slp1 slpl1 in
    slp-of-fa-rev-bin Add a b M slp slpl M2 slp2 slpl2)
| slp-of-fa-rev (Mult a b) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl; (M2, slp2, slpl2) = slp-of-fa-rev
  b M1 slp1 slpl1 in
    slp-of-fa-rev-bin Mult a b M slp slpl M2 slp2 slpl2)
| slp-of-fa-rev (Min a b) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl; (M2, slp2, slpl2) = slp-of-fa-rev
  b M1 slp1 slpl1 in
    slp-of-fa-rev-bin Min a b M slp slpl M2 slp2 slpl2)
| slp-of-fa-rev (Max a b) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl; (M2, slp2, slpl2) = slp-of-fa-rev
  b M1 slp1 slpl1 in
    slp-of-fa-rev-bin Max a b M slp slpl M2 slp2 slpl2)
| slp-of-fa-rev (Powr a b) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl; (M2, slp2, slpl2) = slp-of-fa-rev
  b M1 slp1 slpl1 in
    slp-of-fa-rev-bin Powr a b M slp slpl M2 slp2 slpl2)
| slp-of-fa-rev (Inverse a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Inverse a M
  slp slpl M1 slp1 slpl1)
| slp-of-fa-rev (Cos a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Cos a M slp
  slpl M1 slp1 slpl1)
| slp-of-fa-rev (Arctan a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Arctan a M
  slp slpl M1 slp1 slpl1)
| slp-of-fa-rev (Abs a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Abs a M slp
  slpl M1 slp1 slpl1)
| slp-of-fa-rev (Sqrt a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Sqrt a M slp
  slpl M1 slp1 slpl1)
| slp-of-fa-rev (Exp a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Exp a M slp
  slpl M1 slp1 slpl1)
| slp-of-fa-rev (Ln a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Ln a M slp
  slpl M1 slp1 slpl1)
| slp-of-fa-rev (Minus a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Minus a M
  slp slpl M1 slp1 slpl1)
| slp-of-fa-rev (Floor a) M slp slpl =
  (let (M1, slp1, slpl1) = slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un Floor a M

```

$slp\ slpl\ M1\ slp1\ spl1)$   
 $| slp\text{-of-fa-rev}\ (Power\ a\ n)\ M\ slp\ spl =$   
 $(let\ (M1,\ slp1,\ spl1) = slp\text{-of-fa-rev}\ a\ M\ slp\ spl\ in\ slp\text{-of-fa-rev-un}\ (\lambda a.\ Power$   
 $a\ n)\ a\ M\ slp\ spl\ M1\ slp1\ spl1)$   
 $| slp\text{-of-fa-rev}\ Pi\ M\ slp\ spl = slp\text{-of-fa-rev-cnst}\ Pi\ Pi\ M\ slp\ spl$   
 $| slp\text{-of-fa-rev}\ (Var\ v)\ M\ slp\ spl = slp\text{-of-fa-rev-cnst}\ (Var\ v)\ (Var\ (v + spl))\ M$   
 $slp\ spl$   
 $| slp\text{-of-fa-rev}\ (Num\ n)\ M\ slp\ spl = slp\text{-of-fa-rev-cnst}\ (Num\ n)\ (Num\ n)\ M\ slp\ spl$

**lemma**  $slp\text{-index-length[simp]}$ :  $slp\text{-indexl}\ (length\ xs)\ i = slp\text{-index}\ xs\ i$   
**by**  $(auto\ simp:\ slp\text{-index-def}\ slp\text{-indexl-def})$

**lemma**  $slp\text{-indexl-lookup-length[simp]}$ :  $slp\text{-indexl-lookup}\ (length\ xs)\ i = slp\text{-index-lookup}$   
 $xs\ i$   
**by**  $(auto\ simp:\ slp\text{-index-lookup-def}\ slp\text{-indexl-lookup-def})$

**lemma**  $slp\text{-index-rev[simp]}$ :  $slp\text{-index}\ (rev\ xs)\ i = slp\text{-index}\ xs\ i$   
**by**  $(auto\ simp:\ slp\text{-index-def}\ slp\text{-indexl-def})$

**lemma**  $slp\text{-index-lookup-rev[simp]}$ :  $slp\text{-index-lookup}\ (rev\ xs)\ i = slp\text{-index-lookup}$   
 $xs\ i$   
**by**  $(auto\ simp:\ slp\text{-index-lookup-def}\ slp\text{-indexl-lookup-def})$

**lemma**  $slp\text{-of-fa-bin-slp-of-fa-rev-bin}$ :  
 $slp\text{-of-fa-rev-bin}\ Binop\ a\ b\ M\ slp\ (length\ slp)\ M2\ spl2\ (length\ spl2) =$   
 $(let\ (M,\ spl') = slp\text{-of-fa-bin}\ Binop\ a\ b\ M\ (rev\ slp)\ M2\ (rev\ spl2)\ in\ (M,\ rev$   
 $slp',\ length\ spl'))$   
**by**  $(auto\ simp:\ slp\text{-of-fa-rev-bin-def}\ slp\text{-of-fa-bin-def}$   
 $split:\ prod.splits\ option.splits)$

**lemma**  $slp\text{-of-fa-un-slp-of-fa-rev-un}$ :  
 $slp\text{-of-fa-rev-un}\ Binop\ a\ M\ slp\ (length\ slp)\ M2\ spl2\ (length\ spl2) =$   
 $(let\ (M,\ spl') = slp\text{-of-fa-un}\ Binop\ a\ M\ (rev\ slp)\ M2\ (rev\ spl2)\ in\ (M,\ rev\ spl',$   
 $length\ spl'))$   
**by**  $(auto\ simp:\ slp\text{-of-fa-rev-un-def}\ slp\text{-of-fa-un-def}\ split:\ prod.splits\ option.splits)$

**lemma**  $slp\text{-of-fa-cnst-slp-of-fa-rev-cnst}$ :  
 $slp\text{-of-fa-rev-cnst}\ Cnst\ Cnst'\ M\ slp\ (length\ slp) =$   
 $(let\ (M,\ spl') = slp\text{-of-fa-cnst}\ Cnst\ Cnst'\ M\ (rev\ slp)\ in\ (M,\ rev\ spl',\ length$   
 $slp'))$   
**by**  $(auto\ simp:\ slp\text{-of-fa-rev-cnst-def}\ slp\text{-of-fa-cnst-def}$   
 $split:\ prod.splits\ option.splits)$

**lemma**  $slp\text{-of-fa-rev}$ :  
 $slp\text{-of-fa-rev}\ fa\ M\ slp\ (length\ slp) = (let\ (M,\ spl') = slp\text{-of-fa}\ fa\ M\ (rev\ slp)\ in$   
 $(M,\ rev\ spl',\ length\ spl'))$   
**proof**  $(induction\ fa\ arbitrary:\ M\ slp)$   
**case**  $(Add\ fa1\ fa2)$   
**then show**  $?case$

```

    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (no-types, lifting) Pair-inject length-rev prod.simps(2) rev-rev-ident
slp-of-fa-bin-slp-of-fa-rev-bin)
next
case (Minus fa)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
case (Mult fa1 fa2)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (no-types, lifting) Pair-inject length-rev prod.simps(2) rev-rev-ident
slp-of-fa-bin-slp-of-fa-rev-bin)
next
case (Inverse fa)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
case (Cos fa)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
case (Arctan fa)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
case (Abs fa)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
case (Max fa1 fa2)
then show ?case
    by (auto split: prod.splits simp: Let-def
        slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
        (metis (no-types, lifting) Pair-inject length-rev prod.simps(2) rev-rev-ident
slp-of-fa-bin-slp-of-fa-rev-bin)
next

```

```

case (Min fa1 fa2)
then show ?case
  by (auto split: prod.splits simp: Let-def
    slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
    (metis (no-types, lifting) Pair-inject length-rev prod.simps(2) rev-rev-ident
slp-of-fa-bin-slp-of-fa-rev-bin)
next
  case Pi
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
next
  case (Sqrt fa)
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
      (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
  case (Exp fa)
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
      (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
  case (Powr fa1 fa2)
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
      (metis (no-types, lifting) Pair-inject length-rev prod.simps(2) rev-rev-ident
slp-of-fa-bin-slp-of-fa-rev-bin)
next
  case (Ln fa)
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
      (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
  case (Power fa x2a)
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
      (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next
  case (Floor fa)
  then show ?case
    by (auto split: prod.splits simp: Let-def
      slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
      (metis (mono-tags, lifting) length-rev prod.simps(2) rev-swap slp-of-fa-un-slp-of-fa-rev-un)
next

```



```

case (Var x)
then show ?case
  by (auto split: prod.splits simp: Let-def
    slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
next
case (Num x)
then show ?case
  by (auto split: prod.splits simp: Let-def
    slp-of-fa-cnst-slp-of-fa-rev-cnst slp-of-fa-bin-slp-of-fa-rev-bin slp-of-fa-un-slp-of-fa-rev-un)
qed

```

```

lemma slp-of-fa-code[code]:
  slp-of-fa fa M slp = (let (M, slp', -) = slp-of-fa-rev fa M (rev slp) (length slp) in
    (M, rev slp'))
  using slp-of-fa-rev[of fa M rev slp]
  by (auto split: prod.splits)

```

```

definition norm2-slp n = slp-of-fas [floatarith.Inverse (norm2e n)]

```

```

unbundle no floatarith-syntax

```

```

end

```

## 5 Approximation with Affine Forms

```

theory Affine-Approximation

```

```

imports

```

```

  HOL-Decision-Procs.Approximation

```

```

  HOL-Library.Monad-Syntax

```

```

  HOL-Library.Mapping

```

```

  Executable-Euclidean-Space

```

```

  Affine-Form

```

```

  Straight-Line-Program

```

```

begin

```

```

lemma convex-on-imp-above-tangent:— TODO: generalizes  $\llbracket \text{convex-on } ?A \text{ } ?f; \text{ connected } ?A; ?c \in \text{interior } ?A; ?x \in ?A; (?f \text{ has-real-derivative } ?f') \text{ (at } ?c \text{ within } ?A) \rrbracket$ 
 $\implies ?f' * (?x - ?c) \leq ?f ?x - ?f ?c$ 

```

```

  assumes convex: convex-on A f and connected: connected A

```

```

  assumes c: c ∈ A and x : x ∈ A

```

```

  assumes deriv: (f has-field-derivative f') (at c within A)

```

```

  shows f x - f c ≥ f' * (x - c)

```

```

proof (cases x c rule: linorder-cases)

```

```

  assume xc: x > c

```

```

  let ?A' = {c<.. $x$ }

```

```

  have subs: ?A' ⊆ A using xc x c

```

```

    by (simp add: connected connected-contains-Ioo)

```

```

  have at c within ?A' ≠ bot

```

```

    using xc

```

by (simp add: at-within-eq-bot-iff)  
 moreover from deriv have  $((\lambda y. (f y - f c) / (y - c)) \longrightarrow f')$  (at c within ?A')  
 unfolding has-field-derivative-iff using subs  
 by (blast intro: tendsto-mono at-le)  
 moreover from eventually-at-right-real[OF xc]  
 have eventually  $(\lambda y. (f y - f c) / (y - c) \leq (f x - f c) / (x - c))$  (at-right c)  
 proof eventually-elim  
 fix y assume y:  $y \in \{c <..< x\}$   
 with convex connected x c have  $f y \leq (f x - f c) / (x - c) * (y - c) + f c$   
 using interior-subset[of A]  
 by (intro convex-onD-Icc' convex-on-subset[OF convex] connected-contains-Icc)  
 auto  
 hence  $f y - f c \leq (f x - f c) / (x - c) * (y - c)$  by simp  
 thus  $(f y - f c) / (y - c) \leq (f x - f c) / (x - c)$  using y xc by (simp add: divide-simps)  
 qed  
 hence eventually  $(\lambda y. (f y - f c) / (y - c) \leq (f x - f c) / (x - c))$  (at c within ?A')  
 by (simp add: eventually-at-filter eventually-mono)  
 ultimately have  $f' \leq (f x - f c) / (x - c)$  by (simp add: tendsto-upperbound)  
 thus ?thesis using xc by (simp add: field-simps)  
 next  
 assume xc:  $x < c$   
 let ?A' =  $\{x <..< c\}$   
 have subs:  $?A' \subseteq A$  using xc x c  
 by (simp add: connected connected-contains-Ioo)  
 have at c within ?A'  $\neq$  bot  
 using xc  
 by (simp add: at-within-eq-bot-iff)  
 moreover from deriv have  $((\lambda y. (f y - f c) / (y - c)) \longrightarrow f')$  (at c within ?A')  
 unfolding has-field-derivative-iff using subs  
 by (blast intro: tendsto-mono at-le)  
 moreover from eventually-at-left-real[OF xc]  
 have eventually  $(\lambda y. (f y - f c) / (y - c) \geq (f x - f c) / (x - c))$  (at-left c)  
 proof eventually-elim  
 fix y assume y:  $y \in \{x <..< c\}$   
 with convex connected x c have  $f y \leq (f x - f c) / (c - x) * (c - y) + f c$   
 using interior-subset[of A]  
 by (intro convex-onD-Icc'' convex-on-subset[OF convex] connected-contains-Icc)  
 auto  
 hence  $f y - f c \leq (f x - f c) * ((c - y) / (c - x))$  by simp  
 also have  $(c - y) / (c - x) = (y - c) / (x - c)$  using y xc by (simp add: field-simps)  
 finally show  $(f y - f c) / (y - c) \geq (f x - f c) / (x - c)$  using y xc  
 by (simp add: divide-simps)  
 qed  
 hence eventually  $(\lambda y. (f y - f c) / (y - c) \geq (f x - f c) / (x - c))$  (at c within

?A')  
 by (simp add: eventually-at-filter eventually-mono)  
 ultimately have  $f' \geq (f x - f c) / (x - c)$  by (simp add: tendsto-lowerbound)  
 thus ?thesis using xc by (simp add: field-simps)  
 qed simp-all

Approximate operations on affine forms.

**lemma** Affine-notempty[*intro, simp*]: Affine  $X \neq \{\}$   
 by (auto simp: Affine-def valuate-def)

**lemma** truncate-up-lt:  $x < y \implies x < \text{truncate-up prec } y$   
 by (rule less-le-trans[OF - truncate-up])

**lemma** truncate-up-pos-eq[*simp*]:  $0 < \text{truncate-up } p \ x \iff 0 < x$   
 by (auto simp: truncate-up-lt) (metis (poly-guards-query) not-le truncate-up-nonpos)

**lemma** inner-scaleR-pdevs-0: inner-scaleR-pdevs 0 One-pdevs = zero-pdevs  
 unfolding inner-scaleR-pdevs-def  
 by transfer (auto simp: unop-pdevs-raw-def)

**lemma** Affine-aform-of-point-eq[*simp*]: Affine (aform-of-point  $p$ ) =  $\{p\}$   
 by (simp add: Affine-aform-of-ivl aform-of-point-def)

**lemma** mem-Affine-aform-of-point:  $x \in \text{Affine (aform-of-point } x)$   
 by simp

**lemma**  
 aform-val-aform-of-ivl-innerE:  
 assumes  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
 assumes  $a \leq b \ c \in \text{Basis}$   
 obtains  $f$  where aform-val  $e$  (aform-of-ivl  $a \ b$ )  $\cdot c = \text{aform-val } f$  (aform-of-ivl  
 ( $a \cdot c$ ) ( $b \cdot c$ ))  
 $f \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
 proof -  
 have [simp]:  $a \cdot c \leq b \cdot c$   
 using assms by (auto simp: eucl-le[where 'a='a])  
 have ( $\lambda x. x \cdot c$ ) ' Affine (aform-of-ivl  $a \ b$ ) = Affine (aform-of-ivl ( $a \cdot c$ ) ( $b \cdot c$ ))  
 using assms  
 by (auto simp: Affine-aform-of-ivl eucl-le[where 'a='a])  
 image-eqI[where  $x = \sum_{i \in \text{Basis}} (if \ i = c \ then \ x \ else \ a \cdot i) *_R \ i$  for  $x$ )  
 then obtain  $f$  where  
 aform-val  $e$  (aform-of-ivl  $a \ b$ )  $\cdot c = \text{aform-val } f$  (aform-of-ivl ( $a \cdot c$ ) ( $b \cdot c$ ))  
 $f \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
 using assms  
 by (force simp: Affine-def valuate-def)  
 thus ?thesis ..  
 qed

**lift-definition** coord-pdevs::nat  $\Rightarrow$  real pdevs is  $\lambda n \ i. \text{if } i = n \ \text{then } 1 \ \text{else } 0$  by

*auto*

**lemma** *pdevs-apply-coord-pdevs* [*simp*]: *pdevs-apply* (*coord-pdevs* *i*) *x* = (if *x* = *i* then 1 else 0)  
by *transfer simp*

**lemma** *degree-coord-pdevs*[*simp*]: *degree* (*coord-pdevs* *i*) = *Suc* *i*  
by (*auto intro!*: *degree-eqI*)

**lemma** *pdevs-val-coord-pdevs*[*simp*]: *pdevs-val* *e* (*coord-pdevs* *i*) = *e* *i*  
by (*auto simp*: *pdevs-val-sum if-distrib sum.delta cong: if-cong*)

**definition** *aforms-of-ivls* *ls us* = *map*  
( $\lambda(i, (l, u)). ((l + u)/2, \text{scaleR-pdevs } ((u - l)/2) (\text{coord-pdevs } i))$ )  
(*zip* [ $0..<\text{length } ls$ ] (*zip* *ls us*))

**lemma**

*aforms-of-ivls*:

**assumes** *length* *ls* = *length* *us* *length* *xs* = *length* *ls*

**assumes**  $\bigwedge i. i < \text{length } xs \implies xs ! i \in \{ls ! i .. us ! i\}$

**shows** *xs*  $\in$  *Joints* (*aforms-of-ivls* *ls us*)

**proof** –

{  
  **fix** *i* **assume** *i* < *length* *xs*  
  **then have**  $\exists e. e \in \{-1 .. 1\} \wedge xs ! i = (ls ! i + us ! i) / 2 + e * (us ! i - ls ! i) / 2$   
  **using** *assms*  
  **by** (*force intro!*: *exI*[**where**  $x=(xs ! i - (ls ! i + us ! i) / 2) / (us ! i - ls ! i) * 2$ ]  
    *simp: divide-simps algebra-simps*)  
  **then obtain** *e* **where**  $e : e \in \{-1 .. 1\}$   
     $xs ! i = (ls ! i + us ! i) / 2 + e * (us ! i - ls ! i) / 2$   
  **if** *i* < *length* *xs* **for** *i*  
  **using** *that* **by** *metis*  
  **define** *e'* **where**  $e' i = (\text{if } i < \text{length } xs \text{ then } e \text{ else } 0)$  **for** *i*  
  **show** *?thesis*  
  **using** *e assms*  
  **by** (*auto simp*: *aforms-of-ivls-def Joints-def valuate-def e'-def aform-val-def*  
    *intro!*: *image-eqI*[**where**  $x=e'$ ] *nth-equalityI*)  
}

**qed**

## 5.1 Approximate Operations

**definition** *max-pdev* *x* = *fold* ( $\lambda x y. \text{if } \text{infnorm } (\text{snd } x) \geq \text{infnorm } (\text{snd } y) \text{ then } x \text{ else } y$ ) (*list-of-pdevs* *x*) (0, 0)

### 5.1.1 set of generated endpoints

**fun** *points-of-list* **where**

*points-of-list* *x0* [] = [*x0*]

| *points-of-list*  $x0 ((i, x)\#xs) = (\text{points-of-list } (x0 + x) xs @ \text{points-of-list } (x0 - x) xs)$

**primrec** *points-of-aform* **where**

*points-of-aform*  $(x, xs) = \text{points-of-list } x (\text{list-of-pdevs } xs)$

### 5.1.2 Approximate total deviation

**definition** *sum-list'*:  $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a::\text{executable-euclidean-space}$

**where** *sum-list'*  $p xs = \text{fold } (\lambda a b. \text{eucl-truncate-up } p (a + b)) xs 0$

**definition** *tdev'*  $p x = \text{sum-list}' p (\text{map } (\text{abs } o \text{snd}) (\text{list-of-pdevs } x))$

**lemma**

*eucl-fold-mono*:

**fixes**  $f::'a::\text{ordered-euclidean-space} \Rightarrow 'a \Rightarrow 'a$

**assumes** *mono*:  $\bigwedge w x y z. w \leq x \Longrightarrow y \leq z \Longrightarrow f w y \leq f x z$

**shows**  $x \leq y \Longrightarrow \text{fold } f xs x \leq \text{fold } f xs y$

**by** (*induct xs arbitrary: x y*) (*auto simp: mono*)

**lemma** *sum-list-add-le-fold-eucl-truncate-up*:

**fixes**  $z::'a::\text{executable-euclidean-space}$

**shows** *sum-list*  $xs + z \leq \text{fold } (\lambda x y. \text{eucl-truncate-up } p (x + y)) xs z$

**proof** (*induct xs arbitrary: z*)

**case** (*Cons x xs*)

**have** *sum-list*  $(x \# xs) + z = \text{sum-list } xs + (z + x)$

**by** *simp*

**also have**  $\dots \leq \text{fold } (\lambda x y. \text{eucl-truncate-up } p (x + y)) xs (z + x)$

**using** *Cons* **by** *simp*

**also have**  $\dots \leq \text{fold } (\lambda x y. \text{eucl-truncate-up } p (x + y)) xs (\text{eucl-truncate-up } p (x + z))$

**by** (*auto intro!: add-mono eucl-fold-mono eucl-truncate-up eucl-truncate-up-mono simp: ac-simps*)

**finally show** *?case* **by** *simp*

**qed** *simp*

**lemma** *sum-list-le-sum-list'*:

*sum-list*  $xs \leq \text{sum-list}' p xs$

**unfolding** *sum-list'-def*

**using** *sum-list-add-le-fold-eucl-truncate-up*[*of xs 0*] **by** *simp*

**lemma** *sum-list'-sum-list-le*:

$y \leq \text{sum-list } xs \Longrightarrow y \leq \text{sum-list}' p xs$

**by** (*metis sum-list-le-sum-list' order.trans*)

**lemma** *tdev'*:  $tdev x \leq tdev' p x$

**unfolding** *tdev'-def*

**proof** –

**have**  $tdev x = (\sum i = 0 ..< \text{degree } x. |\text{pdevs-apply } x i|)$

by (auto intro!: sum.mono-neutral-cong-left simp: tdev-def)  
 also have ... = ( $\sum i \leftarrow \text{rev } [0 .. < \text{degree } x]. |pdevs-apply\ x\ i|$ )  
 by (metis atLeastLessThan-upt sum-list-rev rev-map sum-set-upt-conv-sum-list-nat)  
 also have  
 ... = sum-list (map ( $\lambda xa. |pdevs-apply\ x\ xa|$ ) [ $xa \leftarrow \text{rev } [0 .. < \text{degree } x]. pdevs-apply\ x\ xa \neq 0$ ])  
 unfolding filter-map map-map o-def  
 by (subst sum-list-map-filter) auto  
 also note sum-list-le-sum-list'[of - p]  
 also have [ $xa \leftarrow \text{rev } [0 .. < \text{degree } x]. pdevs-apply\ x\ xa \neq 0$ ] =  
 rev (sorted-list-of-set (pdevs-domain x))  
 by (subst rev-is-rev-conv[symmetric])  
 (auto simp: filter-map rev-filter intro!: sorted-distinct-set-unique  
 sorted-filter[of  $\lambda x. x$ , simplified] degree-gt)  
 finally  
 show tdev x  $\leq$  sum-list' p (map (abs  $\circ$  snd) (list-of-pdevs x))  
 by (auto simp: list-of-pdevs-def o-def rev-map filter-map rev-filter)  
 qed

**lemma** tdev'-le:  $x \leq \text{tdev } y \implies x \leq \text{tdev}'\ p\ y$   
 by (metis order.trans tdev')

**lemmas** abs-pdevs-val-le-tdev' = tdev'-le[OF abs-pdevs-val-le-tdev]

**lemma** tdev'-uminus-pdevs[simp]:  $\text{tdev}'\ p\ (\text{uminus-pdevs } x) = \text{tdev}'\ p\ x$   
 by (auto simp: tdev'-def o-def rev-map filter-map rev-filter list-of-pdevs-def pdevs-domain-def)

**abbreviation** Radius::'a::ordered-euclidean-space aform  $\Rightarrow$  'a  
 where Radius X  $\equiv$  tdev (snd X)

**abbreviation** Radius'::nat $\Rightarrow$ 'a::executable-euclidean-space aform  $\Rightarrow$  'a  
 where Radius' p X  $\equiv$  tdev' p (snd X)

**lemma** Radius'-uminus-aform[simp]: Radius' p (uminus-aform X) = Radius' p X  
 by (auto simp: uminus-aform-def)

### 5.1.3 truncate partial deviations

**definition** trunc-pdevs-raw::nat  $\Rightarrow$  (nat  $\Rightarrow$  'a)  $\Rightarrow$  nat  $\Rightarrow$  'a::executable-euclidean-space  
 where trunc-pdevs-raw p x i = eucl-truncate-down p (x i)

**lemma** nonzeros-trunc-pdevs-raw:  
 $\{i. \text{trunc-pdevs-raw } r\ x\ i \neq 0\} \subseteq \{i. x\ i \neq 0\}$   
 by (auto simp: trunc-pdevs-raw-def[abs-def])

**lift-definition** trunc-pdevs::nat  $\Rightarrow$  'a::executable-euclidean-space pdevs  $\Rightarrow$  'a pdevs  
 is trunc-pdevs-raw  
 by (auto intro!: finite-subset[OF nonzeros-trunc-pdevs-raw])

**definition**  $\text{trunc-err-pdevs-raw}::\text{nat} \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a::\text{executable-euclidean-space}$   
**where**  $\text{trunc-err-pdevs-raw } p \ x \ i = \text{trunc-pdevs-raw } p \ x \ i - x \ i$

**lemma**  $\text{nonzeros-trunc-err-pdevs-raw}$ :  
 $\{i. \text{trunc-err-pdevs-raw } r \ x \ i \neq 0\} \subseteq \{i. x \ i \neq 0\}$   
**by** ( $\text{auto simp: trunc-pdevs-raw-def trunc-err-pdevs-raw-def [abs-def]}$ )

**lift-definition**  $\text{trunc-err-pdevs}::\text{nat} \Rightarrow 'a::\text{executable-euclidean-space} \ \text{pdevs} \Rightarrow 'a$   
 $\text{pdevs}$   
**is**  $\text{trunc-err-pdevs-raw}$   
**by** ( $\text{auto intro!: finite-subset[OF nonzeros-trunc-err-pdevs-raw]}$ )

**term**  $\text{float-plus-down}$

**lemma**  $\text{pdevs-apply-trunc-pdevs[simp]}$ :  
**fixes**  $x \ y::'a::\text{euclidean-space}$   
**shows**  $\text{pdevs-apply } (\text{trunc-pdevs } p \ X) \ n = \text{eucl-truncate-down } p \ (\text{pdevs-apply } X \ n)$   
**by**  $\text{transfer (simp add: trunc-pdevs-raw-def)}$

**lemma**  $\text{pdevs-apply-trunc-err-pdevs[simp]}$ :  
**fixes**  $x \ y::'a::\text{euclidean-space}$   
**shows**  $\text{pdevs-apply } (\text{trunc-err-pdevs } p \ X) \ n = \text{eucl-truncate-down } p \ (\text{pdevs-apply } X \ n) - (\text{pdevs-apply } X \ n)$   
**by**  $\text{transfer (auto simp: trunc-err-pdevs-raw-def trunc-pdevs-raw-def)}$

**lemma**  $\text{pdevs-val-trunc-pdevs}$ :  
**fixes**  $x \ y::'a::\text{euclidean-space}$   
**shows**  $\text{pdevs-val } e \ (\text{trunc-pdevs } p \ X) = \text{pdevs-val } e \ X + \text{pdevs-val } e \ (\text{trunc-err-pdevs } p \ X)$   
**proof** –  
**have**  $\text{pdevs-val } e \ X + \text{pdevs-val } e \ (\text{trunc-err-pdevs } p \ X) = \text{pdevs-val } e \ (\text{add-pdevs } X \ (\text{trunc-err-pdevs } p \ X))$   
**by**  $\text{simp}$   
**also have**  $\dots = \text{pdevs-val } e \ (\text{trunc-pdevs } p \ X)$   
**by** ( $\text{auto simp: pdevs-val-def trunc-pdevs-raw-def trunc-err-pdevs-raw-def}$ )  
**finally show**  $?thesis$  **by**  $\text{simp}$   
**qed**

**lemma**  $\text{pdevs-val-trunc-err-pdevs}$ :  
**fixes**  $x \ y::'a::\text{euclidean-space}$   
**shows**  $\text{pdevs-val } e \ (\text{trunc-err-pdevs } p \ X) = \text{pdevs-val } e \ (\text{trunc-pdevs } p \ X) - \text{pdevs-val } e \ X$   
**by** ( $\text{simp add: pdevs-val-trunc-pdevs}$ )

**definition**  $\text{truncate-aform}::\text{nat} \Rightarrow 'a \ \text{aform} \Rightarrow 'a::\text{executable-euclidean-space} \ \text{aform}$   
**where**  $\text{truncate-aform } p \ x = (\text{eucl-truncate-down } p \ (\text{fst } x), \text{trunc-pdevs } p \ (\text{snd } x))$

**definition** *truncate-error-aform*::nat  $\Rightarrow$  'a aform  $\Rightarrow$  'a::executable-euclidean-space aform

**where** *truncate-error-aform* p x =  
 (eucl-truncate-down p (fst x) - fst x, trunc-err-pdevs p (snd x))

**lemma**

*abs-aform-val-le*:

**assumes**  $e \in UNIV \rightarrow \{-1..1\}$

**shows**  $abs (aform\text{-}val\ e\ X) \leq eucl\text{-}truncate\text{-}up\ p\ (|fst\ X| + tdev'\ p\ (snd\ X))$

**proof** -

**have**  $abs (aform\text{-}val\ e\ X) \leq |fst\ X| + |pdevs\text{-}val\ e\ (snd\ X)|$

**by** (*auto simp: aform-val-def intro!: abs-triangle-ineq*)

**also have**  $|pdevs\text{-}val\ e\ (snd\ X)| \leq tdev\ (snd\ X)$

**using** *assms* **by** (*rule abs-pdevs-val-le-tdev*)

**also note** *tdev'*

**also note** *eucl-truncate-up*

**finally show** *?thesis* **by** *simp*

**qed**

#### 5.1.4 truncation with error bound

**definition** *trunc-bound-eucl* p s =

(let  
 d = eucl-truncate-down p s;  
 ed = abs (d - s) in  
 (d, eucl-truncate-up p ed))

**lemma** *trunc-bound-euclE*:

**obtains** *err* **where**

$|err| \leq snd (trunc\text{-}bound\text{-}eucl\ p\ x)$

$fst (trunc\text{-}bound\text{-}eucl\ p\ x) = x + err$

**proof** *atomize-elim*

**have**  $fst (trunc\text{-}bound\text{-}eucl\ p\ x) = x + (eucl\text{-}truncate\text{-}down\ p\ x - x)$

(**is** - = - + *?err*)

**by** (*simp-all add: trunc-bound-eucl-def Let-def*)

**moreover have**  $abs\ ?err \leq snd (trunc\text{-}bound\text{-}eucl\ p\ x)$

**by** (*simp add: trunc-bound-eucl-def Let-def eucl-truncate-up*)

**ultimately show**  $\exists err. |err| \leq snd (trunc\text{-}bound\text{-}eucl\ p\ x) \wedge fst (trunc\text{-}bound\text{-}eucl\ p\ x) = x + err$

**by** *auto*

**qed**

**definition** *trunc-bound-pdevs* p x = (trunc-pdevs p x, tdev' p (trunc-err-pdevs p x))

**lemma** *pdevs-apply-fst-trunc-bound-pdevs*[*simp*]:  $pdevs\text{-}apply\ (fst\ (trunc\text{-}bound\text{-}pdevs\ p\ x)) =$

$pdevs\text{-}apply\ (trunc\text{-}pdevs\ p\ x)$

**by** (*simp add: trunc-bound-pdevs-def*)



**lemma** *trunc-bound-pdevsE*:  
**assumes**  $e \in UNIV \rightarrow \{-1..1\}$   
**obtains** *err* **where**  
 $|err| \leq snd (trunc-bound-pdevs p x)$   
 $pdevs-val e (fst ((trunc-bound-pdevs p x))) = pdevs-val e x + err$   
**proof** *atomize-elim*  
**have**  $pdevs-val e (fst (trunc-bound-pdevs p x)) = pdevs-val e x +$   
 $pdevs-val e (add-pdevs (trunc-pdevs p x) (uminus-pdevs x))$   
**(is**  $- = - + ?err$ **)**  
**by** (*simp-all add: trunc-bound-pdevs-def Let-def*)  
**moreover** **have**  $abs ?err \leq snd (trunc-bound-pdevs p x)$   
**using** *assms*  
**by** (*auto simp add: pdevs-val-trunc-pdevs trunc-bound-pdevs-def Let-def eucl-truncate-up*  
*intro!: order-trans[OF abs-pdevs-val-le-tdev]*)  
**ultimately** **show**  $\exists err. |err| \leq snd (trunc-bound-pdevs p x) \wedge$   
 $pdevs-val e (fst ((trunc-bound-pdevs p x))) = pdevs-val e x + err$   
**by** *auto*  
**qed**

**lemma**  
*degree-add-pdevs-le*:  
**assumes**  $degree X \leq n$   
**assumes**  $degree Y \leq n$   
**shows**  $degree (add-pdevs X Y) \leq n$   
**using** *assms*  
**by** (*auto intro!: degree-le*)

**lemma** *truncate-aform-error-aform-cancel*:  
 $aform-val e (truncate-aform p z) = aform-val e z + aform-val e (truncate-error-aform$   
 $p z)$   
**by** (*simp add: truncate-aform-def aform-val-def truncate-error-aform-def pdevs-val-trunc-pdevs*)

**lemma** *error-absE*:  
**assumes**  $abs err \leq k$   
**obtains**  $e::real$  **where**  $err = e * k$   $e \in \{-1 .. 1\}$   
**using** *assms*  
**by** *atomize-elim*  
*(safe intro!: exI[where x=err / abs k] divide-atLeastAtMost-1-absI, auto)*

**lemma** *eucl-truncate-up-nonneg-eq-zero-iff*:  
 $x \geq 0 \implies eucl-truncate-up p x = 0 \iff x = 0$   
**by** (*metis (poly-guards-query) eq-iff eucl-truncate-up eucl-truncate-up-zero*)

**lemma**  
*aform-val-consume-error*:  
**assumes**  $abs err \leq abs (pdevs-apply (snd X) n)$   
**shows**  $aform-val (e(n := 0)) X + err = aform-val (e(n := err / pdevs-apply (snd$

$X) n)) X$   
**using** *assms*  
**by** (*auto simp add: aform-val-def*)

**lemma**

*aform-val-consume-errorE:*  
**fixes**  $X::\text{real aform}$   
**assumes**  $\text{abs } err \leq \text{abs } (pdevs\text{-apply } (snd\ X) n)$   
**obtains**  $err'$  **where**  $\text{aform-val } (e(n := 0)) X + err = \text{aform-val } (e(n := err'))$   
 $X\ err' \in \{-1 .. 1\}$   
**by** *atomize-elim*  
*(rule aform-val-consume-error assms aform-val-consume-error exI conjI*  
*divide-atLeastAtMost-1-absI)+*

**lemma**

*degree-trunc-pdevs-le:*  
**assumes**  $\text{degree } X \leq n$   
**shows**  $\text{degree } (\text{trunc-pdevs } p\ X) \leq n$   
**using** *assms*  
**by** (*auto intro!: degree-le*)

**lemma** *pdevs-val-sum-less-degree:*

$\text{pdevs-val } e\ X = (\sum i < d. e\ i *_R\ \text{pdevs-apply } X\ i)$  **if**  $\text{degree } X \leq d$   
**unfolding** *pdevs-val-pdevs-domain*  
**apply** (*rule sum.mono-neutral-cong-left*)  
**using** *that*  
**by** *force+*

### 5.1.5 general affine operation

**definition** *affine-binop* ( $X::\text{real aform}$ )  $Y\ a\ b\ c\ d\ k =$

$(a *_R\ \text{fst } X + b *_R\ \text{fst } Y + c,$   
 $\text{pdev-upd } (\text{add-pdevs } (\text{scaleR-pdevs } a\ (snd\ X))\ (\text{scaleR-pdevs } b\ (snd\ Y)))\ k\ d)$

**lemma** *pdevs-domain-One-pdevs[simp]: pdevs-domain (One-pdevs::'a::executable-euclidean-space pdevs) =*

$\{0..<DIM('a)\}$   
**apply** (*auto simp: length-Basis-list split: if-splits*)  
**subgoal for**  $i$   
**using** *nth-Basis-list-in-Basis[of i, where 'a='a]*  
**by** (*auto simp: length-Basis-list*)  
**done**

**lemma** *pdevs-val-One-pdevs:*

$\text{pdevs-val } e\ (\text{One-pdevs::'a::executable-euclidean-space pdevs}) = (\sum i < DIM('a). e\ i *_R\ \text{Basis-list } !\ i)$   
**by** (*auto simp: pdevs-val-pdevs-domain length-Basis-list intro!:sum.cong*)

**lemma** *affine-binop:*

**assumes** *degree-aforms*  $[X, Y] \leq k$   
**shows** *aform-val*  $e$  (*affine-binop*  $X Y a b c d k$ ) =  
 $a * \text{aform-val } e X + b * \text{aform-val } e Y + c + e k * d$   
**using** *assms*  
**by** (*auto simp: aform-val-def affine-binop-def degrees-def*  
*pdevs-val-msum-pdevs degree-add-pdevs-le pdevs-val-One-pdevs Basis-list-real-def*  
*algebra-simps*)

**definition** *affine-binop'*  $p (X::\text{real aform}) Y a b c d k =$   
(*let*  
— **TODO:** more round-off operations here?  
 $(r, e1) = \text{trunc-bound-eucl } p (a * \text{fst } X + b * \text{fst } Y + c);$   
 $(Z, e2) = \text{trunc-bound-pdevs } p (\text{add-pdevs } (\text{scaleR-pdevs } a (\text{snd } X)) (\text{scaleR-pdevs } b (\text{snd } Y)))$   
*in*  
 $(r, \text{pdev-upd } Z k (\text{sum-list}' p [e1, e2, d]))$   
)

**lemma** *sum-list'-nonneg-eq-zero-iff*:  $\text{sum-list}' p xs = 0 \iff (\forall x \in \text{set } xs. x = 0)$  **if**  
 $\bigwedge x. x \in \text{set } xs \implies x \geq 0$

**proof** *safe*

**fix**  $x$  **assume**  $x: \text{sum-list}' p xs = 0 \ x \in \text{set } xs$   
**from** *that* **have**  $0 \leq \text{sum-list } xs$  **by** (*auto intro!: sum-list-nonneg*)  
**with** *that*  $x$  **have**  $\text{sum-list } xs = 0$   
**by** (*metis antisym sum-list-le-sum-list'*)  
**then** **have**  $(\sum_{i < \text{length } xs} xs ! i) = 0$   
**by** (*auto simp: sum-list-sum-nth atLeast0LessThan*)  
**then** **show**  $x = 0$  **using**  $x(2)$  **that**  
**by** (*subst (asm) sum-nonneg-eq-0-iff*) (*auto simp: in-set-conv-nth*)

**next**

**show**  $\forall x \in \text{set } xs. x = 0 \implies \text{sum-list}' p xs = 0$   
**by** (*induction xs*) (*auto simp: sum-list'-def*)

**qed**

**lemma** *affine-binop'E*:

**assumes** *deg: degree-aforms*  $[X, Y] \leq k$

**assumes**  $e: e \in \text{UNIV} \rightarrow \{-1..1\}$

**assumes**  $d: \text{abs } u \leq d$

**obtains**  $ek$  **where**

$a * \text{aform-val } e X + b * \text{aform-val } e Y + c + u = \text{aform-val } (e(k:=ek))$

(*affine-binop'*  $p X Y a b c d k$ )

$ek \in \{-1 .. 1\}$

**proof** —

**have**  $a * \text{aform-val } e X + b * \text{aform-val } e Y + c + u =$

$(a * \text{fst } X + b * \text{fst } Y + c) + \text{pdevs-val } e (\text{add-pdevs } (\text{scaleR-pdevs } a (\text{snd } X))$

$(\text{scaleR-pdevs } b (\text{snd } Y))) + u$

(**is**  $- = ?c + \text{pdevs-val } - ?ps + -$ )

**by** (*auto simp: aform-val-def algebra-simps*)

```

from trunc-bound-euclE[of p ?c] obtain ec where ec:  $\text{abs } ec \leq \text{snd } (\text{trunc-bound-eucl } p \ ?c)$ 
  fst (trunc-bound-eucl p ?c) - ec = ?c
  by (auto simp: algebra-simps)

moreover

from trunc-bound-pdevsE[OF e, of p ?ps]
obtain eps where eps:  $|\text{eps}| \leq \text{snd } (\text{trunc-bound-pdevs } p \ ?ps)$ 
  pdevs-val e (fst (trunc-bound-pdevs p ?ps)) - eps = pdevs-val e ?ps
  by (auto simp: algebra-simps)

moreover
define ek where  $ek = (u - ec - eps) /$ 
   $\text{sum-list}' \ p \ [\text{snd } (\text{trunc-bound-eucl } p \ ?c), \text{snd } (\text{trunc-bound-pdevs } p \ ?ps), \ d]$ 
have  $\text{degree } (\text{fst } (\text{trunc-bound-pdevs } p \ ?ps)) \leq$ 
   $\text{degree-aforms } [X, Y]$ 
  by (auto simp: trunc-bound-pdevs-def degrees-def intro!: degree-trunc-pdevs-le
  degree-add-pdevs-le)
moreover
from this have pdevs-apply (fst (trunc-bound-pdevs p ?ps)) k = 0
  using deg order-trans by blast
ultimately have  $a * \text{aform-val } e \ X + b * \text{aform-val } e \ Y + c + u =$ 
   $\text{aform-val } (e(k:=ek)) \ (\text{affine-binop}' \ p \ X \ Y \ a \ b \ c \ d \ k)$ 
apply (auto simp: affine-binop'-def algebra-simps aform-val-def split: prod.splits)
subgoal for x y z
  apply (cases sum-list' p [x, z, d] = 0)
  subgoal
    apply simp
    apply (subst (asm) sum-list'-noneg-eq-zero-iff)
    using d deg
    by auto
  subgoal
    apply (simp add: divide-simps algebra-simps ek-def)
    using (pdevs-apply (fst (trunc-bound-pdevs p (add-pdevs (scaleR-pdevs a (snd
  X)) (scaleR-pdevs b (snd Y)))))) k = 0) by auto
  done
done
moreover have  $ek \in \{-1 .. 1\}$ 
  unfolding ek-def
  apply (rule divide-atLeastAtMost-1-absI)
  apply (rule abs-triangle-ineq4 [THEN order-trans])
  apply (rule order-trans)
  apply (rule add-right-mono)
  apply (rule abs-triangle-ineq4)
  using ec(1) eps(1)
  by (auto simp: sum-list'-def eucl-truncate-up-real-def add.assoc
  intro!: order-trans[OF - abs-ge-self] order-trans[OF - truncate-up-le] add-mono
  d )

```

ultimately show *?thesis ..*  
 qed

### 5.1.6 Inf/Sup

**definition**  $\text{Inf-aform}' p X = \text{eucl-truncate-down } p (\text{fst } X - \text{tdev}' p (\text{snd } X))$

**definition**  $\text{Sup-aform}' p X = \text{eucl-truncate-up } p (\text{fst } X + \text{tdev}' p (\text{snd } X))$

**lemma**  $\text{Inf-aform}'$ :

**shows**  $\text{Inf-aform}' p X \leq \text{Inf-aform } X$

**unfolding**  $\text{Inf-aform-def } \text{Inf-aform}'\text{-def}$

**by** (*auto intro!*:  $\text{eucl-truncate-down-le add-left-mono tdev}'$ )

**lemma**  $\text{Sup-aform}'$ :

**shows**  $\text{Sup-aform } X \leq \text{Sup-aform}' p X$

**unfolding**  $\text{Sup-aform-def } \text{Sup-aform}'\text{-def}$

**by** (*rule eucl-truncate-up-le add-left-mono tdev}'*)+

**lemma**  $\text{Inf-aform-le-Sup-aform}[intro]$ :

$\text{Inf-aform } X \leq \text{Sup-aform } X$

**by** (*simp add: Inf-aform-def Sup-aform-def algebra-simps*)

**lemma**  $\text{Inf-aform}'\text{-le-Sup-aform}'[intro]$ :

$\text{Inf-aform}' p X \leq \text{Sup-aform}' p X$

**by** (*metis Inf-aform' Inf-aform-le-Sup-aform Sup-aform' order.trans*)

**definition**

$\text{ivls-of-aforms } \text{prec} = \text{map } (\lambda a. \text{Interval}' (\text{float-of } (\text{Inf-aform}' \text{prec } a)) (\text{float-of } (\text{Sup-aform}' \text{prec } a)))$

**lemma**

**assumes**  $\bigwedge i. e'' i \leq 1$

**assumes**  $\bigwedge i. -1 \leq e'' i$

**shows**  $\text{Inf-aform}'\text{-le: } \text{Inf-aform}' p r \leq \text{aform-val } e'' r$

**and**  $\text{Sup-aform}'\text{-le: } \text{aform-val } e'' r \leq \text{Sup-aform}' p r$

**by** (*auto intro!*:  $\text{order-trans}[OF \text{Inf-aform}' ] \text{order-trans}[OF - \text{Sup-aform}' ] \text{Inf-aform } \text{Sup-aform}$

*simp: Affine-def valuate-def intro!*:  $\text{image-eqI}[\mathbf{where } x=e''] \text{assms}$ )

**lemma**  $\text{InfSup-aform}'\text{-in-float}[intro, simp]$ :

$\text{Inf-aform}' p X \in \text{float } \text{Sup-aform}' p X \in \text{float}$

**by** (*auto simp: Inf-aform'\text{-def eucl-truncate-down-real-def*

$\text{Sup-aform}'\text{-def eucl-truncate-up-real-def}$ )

**theorem**  $\text{ivls-of-aforms: } xs \in \text{Joints } XS \implies \text{bounded-by } xs (\text{ivls-of-aforms } \text{prec } XS)$

**by** (*auto simp: bounded-by-def ivls-of-aforms-def Affine-def valuate-def Pi-iff*)

*set-of-eq*  
*intro!*: *Inf-aform'-le Sup-aform'-le*  
*dest!*: *nth-in-AffineI split: Interval'-splits*)

**definition** *isFDERIV-aform prec N xs fas AS = isFDERIV-approx prec N xs fas*  
*(ivls-of-aforms prec AS)*

**theorem** *isFDERIV-aform:*  
**assumes** *isFDERIV-aform prec N xs fas AS*  
**assumes** *vs ∈ Joints AS*  
**shows** *isFDERIV N xs fas vs*  
**apply** (*rule isFDERIV-approx*)  
**apply** (*rule ivls-of-aforms*)  
**apply** (*rule assms*)  
**apply** (*rule assms[unfolded isFDERIV-aform-def]*)  
**done**

**definition** *env-len env l = (∀ xs ∈ env. length xs = l)*

**lemma** *env-len-takeI: env-len xs d1 ⇒ d1 ≥ d ⇒ env-len (take d ' xs) d*  
**by** (*auto simp: env-len-def*)

## 5.2 Min Range approximation

**lemma**

*linear-lower:*

**fixes** *x::real*

**assumes**  $\bigwedge x. x \in \{a .. b\} \implies (f \text{ has-field-derivative } f' x) \text{ (at } x \text{ within } \{a .. b\})$

**assumes**  $\bigwedge x. x \in \{a .. b\} \implies f' x \leq u$

**assumes**  $x \in \{a .. b\}$

**shows**  $f b + u * (x - b) \leq f x$

**proof** –

**from** *assms(2-)*

*mvt-very-simple[of x b f λx. (\*) (f' x),*

*rule-format,*

*OF - has-derivative-subset[OF assms(1)[simplified has-field-derivative-def]]]*

**obtain** *y where*  $y \in \{x .. b\}$   $f b - f x = (b - x) * f' y$

**by** (*auto simp: Bex-def ac-simps*)

**moreover hence**  $f' y \leq u$  **using** *assms* **by** *auto*

**ultimately have**  $f b - f x \leq (b - x) * u$

**by** (*auto intro!: mult-left-mono*)

**thus** *?thesis* **by** (*simp add: algebra-simps*)

**qed**

**lemma**

*linear-lower2:*

**fixes** *x::real*

**assumes**  $\bigwedge x. x \in \{a .. b\} \implies (f \text{ has-field-derivative } f' x) \text{ (at } x \text{ within } \{a .. b\})$

**assumes**  $\bigwedge x. x \in \{a .. b\} \implies l \leq f' x$

**assumes**  $x \in \{a .. b\}$   
**shows**  $f x \geq f a + l * (x - a)$   
**proof** –  
**from** *assms(2-)*  
*mut-very-simple*[of  $a x f \lambda x. (*) (f' x)$ ,  
*rule-format*,  
*OF - has-derivative-subset*[*OF assms(1)*[*simplified has-field-derivative-def*]]]  
**obtain**  $y$  **where**  $y \in \{a .. x\}$   $f x - f a = (x - a) * f' y$   
**by** (*auto simp: Bex-def ac-simps*)  
**moreover hence**  $l \leq f' y$  **using** *assms* **by** *auto*  
**ultimately have**  $(x - a) * l \leq f x - f a$   
**by** (*auto intro!: mult-left-mono*)  
**thus** *?thesis* **by** (*simp add: algebra-simps*)  
**qed**

**lemma**

*linear-upper:*  
**fixes**  $x::real$   
**assumes**  $\bigwedge x. x \in \{a .. b\} \implies (f \text{ has-field-derivative } f' x) \text{ (at } x \text{ within } \{a .. b\})$   
**assumes**  $\bigwedge x. x \in \{a .. b\} \implies f' x \leq u$   
**assumes**  $x \in \{a .. b\}$   
**shows**  $f x \leq f a + u * (x - a)$   
**proof** –  
**from** *assms(2-)*  
*mut-very-simple*[of  $a x f \lambda x. (*) (f' x)$ ,  
*rule-format*,  
*OF - has-derivative-subset*[*OF assms(1)*[*simplified has-field-derivative-def*]]]  
**obtain**  $y$  **where**  $y \in \{a .. x\}$   $f x - f a = (x - a) * f' y$   
**by** (*auto simp: Bex-def ac-simps*)  
**moreover hence**  $f' y \leq u$  **using** *assms* **by** *auto*  
**ultimately have**  $(x - a) * u \geq f x - f a$   
**by** (*auto intro!: mult-left-mono*)  
**thus** *?thesis* **by** (*simp add: algebra-simps*)  
**qed**

**lemma**

*linear-upper2:*  
**fixes**  $x::real$   
**assumes**  $\bigwedge x. x \in \{a .. b\} \implies (f \text{ has-field-derivative } f' x) \text{ (at } x \text{ within } \{a .. b\})$   
**assumes**  $\bigwedge x. x \in \{a .. b\} \implies l \leq f' x$   
**assumes**  $x \in \{a .. b\}$   
**shows**  $f x \leq f b + l * (x - b)$   
**proof** –  
**from** *assms(2-)*  
*mut-very-simple*[of  $x b f \lambda x. (*) (f' x)$ ,  
*rule-format*,  
*OF - has-derivative-subset*[*OF assms(1)*[*simplified has-field-derivative-def*]]]  
**obtain**  $y$  **where**  $y \in \{x .. b\}$   $f b - f x = (b - x) * f' y$   
**by** (*auto simp: Bex-def ac-simps*)

**moreover hence**  $l \leq f' y$  **using** *assms* **by** *auto*  
**ultimately have**  $f b - f x \geq (b - x) * l$   
**by** (*auto intro! mult-left-mono*)  
**thus ?thesis by** (*simp add: algebra-simps*)  
**qed**

**lemma** *linear-enclosure*:

**fixes**  $x::real$   
**assumes**  $\bigwedge x. x \in \{a .. b\} \implies (f \text{ has-field-derivative } f' x) \text{ (at } x \text{ within } \{a .. b\})$   
**assumes**  $\bigwedge x. x \in \{a .. b\} \implies f' x \leq u$   
**assumes**  $x \in \{a .. b\}$   
**shows**  $f x \in \{f b + u * (x - b) .. f a + u * (x - a)\}$   
**using** *linear-lower[OF assms] linear-upper[OF assms]*  
**by** *auto*

**definition** *mid-err ivl* =  $((\text{lower } ivl + \text{upper } ivl::float)/2, (\text{upper } ivl - \text{lower } ivl)/2)$

**lemma** *degree-aform-uminus-aform[simp]*:  $\text{degree-aform } (uminus\text{-aform } X) = \text{degree-aform } X$   
**by** (*auto simp: uminus-aform-def*)

### 5.2.1 Addition

**definition** *add-aform::'a::real-vector aform*  $\Rightarrow 'a \text{ aform} \Rightarrow 'a \text{ aform}$   
**where** *add-aform*  $x y = (fst x + fst y, add\text{-pdevs } (snd x) (snd y))$

**lemma** *aform-val-add-aform*:

**shows** *aform-val*  $e (add\text{-aform } X Y) = \text{aform-val } e X + \text{aform-val } e Y$   
**by** (*auto simp: add-aform-def aform-val-def*)

**type-synonym** *aform-err* =  $real \text{ aform} \times real$

**definition** *add-aform'::nat*  $\Rightarrow \text{aform-err} \Rightarrow \text{aform-err} \Rightarrow \text{aform-err}$   
**where** *add-aform'*  $p x y =$

*(let*  
 $z0 = \text{trunc-bound-eucl } p (fst (fst x) + fst (fst y));$   
 $z = \text{trunc-bound-pdevs } p (add\text{-pdevs } (snd (fst x)) (snd (fst y)))$   
*in*  $((fst z0, fst z), (\text{sum-list}' p [snd z0, snd z, abs (snd x), abs (snd y)]))$

**abbreviation** *degree-aform-err::aform-err*  $\Rightarrow nat$   
**where** *degree-aform-err*  $X \equiv \text{degree-aform } (fst X)$

**lemma** *degree-aform-err-add-aform'*:

**assumes** *degree-aform-err*  $x \leq n$   
**assumes** *degree-aform-err*  $y \leq n$   
**shows** *degree-aform-err*  $(add\text{-aform}' p x y) \leq n$   
**using** *assms*  
**by** (*auto simp: add-aform'-def Let-def trunc-bound-pdevs-def*)



*intro!*: *degree-pdev-upd-le degree-trunc-pdevs-le degree-add-pdevs-le*)

**definition** *aform-err*  $e X e = \{aform-val\ e\ (fst\ X e) - snd\ X e .. aform-val\ e\ (fst\ X e) + snd\ X e::real\}$

**lemma** *aform-errI*:  $x \in aform-err\ e\ X e$   
 if  $abs\ (x - aform-val\ e\ (fst\ X e)) \leq snd\ X e$   
 using *that* by (*auto simp: aform-err-def abs-real-def algebra-simps split: if-splits*)

**lemma** *add-aform'*:  
 assumes  $e: e \in UNIV \rightarrow \{-1..1\}$   
 assumes  $x: x \in aform-err\ e\ X$   
 assumes  $y: y \in aform-err\ e\ Y$   
 shows  $x + y \in aform-err\ e\ (add-aform'\ p\ X\ Y)$   
**proof** –  
 let  $?t1 = trunc-bound-eucl\ p\ (fst\ (fst\ X) + fst\ (fst\ Y))$   
 from *trunc-bound-euclE*  
 obtain  $e1$  where  $abs-e1: |e1| \leq snd\ ?t1$  and  $e1: fst\ ?t1 = fst\ (fst\ X) + fst\ (fst\ Y) + e1$   
 by *blast*  
 let  $?t2 = trunc-bound-pdevs\ p\ (add-pdevs\ (snd\ (fst\ X))\ (snd\ (fst\ Y)))$   
 from *trunc-bound-pdevsE[OF e, of p add-pdevs (snd (fst X)) (snd (fst Y))]*  
 obtain  $e2$  where  $abs-e2: |e2| \leq snd\ (?t2)$   
 and  $e2: pdevs-val\ e\ (fst\ ?t2) = pdevs-val\ e\ (add-pdevs\ (snd\ (fst\ X))\ (snd\ (fst\ Y))) + e2$   
 by *blast*  
  
 have  $e-le: |e1 + e2 + snd\ X + snd\ Y| \leq snd\ (add-aform'\ p\ (X)\ Y)$   
 apply (*auto simp: add-aform'-def Let-def*)  
 apply (*rule sum-list'-sum-list-le*)  
 apply (*simp add: add.assoc*)  
 by (*intro order.trans[OF abs-triangle-ineq] add-mono abs-e1 abs-e2 order-refl*)  
 then show *?thesis*  
 apply (*intro aform-errI*)  
 using  $x\ y\ abs-e1\ abs-e2$   
 apply (*simp add: aform-val-def aform-err-def add-aform-def add-aform'-def Let-def e1 e2 assms*)  
 by (*auto intro!: order-trans[OF - sum-list-le-sum-list']*)  
**qed**

## 5.2.2 Scaling

**definition** *aform-scaleR::real aform*  $\Rightarrow 'a::real-vector \Rightarrow 'a\ aform$   
 where  $aform-scaleR\ x\ y = (fst\ x *_R\ y, pdevs-scaleR\ (snd\ x)\ y)$

**lemma** *aform-val-scaleR-aform[simp]*:  
 shows  $aform-val\ e\ (aform-scaleR\ X\ y) = aform-val\ e\ X *_R\ y$   
 by (*auto simp: aform-scaleR-def aform-val-def scaleR-left-distrib*)

### 5.2.3 Multiplication

**lemma** *aform-val-mult-exact*:

*aform-val*  $e x * aform-val e y =$   
 $fst x * fst y +$   
 $pdevs-val e (add-pdevs (scaleR-pdevs (fst y) (snd x)) (scaleR-pdevs (fst x) (snd y))) +$   
 $(\sum i < d. e i *_R pdevs-apply (snd x) i) * (\sum i < d. e i *_R pdevs-apply (snd y) i)$   
**if**  $degree (snd x) \leq d$   $degree (snd y) \leq d$   
**using** *that*  
**by** (*auto simp: pdevs-val-sum-less-degree* [**where**  $d=d$ ] *aform-val-def algebra-simps*)

**lemma** *sum-times-bound*:— **TODO**: this gives better bounds for the remainder of multiplication

$(\sum i < d. e i * f i :: real) * (\sum i < d. e i * g i) =$   
 $(\sum i < d. (e i)^2 * (f i * g i)) +$   
 $(\sum (i, j) | i < j \wedge j < d. (e i * e j) * (f j * g i + f i * g j))$  **for**  $d :: nat$

**proof** —

**have**  $(\sum i < d. e i * f i) * (\sum i < d. e i * g i) = (\sum (i, j) \in \{..<d\} \times \{..<d\}. e i * f i * e j * g j)$

**unfolding** *sum-product sum.cartesian-product ..*

**also have**  $\dots = (\sum (i, j) \in \{..<d\} \times \{..<d\} \cap \{(i, j). i = j\}. e i * f i * (e j * g j)) +$

$((\sum (i, j) \in \{..<d\} \times \{..<d\} \cap \{(i, j). i < j\}. e i * f i * (e j * g j)) +$

$(\sum (i, j) \in \{..<d\} \times \{..<d\} \cap \{(i, j). j < i\}. e i * f i * (e j * g j)))$

**(is - = ?a + (?b + ?c))**

**by** (*subst sum.union-disjoint* [*symmetric*], *force*, *force*, *force*) + (*auto intro!*: *sum.cong*)

**also have**  $?c = (\sum (i, j) \in \{..<d\} \times \{..<d\} \cap \{(i, j). i < j\}. e i * f j * (e j * g i))$

**by** (*rule sum.reindex-cong* [*of*  $\lambda(x, y). (y, x)$ ]) (*auto intro!*: *inj-onI*)

**also have**  $?b + \dots = (\sum (i, j) \in \{..<d\} \times \{..<d\} \cap \{(i, j). i < j\}. (e i * e j) * (f j * g i + f i * g j))$

**by** (*auto simp: algebra-simps sum.distrib split-beta'*)

**also have**  $\dots = (\sum (i, j) | i < j \wedge j < d. (e i * e j) * (f j * g i + f i * g j))$

**by** (*rule sum.cong*) *auto*

**also have**  $?a = (\sum i < d. (e i)^2 * (f i * g i))$

**by** (*rule sum.reindex-cong* [*of*  $\lambda i. (i, i)$ ]) (*auto simp: power2-eq-square intro!*: *inj-onI*)

**finally show** *?thesis by simp*

**qed**

**definition** *mult-aform::aform-err*  $\Rightarrow aform-err \Rightarrow aform-err$

**where** *mult-aform*  $x y = ((fst (fst x) * fst (fst y),$

$(add-pdevs (scaleR-pdevs (fst (fst y)) (snd (fst x))) (scaleR-pdevs (fst (fst x)) (snd (fst y))))),$

$(tdev (snd (fst x)) * tdev (snd (fst y)) +$

$abs (snd x) * (abs (fst (fst y)) + Radius (fst y)) +$

$abs (snd y) * (abs (fst (fst x)) + Radius (fst x)) + abs (snd x) * abs (snd y)$

$))$

```

lemma mult-aformE:
  fixes X Y::aform-err
  assumes e:  $e \in UNIV \rightarrow \{-1..1\}$ 
  assumes x:  $x \in aform-err\ e\ X$ 
  assumes y:  $y \in aform-err\ e\ Y$ 
  shows  $x * y \in aform-err\ e\ (mult-aform\ X\ Y)$ 
proof -
  define ex where  $ex \equiv x - aform-val\ e\ (fst\ X)$ 
  define ey where  $ey \equiv y - aform-val\ e\ (fst\ Y)$ 

  have [intro, simp]:  $|ex| \leq |snd\ X| \ |ey| \leq |snd\ Y|$ 
    using x y
    by (auto simp: ex-def ey-def aform-err-def)
  have  $x * y =$ 
     $fst\ (fst\ X) * fst\ (fst\ Y) +$ 
     $fst\ (fst\ Y) * pdevs-val\ e\ (snd\ (fst\ X)) +$ 
     $fst\ (fst\ X) * pdevs-val\ e\ (snd\ (fst\ Y)) +$ 
     $(pdevs-val\ e\ (snd\ (fst\ X)) * pdevs-val\ e\ (snd\ (fst\ Y)) +$ 
     $ex * (fst\ (fst\ Y) + pdevs-val\ e\ (snd\ (fst\ Y))) +$ 
     $ey * (fst\ (fst\ X) + pdevs-val\ e\ (snd\ (fst\ X))) +$ 
     $ex * ey)$ 
    (is - = ?c + ?d + ?e + ?err)
    by (auto simp: ex-def ey-def algebra-simps aform-val-def)

  have abs-err:  $abs\ ?err \leq snd\ (mult-aform\ X\ Y)$ 
    by (auto simp: mult-aform-def abs-mult
      intro!: abs-triangle-ineq[THEN order-trans] add-mono mult-mono
      abs-pdevs-val-le-tdev e)
  show ?thesis
    apply (auto simp: intro!: aform-errI order-trans[OF - abs-err])
    apply (subst mult-aform-def)
    apply (auto simp: aform-val-def ex-def ey-def algebra-simps)
  done
qed

```

```

definition mult-aform'::nat  $\Rightarrow$  aform-err  $\Rightarrow$  aform-err  $\Rightarrow$  aform-err
  where mult-aform' p x y = (
    let
       $(fx, sx) = x;$ 
       $(fy, sy) = y;$ 
       $ex = abs\ sx;$ 
       $ey = abs\ sy;$ 
       $z0 = trunc-bound-eucl\ p\ (fst\ fx * fst\ fy);$ 
       $u = trunc-bound-pdevs\ p\ (scaleR-pdevs\ (fst\ fy)\ (snd\ fx));$ 
       $v = trunc-bound-pdevs\ p\ (scaleR-pdevs\ (fst\ fx)\ (snd\ fy));$ 
       $w = trunc-bound-pdevs\ p\ (add-pdevs\ (fst\ u)\ (fst\ v));$ 
       $tx = tdev'\ p\ (snd\ fx);$ 

```

```

    ty = tdev' p (snd fy);
    l = truncate-up p (tx * ty);
    ee = truncate-up p (ex * ey);
    e1 = truncate-up p (ex * truncate-up p (abs (fst fy) + ty));
    e2 = truncate-up p (ey * truncate-up p (abs (fst fx) + tx))
  in
    ((fst z0, (fst w)), (sum-list' p [ee, e1, e2, l, snd z0, snd u, snd v, snd w]))

```

**lemma** *aform-errE*:

```

  abs (x - aform-val e (fst X)) ≤ snd X
  if x ∈ aform-err e X
  using that by (auto simp: aform-err-def)

```

**lemma** *mult-aform'E*:

```

  fixes X Y::aform-err
  assumes e: e ∈ UNIV → {- 1..1}
  assumes x: x ∈ aform-err e X
  assumes y: y ∈ aform-err e Y
  shows x * y ∈ aform-err e (mult-aform' p X Y)
  proof -
    let ?z0 = trunc-bound-eucl p (fst (fst X) * fst (fst Y))
    from trunc-bound-euclE
    obtain e1 where abs-e1: |e1| ≤ snd ?z0 and e1: fst ?z0 = fst (fst X) * fst (fst
  Y) + e1
      by blast
    let ?u = trunc-bound-pdevs p (scaleR-pdevs (fst (fst Y)) (snd (fst X)))
    from trunc-bound-pdevsE[OF e]
    obtain e2 where abs-e2: |e2| ≤ snd (?u)
      and e2: pdevs-val e (fst ?u) = pdevs-val e (scaleR-pdevs (fst (fst Y)) (snd (fst
  X))) + e2
      by blast
    let ?v = trunc-bound-pdevs p (scaleR-pdevs (fst (fst X)) (snd (fst Y)))
    from trunc-bound-pdevsE[OF e]
    obtain e3 where abs-e3: |e3| ≤ snd (?v)
      and e3: pdevs-val e (fst ?v) = pdevs-val e (scaleR-pdevs (fst (fst X)) (snd (fst
  Y))) + e3
      by blast
    let ?w = trunc-bound-pdevs p (add-pdevs (fst ?u) (fst ?v))
    from trunc-bound-pdevsE[OF e]
    obtain e4 where abs-e4: |e4| ≤ snd (?w)
      and e4: pdevs-val e (fst ?w) = pdevs-val e (add-pdevs (fst ?u) (fst ?v)) + e4
      by blast
    let ?tx = tdev' p (snd (fst X)) and ?ty = tdev' p (snd (fst Y))
    let ?l = truncate-up p (?tx * ?ty)
    let ?ee = truncate-up p (abs (snd X) * abs (snd Y))
    let ?e1 = truncate-up p (abs (snd X) * truncate-up p (|fst (fst Y)| + ?ty))
    let ?e2 = truncate-up p (abs (snd Y) * truncate-up p (|fst (fst X)| + ?tx))

    let ?e0 = x * y - fst (fst X) * fst (fst Y) -

```

```

      fst (fst X) * pdevs-val e (snd (fst Y)) -
      fst (fst Y) * pdevs-val e (snd (fst X))
let ?err = ?e0 - (e1 + e2 + e3 + e4)
have abs ?err ≤ abs ?e0 + abs e1 + abs e2 + abs e3 + abs e4
  by arith
also have ... ≤ abs ?e0 + snd ?z0 + snd ?u + snd ?v + snd ?w
  unfolding abs-mult
  by (auto intro!: add-mono mult-mono e abs-pdevs-val-le-tdev' abs-ge-zero abs-e1
abs-e2 abs-e3
abs-e4 intro: tdev'-le)
also
have asdf: snd (mult-aform X Y) ≤ tdev' p (snd (fst X)) * tdev' p (snd (fst Y))
+ ?e1 + ?e2 + ?ee
  by (auto simp: mult-aform-def intro!: add-mono mult-mono order-trans[OF -
tdev'] truncate-up-le)
have abs ?e0 ≤ ?ee + ?e1 + ?e2 + tdev' p (snd (fst X)) * tdev' p (snd (fst Y))
  using mult-aformE[OF e x y, THEN aform-errE, THEN order-trans, OF asdf]
  by (simp add: aform-val-def mult-aform-def) arith
also have tdev' p (snd (fst X)) * tdev' p (snd (fst Y)) ≤ ?l
  by (auto intro!: truncate-up-le)
also have ?ee + ?e1 + ?e2 + ?l + snd ?z0 + snd ?u + snd ?v + snd ?w ≤
sum-list' p [?ee, ?e1, ?e2, ?l, snd ?z0, snd ?u, snd ?v, snd ?w]
  by (rule order-trans[OF - sum-list-le-sum-list']) simp
also have ... ≤ (snd (mult-aform' p X Y))
  by (auto simp: mult-aform'-def Let-def assms split: prod.splits)
finally have err-le: abs ?err ≤ (snd (mult-aform' p X Y)) by arith

show ?thesis
  apply (rule aform-errI[OF order-trans[OF - err-le]])
  apply (subst mult-aform'-def)
  using e1 e2 e3 e4
  apply (auto simp: aform-val-def Let-def assms split: prod.splits)
  done
qed

```

```

lemma degree-aform-mult-aform':
  assumes degree-aform-err x ≤ n
  assumes degree-aform-err y ≤ n
  shows degree-aform-err (mult-aform' p x y) ≤ n
  using assms
  by (auto simp: mult-aform'-def Let-def trunc-bound-pdevs-def split: prod.splits
intro!: degree-pdev-upd-le degree-trunc-pdevs-le degree-add-pdevs-le)

```

```

lemma
  fixes x a b::real
  assumes a > 0
  assumes x ∈ {a ..b}
  assumes - inverse (b*b) ≤ alpha
  shows inverse-linear-lower: inverse b + alpha * (x - b) ≤ inverse x (is ?lower)

```

**and** *inverse-linear-upper*:  $\text{inverse } x \leq \text{inverse } a + \text{alpha} * (x - a)$  (**is** *?upper*)  
**proof** –  
**have** *deriv-inv*:  
 $\bigwedge x. x \in \{a .. b\} \implies (\text{inverse has-field-derivative} - \text{inverse } (x*x))$  (*at x within*  
 $\{a .. b\}$ )  
**using** *assms*  
**by** (*auto intro!*: *derivative-eq-intros*)  
**show** *?lower*  
**using** *assms*  
**by** (*intro linear-lower*[*OF deriv-inv*])  
*(auto simp: mult-mono intro! order-trans[OF - assms(3)])*  
**show** *?upper*  
**using** *assms*  
**by** (*intro linear-upper*[*OF deriv-inv*])  
*(auto simp: mult-mono intro! order-trans[OF - assms(3)])*  
**qed**

#### 5.2.4 Inverse

**definition** *inverse-aform'*::*nat*  $\Rightarrow$  *real aform*  $\Rightarrow$  *real aform*  $\times$  *real* **where**

*inverse-aform' p X* = (  
*let l = Inf-aform' p X in*  
*let u = Sup-aform' p X in*  
*let a = min (abs l) (abs u) in*  
*let b = max (abs l) (abs u) in*  
*let sq = truncate-up p (b \* b) in*  
*let alpha = - real-divl p 1 sq in*  
*let dmax = truncate-up p (real-divr p 1 a - alpha \* a) in*  
*let dmin = truncate-down p (real-divl p 1 b - alpha \* b) in*  
*let zeta' = truncate-up p ((dmin + dmax) / 2) in*  
*let zeta = if l < 0 then - zeta' else zeta' in*  
*let delta = truncate-up p (zeta - dmin) in*  
*let res1 = trunc-bound-eucl p (alpha \* fst X) in*  
*let res2 = trunc-bound-eucl p (fst res1 + zeta) in*  
*let zs = trunc-bound-pdevs p (scaleR-pdevs alpha (snd X)) in*  
*((fst res2, fst zs), (sum-list' p [delta, snd res1, snd res2, snd zs]))*)

**lemma** *inverse-aform'E*:

**fixes** *X*::*real aform*

**assumes** *e*:  $e \in UNIV \rightarrow \{-1 .. 1\}$

**assumes** *Inf-pos*: *Inf-aform' p X* > 0

**assumes** *x = aform-val e X*

**shows**  $\text{inverse } x \in \text{aform-err } e$  (*inverse-aform' p X*)

**proof** –

**define** *l* **where** *l = Inf-aform' p X*

**define** *u* **where** *u = Sup-aform' p X*

**define** *a* **where** *a = min (abs l) (abs u)*

**define** *b* **where** *b = max (abs l) (abs u)*

**define** *sq* **where** *sq = truncate-up p (b \* b)*

```

define alpha where alpha = - (real-divl p 1 sq)
define d-max' where d-max' = truncate-up p (real-divr p 1 a - alpha * a)
define d-min' where d-min' = truncate-down p (real-divl p 1 b - alpha * b)
define zeta where zeta = truncate-up p ((d-min' + d-max') / 2)
define delta where delta = truncate-up p (zeta - d-min')
note vars = l-def u-def a-def b-def sq-def alpha-def d-max'-def d-min'-def zeta-def
delta-def
let ?x = aform-val e X

have 0 < l using assms by (auto simp add: l-def Inf-aform-def)
have l ≤ u by (auto simp: l-def u-def)

hence a-def': a = l and b-def': b = u and 0 < a 0 < b
using ⟨0 < l⟩ by (simp-all add: a-def b-def)
have 0 < ?x
by (rule less-le-trans[OF Inf-pos order.trans[OF Inf-aform' Inf-aform], OF e])
have a ≤ ?x
by (metis order.trans Inf-aform e Inf-aform' a-def' l-def)
have ?x ≤ b
by (metis order.trans Sup-aform e Sup-aform' b-def' u-def)
hence ?x ∈ {?x .. b}
by simp

have - inverse (b * b) ≤ alpha
by (auto simp add: alpha-def inverse-mult-distrib[symmetric] inverse-eq-divide
sq-def
intro!: order-trans[OF real-divl] divide-left-mono truncate-up mult-pos-pos ⟨0
< b⟩)

{
note ⟨0 < a⟩
moreover
have ?x ∈ {a .. b} using ⟨a ≤ ?xx ≤ b⟩ by simp
moreover
note ⟨- inverse (b * b) ≤ alpha⟩
ultimately have inverse ?x ≤ inverse a + alpha * (?x - a)
by (rule inverse-linear-upper)
also have ... = alpha * ?x + (inverse a - alpha * a)
by (simp add: algebra-simps)
also have inverse a - (alpha * a) ≤ (real-divr p 1 a - alpha * a)
by (auto simp: inverse-eq-divide real-divr)
also have ... ≤ (truncate-down p (real-divl p 1 b - alpha * b) +
(real-divr p 1 a - alpha * a)) / 2 +
(truncate-up p (real-divr p 1 a - alpha * a) -
truncate-down p (real-divl p 1 b - alpha * b)) / 2
(is - ≤ (truncate-down p ?lb + ?ra) / 2 + (truncate-up p ?ra - truncate-down
p ?lb) / 2)
by (auto simp add: field-simps
intro!: order-trans[OF - add-left-mono[OF mult-left-mono[OF truncate-up]]])

```

```

also have (truncate-down p ?lb + ?ra) / 2 ≤
  truncate-up p ((truncate-down p ?lb + truncate-up p ?ra) / 2)
by (intro truncate-up-le divide-right-mono add-left-mono truncate-up) auto
also
have (truncate-up p ?ra - truncate-down p ?lb) / 2 + truncate-down p ?lb ≤
  (truncate-up p ((truncate-down p ?lb + truncate-up p ?ra) / 2))
by (rule truncate-up-le) (simp add: field-simps)
hence (truncate-up p ?ra - truncate-down p ?lb) / 2 ≤ truncate-up p
  (truncate-up p ((truncate-down p ?lb + truncate-up p ?ra) / 2) - trun-
  cate-down p ?lb)
by (intro truncate-up-le) (simp add: field-simps)
finally have inverse ?x ≤ alpha * ?x + zeta + delta
by (auto simp: zeta-def delta-def d-min'-def d-max'-def right-diff-distrib
  ac-simps)
} note upper = this

{
have alpha * b + truncate-down p (real-divl p 1 b - alpha * b) ≤ inverse b
by (rule order-trans[OF add-left-mono[OF truncate-down]])
  (auto simp: inverse-eq-divide real-divl)
hence zeta + alpha * b ≤ delta + inverse b
by (auto simp: zeta-def delta-def d-min'-def d-max'-def right-diff-distrib
  intro!: order-trans[OF - add-right-mono[OF truncate-up]])
hence alpha * ?x + zeta - delta ≤ inverse b + alpha * (?x - b)
by (simp add: algebra-simps)
also
{
note ⟨0 < aform-val e X⟩
moreover
note ⟨aform-val e X ∈ {aform-val e X .. b}⟩
moreover

note ⟨- inverse (b * b) ≤ alpha⟩
ultimately
have inverse b + alpha * (aform-val e X - b) ≤ inverse (aform-val e X)
by (rule inverse-linear-lower)
}
finally have alpha * (aform-val e X) + zeta - delta ≤ inverse (aform-val e
  X) .
} note lower = this

have inverse (aform-val e X) = alpha * (aform-val e X) + zeta +
  (inverse (aform-val e X) - alpha * (aform-val e X) - zeta)
(is - = - + ?linerr)
by simp
also
have ?linerr ∈ {- delta .. delta}
using lower upper by simp

```



```

hence linerr-le: abs ?linerr ≤ delta
  by auto

let ?z0 = trunc-bound-eucl p (alpha * fst X)
from trunc-bound-euclE
obtain e1 where abs-e1:  $|e1| \leq \text{snd } ?z0$  and e1:  $\text{fst } ?z0 = \text{alpha} * \text{fst } X + e1$ 
  by blast
let ?z1 = trunc-bound-eucl p (fst ?z0 + zeta)
from trunc-bound-euclE
obtain e1' where abs-e1':  $|e1'| \leq \text{snd } ?z1$  and e1':  $\text{fst } ?z1 = \text{fst } ?z0 + \text{zeta} + e1'$ 
  by blast

let ?zs = trunc-bound-pdevs p (scaleR-pdevs alpha (snd X))
from trunc-bound-pdevsE[OF e]
obtain e2 where abs-e2:  $|e2| \leq \text{snd } (?zs)$ 
  and e2:  $\text{pdevs-val } e (\text{fst } ?zs) = \text{pdevs-val } e (\text{scaleR-pdevs alpha (snd X)}) + e2$ 
  by blast

have  $\text{alpha} * (\text{aform-val } e X) + \text{zeta} =$ 
   $\text{aform-val } e (\text{fst } (\text{inverse-aform}' p X)) + (- e1 - e1' - e2)$ 
  unfolding inverse-aform'-def Let-def vars[symmetric]
  using  $\langle 0 < l \rangle$ 
  by (simp add: aform-val-def assms e1') (simp add: e1 e2 algebra-simps)
also
let ?err =  $(- e1 - e1' - e2 + \text{inverse } (\text{aform-val } e X) - \text{alpha} * \text{aform-val } e X - \text{zeta})$ 
  {
    have  $\text{abs } ?err \leq \text{abs } ?linerr + \text{abs } e1 + \text{abs } e1' + \text{abs } e2$ 
      by simp
    also have  $\dots \leq \text{delta} + \text{snd } ?z0 + \text{snd } ?z1 + \text{snd } ?zs$ 
      by (blast intro: add-mono linerr-le abs-e1 abs-e1' abs-e2)
    also have  $\dots \leq (\text{snd } (\text{inverse-aform}' p X))$ 
      unfolding inverse-aform'-def Let-def vars[symmetric]
      using  $\langle 0 < l \rangle$ 
      by (auto simp add: inverse-aform'-def pdevs-apply-trunc-pdevs assms vars[symmetric])
      (intro!: order.trans[OF - sum-list'-sum-list-le])
    finally have  $\text{abs } ?err \leq \text{snd } (\text{inverse-aform}' p X)$  by simp
  }
note err-le = this
have  $\text{aform-val } (e) (\text{fst } (\text{inverse-aform}' p X)) + (- e1 - e1' - e2) +$ 
   $(\text{inverse } (\text{aform-val } e X) - \text{alpha} * \text{aform-val } e X - \text{zeta}) =$ 
   $\text{aform-val } e (\text{fst } (\text{inverse-aform}' p X)) + ?err$ 
  by simp
finally
show ?thesis
  apply (intro aform-errI)
  using err-le
  by (auto simp: assms)
qed

```

**definition** *inverse-aform*  $p$   $a$  =  
do {  
  let  $l = \text{Inf-aform}' p a$ ;  
  let  $u = \text{Sup-aform}' p a$ ;  
  if  $(l \leq 0 \wedge 0 \leq u)$  then None  
  else if  $(l \leq 0)$  then (Some (apfst uminus-aform (inverse-aform'  $p$  (uminus-aform  
 $a$ ))))  
  else Some (inverse-aform'  $p$   $a$ )  
}

**lemma** *eucl-truncate-up-eq-eucl-truncate-down*:  
*eucl-truncate-up*  $p$   $x = -$  (*eucl-truncate-down*  $p$  ( $- x$ ))  
**by** (*auto simp: eucl-truncate-up-def eucl-truncate-down-def truncate-up-eq-truncate-down  
sum-negf*)

**lemma** *inverse-aformE*:  
**fixes**  $X::\text{real aform}$   
**assumes**  $e: e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
  **and** *disj*:  $\text{Inf-aform}' p X > 0 \vee \text{Sup-aform}' p X < 0$   
**obtains**  $Y$  **where**  
  *inverse-aform*  $p X = \text{Some } Y$   
  *inverse* (*aform-val*  $e X$ )  $\in$  *aform-err*  $e Y$

**proof** –  
{  
  **assume** *neg*:  $\text{Sup-aform}' p X < 0$   
  **from** *neg* **have** [*simp*]:  $\text{Inf-aform}' p X \leq 0$   
  **by** (*metis Inf-aform'-le-Sup-aform' dual-order.strict-trans1 less-asym not-less*)  
  **from** *neg disj* **have**  $0 < \text{Inf-aform}' p (\text{uminus-aform } X)$   
  **by** (*auto simp: Inf-aform'-def Sup-aform'-def eucl-truncate-up-eq-eucl-truncate-down  
ac-simps*)  
  **from** *inverse-aform'E[OF e(1) this]*  
  **have** *iin*:  $\text{inverse} (\text{aform-val } e (\text{uminus-aform } X)) \in \text{aform-err } e (\text{inverse-aform}'$   
 $p (\text{uminus-aform } X))$   
  **by** *simp*  
  **let**  $?Y = \text{apfst uminus-aform} (\text{inverse-aform}' p (\text{uminus-aform } X))$   
  **have**  $\text{inverse-aform } p X = \text{Some } ?Y$   
  **by**  $\text{inverse} (\text{aform-val } e X) \in \text{aform-err } e ?Y$   
  **using** *neg iin* **by** (*auto simp: inverse-aform-def aform-err-def*)  
  **then** **have** *?thesis ..*  
} **moreover** {  
  **assume** *pos*:  $\text{Inf-aform}' p X > 0$   
  **from** *pos* **have** *eq*:  $\text{inverse-aform } p X = \text{Some} (\text{inverse-aform}' p X)$   
  **by** (*auto simp: inverse-aform-def*)  
  **moreover**  
  **from** *inverse-aform'E[OF e(1) pos refl]*  
  **have**  $\text{inverse} (\text{aform-val } e X) \in \text{aform-err } e (\text{inverse-aform}' p X)$  .  
  **ultimately** **have** *?thesis ..*  
} **ultimately** **show** *?thesis*

using *assms* by *auto*  
 qed

**definition** *aform-err-to-aform*::*aform-err*  $\Rightarrow$  *nat*  $\Rightarrow$  *real aform*

where *aform-err-to-aform* *X n* = (*fst* (*fst X*), *pdev-upd* (*snd* (*fst X*)) *n* (*snd X*))

**lemma** *aform-err-to-aformE*:

assumes *x*  $\in$  *aform-err e X*

assumes *deg*: *degree-aform-err X*  $\leq$  *n*

obtains *err* where *x* = *aform-val* (*e(n:=err)*) (*aform-err-to-aform X n*)  
 $-1 \leq$  *err* *err*  $\leq$  *1*

**proof** –

from *aform-errE*[*OF assms(1)*] have  $|x - \text{aform-val } e \text{ (fst X)}| \leq \text{snd X}$  by *auto*

from *error-absE*[*OF this*] obtain *err* where *err*:

$x - \text{aform-val } e \text{ (fst X)} = \text{err} * \text{snd X}$  *err*  $\in$   $\{-1..1\}$

by *auto*

have *x* = *aform-val* (*e(n:=err)*) (*aform-err-to-aform X n*)

$-1 \leq$  *err* *err*  $\leq$  *1*

using *err deg*

by (*auto simp: aform-val-def aform-err-to-aform-def*)

then show *?thesis* ..

qed

**definition** *aform-to-aform-err*::*real aform*  $\Rightarrow$  *nat*  $\Rightarrow$  *aform-err*

where *aform-to-aform-err X n* = (*fst X*, *pdev-upd* (*snd X*) *n* *0*), *abs* (*pdevs-apply* (*snd X*) *n*)

**lemma** *aform-to-aform-err*: *aform-val e X*  $\in$  *aform-err e* (*aform-to-aform-err X n*)

if *e*  $\in$  *UNIV*  $\rightarrow$   $\{-1 .. 1\}$

**proof** –

from *that* have *abs-e[simp]*:  $\bigwedge i. |e \ i| \leq 1$  by (*auto simp: abs-real-def*)

have  $- e \ n * \text{pdevs-apply (snd X) n} \leq |\text{pdevs-apply (snd X) n}|$

**proof** –

have  $- e \ n * \text{pdevs-apply (snd X) n} \leq | - e \ n * \text{pdevs-apply (snd X) n} |$

by *auto*

also have  $\dots \leq \text{abs (pdevs-apply (snd X) n)}$

using *that*

by (*auto simp: abs-mult intro!: mult-left-le-one-le*)

finally show *?thesis* .

qed

moreover have *e n* \* *pdevs-apply* (*snd X*) *n*  $\leq$   $|\text{pdevs-apply (snd X) n}|$

**proof** –

have *e n* \* *pdevs-apply* (*snd X*) *n*  $\leq$   $|e \ n * \text{pdevs-apply (snd X) n}|$

by *auto*

also have  $\dots \leq \text{abs (pdevs-apply (snd X) n)}$

using *that*

by (*auto simp: abs-mult intro!: mult-left-le-one-le*)

**finally show** *?thesis* .  
**qed**  
**ultimately**  
**show** *?thesis*  
**by** (*auto simp: aform-to-aform-err-def aform-err-def aform-val-def*)  
**qed**

**definition** *acc-err p x e*  $\equiv$  (*fst x*, *truncate-up p (snd x + e)*)

**definition** *ivl-err* :: *real interval*  $\Rightarrow$  (*real*  $\times$  *real pdevs*)  $\times$  *real*  
**where** *ivl-err ivl*  $\equiv$  (((*upper ivl + lower ivl*)/2, *zero-pdevs::real pdevs*), (*upper ivl - lower ivl* / 2))

**lemma** *inverse-aform*:  
**fixes** *X::real aform*  
**assumes** *e*: *e*  $\in$  *UNIV*  $\rightarrow$   $\{-1 .. 1\}$   
**assumes** *inverse-aform p X = Some Y*  
**shows** *inverse (aform-val e X)  $\in$  aform-err e Y*  
**proof** –  
**from** *assms* **have** *Inf-aform' p X > 0  $\vee$  0 > Sup-aform' p X*  
**by** (*auto simp: inverse-aform-def Let-def bind-eq-Some-conv split: if-splits*)  
**from** *inverse-aformE[OF e this]* **obtain** *Y* **where**  
*inverse-aform p X = Some Y* *inverse (aform-val e X)  $\in$  aform-err e Y*  
**by** *auto*  
**with** *assms* **show** *?thesis* **by** *auto*  
**qed**

**lemma** *aform-err-acc-err-leI*:  
*fx  $\in$  aform-err e (acc-err p X err)*  
**if** *aform-val e (fst X) - (snd X + err)  $\leq$  fx* *fx  $\leq$  aform-val e (fst X) + (snd X + err)*  
**using** *truncate-up[of (snd X + err) p] truncate-down[of p (snd X + err)]* **that**  
**by** (*auto simp: aform-err-def acc-err-def*)

**lemma** *aform-err-acc-errI*:  
*fx  $\in$  aform-err e (acc-err p X err)*  
**if** *fx  $\in$  aform-err e (fst X, snd X + err)*  
**using** *truncate-up[of (snd X + err) p] truncate-down[of p (snd X + err)]* **that**  
**by** (*auto simp: aform-err-def acc-err-def*)

**lemma** *minus-times-le-abs*:  $- (err * B) \leq |B|$  **if**  $-1 \leq err$   $err \leq 1$  **for** *err::real*  
**proof** –  
**have** [*simp*]: *abs err  $\leq$  1* **using** *that* **by** *auto*  
**have**  $- (err * B) \leq abs (- err * B)$  **by** *auto*  
**also** **have**  $\dots \leq abs B$   
**by** (*auto simp: abs-mult intro!: mult-left-le-one-le*)  
**finally** **show** *?thesis* **by** *simp*  
**qed**

**lemma** *times-le-abs*:  $err * B \leq |B|$  **if**  $-1 \leq err$   $err \leq 1$  **for**  $err::real$   
**proof** –  
**have** [*simp*]:  $abs\ err \leq 1$  **using** *that* **by** *auto*  
**have**  $err * B \leq abs\ (err * B)$  **by** *auto*  
**also have**  $\dots \leq abs\ B$   
**by** (*auto simp: abs-mult intro!: mult-left-le-one-le*)  
**finally show** *?thesis* **by** *simp*  
**qed**

**lemma** *aform-err-lemma1*:  $-1 \leq err \implies err \leq 1 \implies$   
 $X1 + (A - e\ d * B + err * B) - e1 \leq x \implies$   
 $X1 + (A - e\ d * B) - truncate-up\ p\ (|B| + e1) \leq x$   
**apply** (*rule order-trans*)  
**apply** (*rule diff-mono*)  
**apply** (*rule order-refl*)  
**apply** (*rule truncate-up-le*[**where**  $x=e1 - err * B$ ])  
**by** (*auto simp: minus-times-le-abs*)

**lemma** *aform-err-lemma2*:  $-1 \leq err \implies err \leq 1 \implies$   
 $x \leq X1 + (A - e\ d * B + err * B) + e1 \implies$   
 $x \leq X1 + (A - e\ d * B) + truncate-up\ p\ (|B| + e1)$   
**apply** (*rule order-trans*[*rotated*])  
**apply** (*rule add-mono*)  
**apply** (*rule order-refl*)  
**apply** (*rule truncate-up-le*[**where**  $x=e1 + err * B$ ])  
**by** (*auto simp: times-le-abs*)

**lemma** *aform-err-acc-err-aform-to-aform-errI*:  
 $x \in aform-err\ e\ (acc-err\ p\ (aform-to-aform-err\ X1\ d)\ e1)$   
**if**  $-1 \leq err$   $err \leq 1$   $x \in aform-err\ (e(d := err))\ (X1, e1)$   
**using** *that*  
**by** (*auto simp: acc-err-def aform-err-def aform-val-def aform-to-aform-err-def*  
*aform-err-to-aform-def aform-err-lemma1 aform-err-lemma2*)

**definition** *map-aform-err*  $I\ p\ X =$   
 (*do* {  
*let*  $X0 = aform-err-to-aform\ X\ (degree-aform-err\ X)$ ;  
 $(X1, e1) \leftarrow I\ X0$ ;  
*Some* ( $acc-err\ p\ (aform-to-aform-err\ X1\ (degree-aform-err\ X))\ e1$ )  
 })

**lemma** *map-aform-err*:  
 $i\ x \in aform-err\ e\ Y$   
**if**  $I: \bigwedge e\ X\ Y. e \in UNIV \rightarrow \{-1 .. 1\} \implies I\ X = Some\ Y \implies i\ (aform-val\ e\ X) \in aform-err\ e\ Y$   
**and**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**and**  $Y: map-aform-err\ I\ p\ X = Some\ Y$   
**and**  $x: x \in aform-err\ e\ X$   
**proof** –

**obtain**  $X1\ e1$  **where**  
 $X1: (I\ (aform\text{-}err\text{-}to\text{-}aform\ X\ (degree\text{-}aform\text{-}err\ X))) = Some\ (X1,\ e1)$   
**and**  $Y: Y = acc\text{-}err\ p\ (aform\text{-}to\text{-}aform\text{-}err\ X1\ (degree\text{-}aform\ (fst\ X)))\ e1$   
**using**  $Y$  **by**  $(auto\ simp: map\text{-}aform\text{-}err\text{-}def\ bind\text{-}eq\text{-}Some\ conv\ Let\text{-}def)$   
**from**  $aform\text{-}err\text{-}to\text{-}aformE[OF\ x]$  **obtain**  $err$  **where**  
 $err: x = aform\text{-}val\ (e\ (degree\text{-}aform\text{-}err\ X\ :=\ err))\ (aform\text{-}err\text{-}to\text{-}aform\ X\ (degree\text{-}aform\text{-}err\ X))$   
 $(is\ =\ aform\text{-}val\ ?e\ -)$   
**and**  $-1 \leq err\ err \leq 1$   
**by**  $auto$   
**then have**  $e': ?e \in UNIV \rightarrow \{-1 .. 1\}$  **using**  $e$  **by**  $auto$   
**from**  $err$  **have**  $i\ x =$   
 $i\ (aform\text{-}val\ (e\ (degree\text{-}aform\text{-}err\ X\ :=\ err))\ (aform\text{-}err\text{-}to\text{-}aform\ X\ (degree\text{-}aform\text{-}err\ X)))$   
**by**  $simp$   
**also note**  $I[OF\ e'\ X1]$   
**also have**  $aform\text{-}err\ (e\ (degree\text{-}aform\text{-}err\ X\ :=\ err))\ (X1,\ e1) \subseteq aform\text{-}err\ e\ Y$   
**apply**  $rule$   
**unfolding**  $Y$  **using**  $\langle -1 \leq err \rangle\ \langle err \leq 1 \rangle$   
**by**  $(rule\ aform\text{-}err\text{-}acc\text{-}err\text{-}aform\text{-}to\text{-}aform\text{-}errI)$   
**finally show**  $?thesis$  .  
**qed**

**definition**  $inverse\text{-}aform\text{-}err\ p\ X = map\text{-}aform\text{-}err\ (inverse\text{-}aform\ p)\ p\ X$

**lemma**  $inverse\text{-}aform\text{-}err$ :  
 $inverse\ x \in aform\text{-}err\ e\ Y$   
**if**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**and**  $Y: inverse\text{-}aform\text{-}err\ p\ X = Some\ Y$   
**and**  $x: x \in aform\text{-}err\ e\ X$   
**using**  $map\text{-}aform\text{-}err[OF\ inverse\text{-}aform[where\ p=p]]\ e\ Y[unfolded\ inverse\text{-}aform\text{-}err\text{-}def\ x]$   
**by**  $auto$

### 5.3 Reduction (Summarization of Coefficients)

**definition**  $pdevs\text{-}of\text{-}centered\text{-}ivl\ r = (inner\text{-}scaleR\text{-}pdevs\ r\ One\text{-}pdevs)$

**lemma**  $pdevs\text{-}of\text{-}centered\text{-}ivl\text{-}eq\text{-}pdevs\text{-}of\text{-}ivl[simp]$ :  $pdevs\text{-}of\text{-}centered\text{-}ivl\ r = pdevs\text{-}of\text{-}ivl\ (-r)\ r$   
**by**  $(auto\ simp: pdevs\text{-}of\text{-}centered\text{-}ivl\text{-}def\ pdevs\text{-}of\text{-}ivl\text{-}def\ algebra\text{-}simps\ intro!:\ pdevs\text{-}eqI)$

**lemma**  $filter\text{-}pdevs\text{-}raw\text{-}nonzeros$ :  $\{i.\ filter\text{-}pdevs\text{-}raw\ s\ f\ i \neq 0\} = \{i.\ f\ i \neq 0\} \cap \{x.\ s\ x\ (f\ x)\}$   
**by**  $(auto\ simp: filter\text{-}pdevs\text{-}raw\text{-}def)$

**definition**  $summarize\text{-}pdevs$ ::  
 $nat \Rightarrow (nat \Rightarrow 'a \Rightarrow bool) \Rightarrow nat \Rightarrow 'a::executable\text{-}euclidean\text{-}space\ pdevs \Rightarrow 'a\ pdevs$

**where** *summarize-pdevs*  $p I d x =$   
 (let  $t = tdev' p (filter-pdevs (-I) x)$   
 in *msum-pdevs*  $d (filter-pdevs I x) (pdevs-of-centered-ivl t)$ )

**definition** *summarize-threshold*

**where** *summarize-threshold*  $p t x y \longleftrightarrow infnorm y \geq t * infnorm (eucl-truncate-up p (tdev' p x))$

**lemma** *error-abs-euclE*:

**fixes**  $err::'a::ordered-euclidean-space$

**assumes**  $abs\ err \leq k$

**obtains**  $e::'a \Rightarrow real$  **where**  $err = (\sum i \in Basis. (e\ i * (k \cdot i)) *_R i) \wedge e \in UNIV \rightarrow \{-1 .. 1\}$

**proof** *atomize-elim*

{  
**fix**  $i::'a$   
**assume**  $i \in Basis$   
**hence**  $abs (err \cdot i) \leq (k \cdot i)$  **using** *assms* **by** (*auto simp add: eucl-le*[**where**  $'a='a$ ] *abs-inner*)  
**hence**  $\exists e. (err \cdot i = e * (k \cdot i)) \wedge e \in \{-1..1\}$   
**by** (*rule error-absE*) *auto*  
 }  
**then obtain**  $e$  **where**  $e$ :  
 $\bigwedge i. i \in Basis \implies err \cdot i = e\ i * (k \cdot i)$   
 $\bigwedge i. i \in Basis \implies e\ i \in \{-1 .. 1\}$   
**by** *metis*  
**have** *singleton*:  $\bigwedge b. b \in Basis \implies (\sum i \in Basis. e\ i * (k \cdot i) * (if\ i = b\ then\ 1\ else\ 0)) =$   
 $(\sum i \in \{b\}. e\ i * (k \cdot i) * (if\ i = b\ then\ 1\ else\ 0))$   
**by** (*rule sum.mono-neutral-cong-right*) *auto*  
**show**  $\exists e::'a \Rightarrow real. err = (\sum i \in Basis. (e\ i * (k \cdot i)) *_R i) \wedge (e \in UNIV \rightarrow \{-1..1\})$   
**using**  $e$   
**by** (*auto intro!*: *exI*[**where**  $x=\lambda i. if\ i \in Basis\ then\ e\ i\ else\ 0$ ] *euclidean-eqI*[**where**  $'a='a$ ]  
*simp: inner-sum-left inner-Basis singleton*)

**qed**

**lemma** *summarize-pdevsE*:

**fixes**  $x::'a::executable-euclidean-space\ pdevs$

**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$

**assumes**  $d: degree\ x \leq d$

**obtains**  $e'$  **where**  $pdevs-val\ e\ x = pdevs-val\ e' (summarize-pdevs\ p\ I\ d\ x)$

$\bigwedge i. i < d \implies e\ i = e'\ i$

$e' \in UNIV \rightarrow \{-1 .. 1\}$

**proof** *atomize-elim*

**have**  $pdevs-val\ e\ x = (\sum i < degree\ x. e\ i *_R pdevs-apply\ x\ i)$

**by** (*auto simp add: pdevs-val-sum intro!*: *sum.cong*)

**also have**  $\dots = (\sum i \in \{..<degree\ x\} \cap \{i. I\ i\ (pdevs-apply\ x\ i)\}. e\ i *_R$

$pdevs\text{-}apply\ x\ i) +$   
 $(\sum i \in \{..<degree\ x\} - \{i.\ I\ i\ (pdevs\text{-}apply\ x\ i)\}.\ e\ i\ *_R\ pdevs\text{-}apply\ x\ i)$   
 $(is\ - = ?large + ?small)$   
**by**  $(subst\ sum.\ union\text{-}disjoint[symmetric])\ (auto\ simp:\ ac\text{-}simps\ intro!\: sum.\ cong)$   
**also have**  $?large = pdevs\text{-}val\ e\ (filter\text{-}pdevs\ I\ x)$   
**by**  $(simp\ add:\ pdevs\text{-}val\ filter\text{-}pdevs)$   
**also have**  $?small = pdevs\text{-}val\ e\ (filter\text{-}pdevs\ (-I)\ x)$   
**by**  $(simp\ add:\ pdevs\text{-}val\ filter\text{-}pdevs\ Collect\text{-}neg\text{-}eq\ Diff\text{-}eq)$   
**also**  
**have**  $abs\ \dots \leq tdev'\ p\ (filter\text{-}pdevs\ (-I)\ x)\ (is\ abs\ ?r \leq ?t)$   
**using**  $e\ \mathbf{by}\ (rule\ abs\text{-}pdevs\text{-}val\ le\text{-}tdev')$   
**hence**  $?r \in \{-?t .. ?t\}$   
**by**  $(metis\ abs\text{-}le\text{-}D1\ abs\text{-}le\text{-}D2\ atLeastAtMost\text{-}iff\ minus\text{-}le\text{-}iff)$   
**from**  $in\text{-}ivl\text{-}affine\text{-}of\text{-}ivlE[OF\ this]\ \mathbf{obtain}\ e2$   
**where**  $?r = aform\text{-}val\ e2\ (aform\text{-}of\text{-}ivl\ (-\ ?t)\ ?t)$   
**and**  $e2: e2 \in UNIV \rightarrow \{-1..1\}$   
**by**  $metis$   
**note**  $this(1)$   
**also**  
**define**  $e'$  **where**  $e'\ i = (if\ i < d\ then\ e\ i\ else\ e2\ (i - d))\ \mathbf{for}\ i$   
**hence**  $aform\text{-}val\ e2\ (aform\text{-}of\text{-}ivl\ (-\ ?t)\ ?t) =$   
 $pdevs\text{-}val\ (\lambda i.\ e'\ (i + d))\ (pdevs\text{-}of\text{-}ivl\ (-\ ?t)\ ?t)$   
**by**  $(auto\ simp:\ aform\text{-}of\text{-}ivl\text{-}def\ aform\text{-}val\text{-}def)$   
**also**  
**have**  $pdevs\text{-}val\ e\ (filter\text{-}pdevs\ I\ x) = pdevs\text{-}val\ e'\ (filter\text{-}pdevs\ I\ x)$   
**using**  $assms\ \mathbf{by}\ (auto\ simp:\ e'\text{-}def\ pdevs\text{-}val\text{-}sum\ intro!\: sum.\ cong)$   
**finally have**  $pdevs\text{-}val\ e\ x =$   
 $pdevs\text{-}val\ e'\ (filter\text{-}pdevs\ I\ x) + pdevs\text{-}val\ (\lambda i.\ e'\ (i + d))\ (pdevs\text{-}of\text{-}ivl\ (-\ ?t)$   
 $?t)$   
 $.$   
**also note**  $pdevs\text{-}val\ msum\text{-}pdevs[symmetric,\ OF\ order\text{-}trans[OF\ degree\text{-}filter\text{-}pdevs\text{-}le\ d]]$   
**finally have**  $pdevs\text{-}val\ e\ x = pdevs\text{-}val\ e'\ (summarize\text{-}pdevs\ p\ I\ d\ x)$   
**by**  $(auto\ simp:\ summarize\text{-}pdevs\text{-}def\ Let\text{-}def)$   
**moreover have**  $e' \in UNIV \rightarrow \{-1 .. 1\}$  **using**  $e\ e2\ \mathbf{by}\ (auto\ simp:\ e'\text{-}def\ Pi\text{-}iff)$   
**moreover have**  $\forall i < d.\ e'\ i = e\ i$   
**by**  $(auto\ simp:\ e'\text{-}def)$   
**ultimately show**  $\exists e'. pdevs\text{-}val\ e\ x = pdevs\text{-}val\ e'\ (summarize\text{-}pdevs\ p\ I\ d\ x) \wedge$   
 $(\forall i < d.\ e\ i = e'\ i) \wedge e' \in UNIV \rightarrow \{-1..1\}$   
**by**  $auto$   
**qed**

**definition**  $summarize\text{-}pdevs\text{-}list\ p\ I\ d\ xs =$   
 $map\ (\lambda(d,\ x).\ summarize\text{-}pdevs\ p\ (\lambda i\ -. I\ i\ (pdevs\text{-}applies\ xs\ i))\ d\ x)\ (zip\ [d..<d\ +\ length\ xs]\ xs)$

**lemma**  $filter\text{-}pdevs\text{-}cong[cong]:$   
**assumes**  $x = y$



**assumes**  $\bigwedge i. i \in \text{pdevs-domain } y \implies P i (\text{pdevs-apply } x i) = Q i (\text{pdevs-apply } y i)$   
**shows**  $\text{filter-pdevs } P x = \text{filter-pdevs } Q y$   
**using** *assms*  
**by** (*force intro!*: *pdevs-eqI*)

**lemma** *summarize-pdevs-cong*[*cong*]:  
**assumes**  $p = q \ a = c \ b = d$   
**assumes**  $PQ: \bigwedge i. i \in \text{pdevs-domain } d \implies P i (\text{pdevs-apply } b i) = Q i (\text{pdevs-apply } d i)$   
**shows**  $\text{summarize-pdevs } p P a b = \text{summarize-pdevs } q Q c d$   
**proof** –  
**have**  $(\text{filter-pdevs } P b) = \text{filter-pdevs } Q d$   
 $(\text{filter-pdevs } (\lambda a b. \neg P a b) b) = \text{filter-pdevs } (\lambda a b. \neg Q a b) d$   
**using** *assms*  
**by** (*auto intro!*: *filter-pdevs-cong*)  
**then show** *?thesis* **by** (*auto simp add: assms summarize-pdevs-def Let-def*)  
**qed**

**lemma** *lookup-eq-None-iff*:  $(\text{Mapping.lookup } M x = \text{None}) = (x \notin \text{Mapping.keys } M)$   
**by** (*transfer*) *auto*

**lemma** *lookup-eq-SomeD*:  
 $(\text{Mapping.lookup } M x = \text{Some } y) \implies (x \in \text{Mapping.keys } M)$   
**by** *transfer auto*

**definition** *domain-pdevs*  $xs = (\bigcup (\text{pdevs-domain } `(\text{set } xs)))$

**definition** *pdevs-mapping*  $xs =$   
*(let*  
 $D = \text{sorted-list-of-set } (\text{domain-pdevs } xs);$   
 $M = \text{Mapping.tabulate } D (\text{pdevs-applys } xs);$   
 $\text{zeroes} = \text{replicate } (\text{length } xs) 0$   
*in*  $\text{Mapping.lookup-default } \text{zeroes } M)$

**lemma** *pdevs-mapping-eq*[*simp*]:  $\text{pdevs-mapping } xs = \text{pdevs-applys } xs$   
**unfolding** *pdevs-mapping-def pdevs-applys-def*  
**apply** (*auto simp: Mapping.lookup-default-def lookup-eq-None-iff domain-pdevs-def split: option.splits intro!: ext*)  
**subgoal** **by** (*auto intro!: nth-equalityI*)  
**subgoal** **apply** (*auto intro!: nth-equalityI dest:* )  
**subgoal**  
**apply** (*frule lookup-eq-SomeD*)  
**apply** *auto*  
**by** (*metis distinct-sorted-list-of-set keys-tabulate length-map lookup-eq-SomeD lookup-tabulate option.inject*)  
**subgoal**  
**apply** (*frule lookup-eq-SomeD*)

```

apply (auto simp: map-nth)
by (metis (mono-tags, lifting) keys-tabulate
      lookup-eq-SomeD lookup-tabulate option.inject distinct-sorted-list-of-set)
done
done

```

```

lemma compute-summarize-pdevs-list[code]:
  summarize-pdevs-list p I d xs =
    (let M = pdevs-mapping xs
     in map ( $\lambda(x, y). \text{summarize-pdevs } p (\lambda i -. I i (M i)) x y$ ) (zip [d.. $d + \text{length } xs$ ] xs))
unfolding summarize-pdevs-list-def pdevs-mapping-eq
by auto

```

```

lemma
  in-centered-ivlE:
fixes r t::real
assumes r  $\in \{-t .. t\}$ 
obtains e where e  $\in \{-1 .. 1\}$  r = e * t
using assms
by (atomize-elim) (auto intro!: exI[where x=r / t] simp: divide-simps)

```

```

lift-definition singleton-pdevs::'a  $\Rightarrow$  'a::real-normed-vector pdevs is  $\lambda x i. \text{if } i = 0 \text{ then } x \text{ else } 0$ 
by auto
lemmas [simp] = singleton-pdevs.rep-eq

```

```

lemma singleton-0[simp]: singleton-pdevs 0 = zero-pdevs
by (auto intro!: pdevs-eqI)

```

```

lemma degree-singleton-pdevs[simp]: degree (singleton-pdevs x) = (if x = 0 then 0 else Suc 0)
by (auto simp: intro!: degree-eqI)

```

```

lemma pdevs-val-singleton-pdevs[simp]: pdevs-val e (singleton-pdevs x) = e 0 *R x
by (auto simp: pdevs-val-sum if-distrib sum.delta cong: if-cong)

```

```

lemma pdevs-of-ivl-real:
fixes a b::real
shows pdevs-of-ivl a b = singleton-pdevs ((b - a) / 2)
by (auto simp: pdevs-of-ivl-def Basis-list-real-def intro!: pdevs-eqI)

```

```

lemma summarize-pdevs-listE:
fixes X::real pdevs list
assumes e: e  $\in UNIV \rightarrow \{-1 .. 1\}$ 
assumes d: degrees X  $\leq d$ 
obtains e' where pdevs-vals e X = pdevs-vals e' (summarize-pdevs-list p I d X)
   $\bigwedge i. i < d \implies e i = e' i$ 
  e'  $\in UNIV \rightarrow \{-1 .. 1\}$ 

```

```

proof –
  let ?I = {i. I i (pdevs-applys X i)}
  let ?J = λi x. I i (pdevs-applys X i)

  have pdevs-vals e X = map (λx. ∑ i<degree x. e i *R pdevs-apply x i) X
    using d
    by (auto simp: pdevs-vals-def
      simp del: real-scaleR-def
      intro!: pdevs-val-sum-le
      dest!: degrees-leD)
  also have ... = map (λx.
    (∑ i∈{..by (rule map-cong[OF refl], subst sum.union-disjoint[symmetric]) (auto intro!:
sum.cong)
  also
    have ... = map (λx. pdevs-val e (filter-pdevs ?J x) + pdevs-val e (filter-pdevs
(–?J) x)) X
      (is - = map (λx. ?large x + ?small x) -)
      by (auto simp: pdevs-val-filter-pdevs Diff-eq Compl-eq)
    also have ... = map snd (zip [d..by simp
    also have ... = map (λ(d, x). ?large x + ?small x) (zip [d..unfolding map-zip-map2
      by simp
    also have ... = map (λ(d', x). ?large x + ?small (snd (?z ! (d' – d)))) ?z
      by (auto simp: in-set-zip)
    also
      let ?t = λx. tdev' p (filter-pdevs (–?J) x)
      let ?x = λd'. snd (?z ! (d' – d))
      {
        fix d' assume d ≤ d' d' < d + length X
        have abs (?small (?x d')) ≤ ?t (?x d')
          using <e ∈ -> by (rule abs-pdevs-val-le-tdev')
        then have ?small (?x d') ∈ {–?t (?x d') .. ?t (?x d')}
          by auto
        from in-centered-ivlE[OF this] have ∃ e∈{–1 .. 1}. ?small (?x d') = e * ?t
(?x d') by blast
      } then obtain e'' where e'':
        e'' d' ∈ {–1 .. 1}
        ?small (?x d') = e'' d' * ?t (?x d')
        if d' ∈ {d ..< d + length X} for d'
        apply atomize-elim
        unfolding all-conj-distrib[symmetric] imp-conjR[symmetric]
        unfolding Ball-def[symmetric] atLeastAtMost-iff[symmetric]
        apply (rule bchoice)
        apply (auto simp: Bex-def)
        done
      define e' where e' ≡ λi. if i < d then e i else if i < d + length X then e'' i else

```

```

0
have e': e' d' ∈ {-1 .. 1}
  ?small (?x d') = e' d' * ?t (?x d')
if d' ∈ {d ..< d + length X} for d'
using e'' that
by (auto simp: e'-def split: if-splits)
then have *: pdevs-val e (filter-pdevs (λa b. ¬ I a (pdevs-applys X a)) (?x d'))
=
  e' d' * ?t (?x d') if d' ∈ {d ..< d + length X} for d'
using that
by auto
have map (λ(d', x). ?large x + ?small (?x d')) ?z =
  map (λ(d', x). ?large x + e' d' * ?t (?x d')) ?z
apply (auto simp: in-set-zip)
subgoal for n
  using e'(2)[of d + n]
  by auto
done
also have ... = map (λ(d', x). pdevs-val e' (summarize-pdevs p ?J d' x)) (zip
[d..<d + length X] X)
apply (auto simp: summarize-pdevs-def pdevs-val-msum-pdevs Let-def in-set-zip)
apply (subst pdevs-val-msum-pdevs)
using d
  apply (auto intro!: degree-filter-pdevs-le[THEN order-trans])
subgoal by (auto dest!: degrees-leD nth-mem)
apply (auto simp: pdevs-of-ivl-real intro!: )
subgoal premises prems
proof -
  have degree (filter-pdevs (λi x. I i (pdevs-applys X i)) (X ! n)) ≤ d if n <
length X for n
  using d that
  by (intro degree-filter-pdevs-le[THEN order-trans]) (simp add: degrees-leD)
then show ?thesis
  using prems e''
  apply (intro pdevs-val-degree-cong)
  apply (auto dest!: )
  apply (auto simp: e'-def)
  apply (meson ‹∧n. [n < length X; degrees X ≤ d] ⇒ degree (X ! n) ≤ d
+ n› degree-filter-pdevs-le less-le-trans)
  by (meson less-le-trans trans-less-add1)
qed
done
also have ... = pdevs-vals e' (summarize-pdevs-list p I d X)
by (auto simp: summarize-pdevs-list-def pdevs-vals-def)
finally have pdevs-vals e X = pdevs-vals e' (summarize-pdevs-list p I d X) .
moreover have (∧i. i < d ⇒ e i = e' i) e' ∈ UNIV → {- 1..1}
  using ‹e ∈ -› e''
  by (auto simp: e'-def)
ultimately show ?thesis ..

```

qed

**fun** *list-ex2* **where**

*list-ex2*  $P \ [] \ xs = False$   
| *list-ex2*  $P \ xs \ [] = False$   
| *list-ex2*  $P \ (x\#xs) \ (y\#ys) = (P \ x \ y \ \vee \ list-ex2 \ P \ xs \ ys)$

**lemma** *list-ex2-iff*:

*list-ex2*  $P \ xs \ ys \ \longleftrightarrow \ (\neg list-all2 \ (\neg P) \ (take \ (length \ ys) \ xs) \ (take \ (length \ xs) \ ys))$   
**by** (*induction*  $P \ xs \ ys$  *rule: list-ex2.induct*) *auto*

**definition** *summarize-aforms*  $p \ C \ d \ (X::real \ aform \ list) =$

$(zip \ (map \ fst \ X) \ (summarize-pdevs-list \ p \ (C \ X) \ d \ (map \ snd \ X)))$

**lemma** *aform-vals-pdevs-vals*:

*aform-vals*  $e \ X = map \ (\lambda(x, y). \ x + y) \ (zip \ (map \ fst \ X) \ (pdevs-vals \ e \ (map \ snd \ X)))$   
**by** (*auto simp: pdevs-vals-def aform-vals-def aform-val-def[abs-def]*)  
*map-zip-map map-zip-map2 split-beta' zip-same-conv-map*

**lemma** *summarize-aformsE*:

**fixes**  $X::real \ aform \ list$

**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$

**assumes**  $d: degree-aforms \ X \leq d$

**obtains**  $e'$  **where** *aform-vals*  $e \ X = aform-vals \ e' \ (summarize-aforms \ p \ C \ d \ X)$

$\bigwedge i. i < d \implies e \ i = e' \ i$

$e' \in UNIV \rightarrow \{-1 .. 1\}$

**proof** –

**define**  $Xs$  **where**  $Xs = map \ snd \ X$

**have** *aform-vals*  $e \ X = map \ (\lambda(x, y). \ x + y) \ (zip \ (map \ fst \ X) \ (pdevs-vals \ e \ Xs))$

**by** (*auto simp: aform-vals-pdevs-vals Xs-def*)

**also obtain**  $e'$  **where**  $e': e' \in UNIV \rightarrow \{-1 .. 1\}$

$\bigwedge i. i < d \implies e \ i = e' \ i$

*pdevs-vals*  $e \ Xs = pdevs-vals \ e' \ (summarize-pdevs-list \ p \ (C \ X) \ d \ Xs)$

**using** *summarize-pdevs-listE[OF e d, of p C X]*

**by** (*metis Xs-def*)

**note** *this(3)*

**also have**  $map \ (\lambda(x, y). \ x + y) \ (zip \ (map \ fst \ X) \ \dots) = aform-vals \ e' \ (summarize-aforms \ p \ C \ d \ X)$

**unfolding** *aform-vals-pdevs-vals*

**by** (*simp add: summarize-aforms-def Let-def Xs-def summarize-pdevs-list-def split-beta'*)

**finally have** *aform-vals*  $e \ X = aform-vals \ e' \ (summarize-aforms \ p \ C \ d \ X)$

$\bigwedge i. i < d \implies e \ i = e' \ i$

$e' \in UNIV \rightarrow \{-1 .. 1\}$

**using**  $e' \ d$

**by** (*auto simp: Xs-def*)

**then show** *?thesis ..*

qed

Different reduction strategies:

**definition** *collect-threshold*  $p$   $ta$   $t$  ( $X::real$  *aform list*) =  
 (let  
    $Xs = map\ snd\ X$ ;  
    $as = map\ (\lambda X. max\ ta\ (t * tdev'\ p\ X))\ Xs$   
 in  $(\lambda(i::nat)\ xs. list-ex2\ (\le)\ as\ (map\ abs\ xs))$ )

**definition** *collect-girard*  $p$   $m$  ( $X::real$  *aform list*) =  
 (let  
    $Xs = map\ snd\ X$ ;  
    $M = pdevs-mapping\ Xs$ ;  
    $D = domain-pdevs\ Xs$ ;  
    $N = length\ X$   
 in if  $card\ D \leq m$  then  $(\lambda- . True)$  else  
 let  
    $Ds = sorted-list-of-set\ D$ ;  
    $ortho-indices = map\ fst\ (take\ (2 * N)\ (sort-key\ (\lambda(i, r). r)\ (map\ (\lambda i. let\ xs = M\ i\ in\ (i, sum-list'\ p\ xs - fold\ max\ xs\ 0))\ Ds)))$ ;  
    $- = ()$   
 in  $(\lambda i\ (xs::real\ list). i \in set\ ortho-indices)$ )

## 5.4 Splitting with heuristics

**definition** *abs-pdevs* = *unop-pdevs abs*

**definition** *abssum-of-pdevs-list*  $X = fold\ (\lambda a\ b. (add-pdevs\ (abs-pdevs\ a)\ b))\ X\ zero-pdevs$

**definition** *split-aforms*  $xs\ i = (let\ splits = map\ (\lambda x. split-aform\ x\ i)\ xs\ in\ (map\ fst\ splits, map\ snd\ splits))$

**definition** *split-aforms-largest-uncond*  $X = (let\ (i, x) = max-pdev\ (abssum-of-pdevs-list\ (map\ snd\ X))\ in\ split-aforms\ X\ i)$

**definition** *Inf-aform-err*  $p\ Rd = (float-of\ (truncate-down\ p\ (Inf-aform'\ p\ (fst\ Rd) - abs(snd\ Rd))))$

**definition** *Sup-aform-err*  $p\ Rd = (float-of\ (truncate-up\ p\ (Sup-aform'\ p\ (fst\ Rd) + abs(snd\ Rd))))$

**context includes** *interval.lifting begin*

**lift-definition** *ivl-of-aform-err*:: $nat \Rightarrow aform-err \Rightarrow float\ interval$

**is**  $\lambda p\ Rd. (Inf-aform-err\ p\ Rd, Sup-aform-err\ p\ Rd)$

**by** (*auto simp: aform-err-def Inf-aform-err-def Sup-aform-err-def*

*intro!: truncate-down-le truncate-up-le add-increasing[OF - Inf-aform'-le-Sup-aform']*)

**lemma** *lower-ivl-of-aform-err*:  $lower\ (ivl-of-aform-err\ p\ Rd) = Inf-aform-err\ p\ Rd$

**and** *upper-ivl-of-aform-err*:  $upper\ (ivl-of-aform-err\ p\ Rd) = Sup-aform-err\ p\ Rd$

**by** (*transfer, simp*) $+$

**end**

**definition** *approx-un::nat*  
 $\Rightarrow$  (*float interval*  $\Rightarrow$  *float interval option*)  
 $\Rightarrow$  ((*real*  $\times$  *real pdevs*)  $\times$  *real option*)  
 $\Rightarrow$  ((*real*  $\times$  *real pdevs*)  $\times$  *real option*)  
**where** *approx-un p f a* = **do** {  
*rd*  $\leftarrow$  *a*;  
*ivl*  $\leftarrow$  *f (ivl-of-aform-err p rd)*;  
*Some (ivl-err (real-interval ivl))*  
}

**definition** *interval-extension1::(float interval  $\Rightarrow$  (float interval) option)  $\Rightarrow$  (real  $\Rightarrow$  real)  $\Rightarrow$  bool*  
**where** *interval-extension1 F f*  $\longleftrightarrow$  ( $\forall$  *ivl ivl'*. *F ivl* = *Some ivl'*  $\longrightarrow$  ( $\forall$  *x*. *x*  $\in_r$  *ivl*  $\longrightarrow$  *f x*  $\in_r$  *ivl'*))

**lemma** *interval-extension1D*:  
**assumes** *interval-extension1 F f*  
**assumes** *F ivl* = *Some ivl'*  
**assumes** *x*  $\in_r$  *ivl*  
**shows** *f x*  $\in_r$  *ivl'*  
**using** *assms* **by** (*auto simp: interval-extension1-def*)

**lemma** *approx-un-argE*:  
**assumes** *au: approx-un p F X* = *Some Y*  
**obtains** *X'* **where** *X* = *Some X'*  
**using** *assms*  
**by** (*auto simp: approx-un-def bind-eq-Some-conv*)

**lemma** *degree-aform-independent-from*:  
 $\text{degree-aform (independent-from } d1 \text{ } X) \leq d1 + \text{degree-aform } X$   
**by** (*auto simp: independent-from-def degree-msum-pdevs-le*)

**lemma** *degree-aform-of-ivl*:  
**fixes** *a b::'a::executable-euclidean-space*  
**shows**  $\text{degree-aform (aform-of-ivl } a \text{ } b) \leq \text{length (Basis-list::'a list)}$   
**by** (*auto simp: aform-of-ivl-def degree-pdevs-of-ivl-le*)

**lemma** *aform-err-ivl-err[simp]*: *aform-err e (ivl-err ivl')* = *set-of ivl'*  
**by** (*auto simp: aform-err-def ivl-err-def aform-val-def divide-simps set-of-eq*)

**lemma** *Inf-Sup-aform-err*:  
**fixes** *X*  
**assumes** *e*: *e*  $\in$  *UNIV*  $\rightarrow$   $\{-1 .. 1\}$   
**defines** *X'*  $\equiv$  *fst X*  
**shows** *aform-err e X*  $\subseteq$   $\{\text{Inf-aform-err } p \text{ } X .. \text{Sup-aform-err } p \text{ } X\}$   
**using** *Inf-aform[OF e, of X]* *Sup-aform[OF e, of X]* *Inf-aform'[of p X]* *Sup-aform'[of X' p]*  
**by** (*auto simp: aform-err-def X'-def Inf-aform-err-def Sup-aform-err-def intro!: truncate-down-le truncate-up-le*)

**lemma** *ivl-of-aform-err*:

**fixes**  $X$

**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$

**shows**  $x \in aform-err\ e\ X \implies x \in_r ivl-of-aform-err\ p\ X$

**using** *Inf-Sup-aform-err*[*OF e, of X p*]

**by** (*auto simp: set-of-eq lower-ivl-of-aform-err upper-ivl-of-aform-err*)

**lemma** *approx-unE*:

**assumes** *ie: interval-extension1 F f*

**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$

**assumes** *au: approx-un p F X'err = Some Ye*

**assumes**  $x: case\ X'err\ of\ None \implies True \mid Some\ X'err \implies x \in aform-err\ e\ X'err$

**shows**  $f\ x \in aform-err\ e\ Ye$

**proof** –

**from** *au* **obtain**  $ivl'\ X'\ err$

**where**  $F: F\ (ivl-of-aform-err\ p\ (X',\ err)) = Some\ (ivl')$

**and**  $Y: Ye = ivl-err\ (real-interval\ ivl')$

**and**  $X'err: X'err = Some\ (X',\ err)$

**by** (*auto simp: approx-un-def bind-eq-Some-conv*)

**from**  $x$

**have**  $x \in aform-err\ e\ (X',\ err)$  **by** (*auto simp: X'err*)

**from** *ivl-of-aform-err*[*OF e this*]

**have**  $x \in_r ivl-of-aform-err\ p\ (X',\ err)$  .

**from** *interval-extension1D*[*OF ie F this*]

**have**  $f\ x \in_r ivl'$  .

**also** **have**  $\dots = aform-err\ e\ Ye$

**unfolding**  $Y\ aform-err-ivl-err$  ..

**finally** **show** *?thesis* .

**qed**

**definition** *approx-bin p f rd sd = do* {

*ivl*  $\leftarrow f\ (ivl-of-aform-err\ p\ rd)$

*(ivl-of-aform-err p sd);*

*Some (ivl-err (real-interval ivl))*

}

**definition** *interval-extension2::(float interval  $\implies$  float interval  $\implies$  float interval option)  $\implies$  (real  $\implies$  real  $\implies$  real)  $\implies$  bool*

**where** *interval-extension2 F f  $\longleftrightarrow$  ( $\forall\ ivl1\ ivl2\ ivl.\ F\ ivl1\ ivl2 = Some\ ivl \longrightarrow$*

*( $\forall\ x\ y.\ x \in_r ivl1 \longrightarrow y \in_r ivl2 \longrightarrow f\ x\ y \in_r ivl$ ))*

**lemma** *interval-extension2D*:

**assumes** *interval-extension2 F f*

**assumes**  $F\ ivl1\ ivl2 = Some\ ivl$

**shows**  $x \in_r ivl1 \implies y \in_r ivl2 \implies f\ x\ y \in_r ivl$

**using** *assms* **by** (*auto simp: interval-extension2-def*)



**lemma** *approx-binE*:

**assumes** *ie*: *interval-extension2* *F* *f*

**assumes** *w*:  $w \in \text{aform-err } e (W', \text{err}w)$

**assumes** *x*:  $x \in \text{aform-err } e (X', \text{err}x)$

**assumes** *ab*: *approx-bin* *p* *F*  $((W', \text{err}w)) ((X', \text{err}x)) = \text{Some } Ye$

**assumes** *e*:  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$

**shows**  $f w x \in \text{aform-err } e Ye$

**proof** –

**from** *ab* **obtain** *ivl'*

**where** *F*:  $F (\text{ivl-of-aform-err } p (W', \text{err}w)) (\text{ivl-of-aform-err } p (X', \text{err}x)) = \text{Some } ivl'$

**and** *Y*:  $Ye = \text{ivl-err } (\text{real-interval } ivl')$

**by** (*auto simp: approx-bin-def bind-eq-Some-conv max-def*)

**from** *interval-extension2D*[*OF ie F*

*ivl-of-aform-err*[*OF e, where p=p, OF w*]

*ivl-of-aform-err*[*OF e, where p=p, OF x*]]

**have**  $f w x \in_r ivl'$  .

**also have**  $\dots = \text{aform-err } e Ye$  **unfolding** *Y aform-err-ivl-err* ..

**finally show** *?thesis* .

**qed**

**definition** *min-aform-err* *p* *a1*  $(a2::\text{aform-err}) =$

(*let*

*ivl1* = *ivl-of-aform-err* *p* *a1*;

*ivl2* = *ivl-of-aform-err* *p* *a2*

*in if upper ivl1 < lower ivl2 then a1*

*else if upper ivl2 < lower ivl1 then a2*

*else ivl-err (real-interval (min-interval ivl1 ivl2)))*)

**definition** *max-aform-err* *p* *a1*  $(a2::\text{aform-err}) =$

(*let*

*ivl1* = *ivl-of-aform-err* *p* *a1*;

*ivl2* = *ivl-of-aform-err* *p* *a2*

*in if upper ivl1 < lower ivl2 then a2*

*else if upper ivl2 < lower ivl1 then a1*

*else ivl-err (real-interval (max-interval ivl1 ivl2)))*)

## 5.5 Approximate Min Range - Kind Of Trigonometric Functions

**definition** *affine-unop*  $:: \text{nat} \Rightarrow \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \Rightarrow \text{aform-err} \Rightarrow \text{aform-err}$

**where**

*affine-unop* *p* *a* *b* *d* *X* = (*let*

$((x, xs), xe) = X$ ;

$(ax, axe) = \text{trunc-bound-eucl } p (a * x)$ ;

$(y, ye) = \text{trunc-bound-eucl } p (ax + b)$ ;

$(ys, yse) = \text{trunc-bound-pdevs } p (\text{scaleR-pdevs } a xs)$

*in*  $((y, ys), \text{sum-list}' p [\text{truncate-up } p (|a| * xe), axe, ye, yse, d])$ )

— TODO: also do binop

**lemma** *aform-err-leI*:  
 $y \in \text{aform-err } e (c, d)$   
**if**  $y \in \text{aform-err } e (c, d')$   $d' \leq d$   
**using that by** (*auto simp: aform-err-def*)

**lemma** *aform-err-eqI*:  
 $y \in \text{aform-err } e (c, d)$   
**if**  $y \in \text{aform-err } e (c, d')$   $d' = d$   
**using that by** (*auto simp: aform-err-def*)

**lemma** *sum-list'-append[simp]*:  $\text{sum-list}' p (ds@[d]) = \text{truncate-up } p (d + \text{sum-list}' p ds)$   
**unfolding** *sum-list'-def*  
**by** (*simp add: eucl-truncate-up-real-def*)

**lemma** *aform-err-sum-list'*:  
 $y \in \text{aform-err } e (c, \text{sum-list}' p ds)$   
**if**  $y \in \text{aform-err } e (c, \text{sum-list } ds)$   
**using that** (*1*)  
**apply** (*rule aform-err-leI*)  
**by** (*rule sum-list-le-sum-list'*)

**lemma** *aform-err-trunc-bound-eucl*:  
 $y \in \text{aform-err } e ((\text{fst } (\text{trunc-bound-eucl } p X), xs), \text{snd } (\text{trunc-bound-eucl } p X) + d)$   
**if**  $y: y \in \text{aform-err } e ((X, xs), d)$   
**using that**  
**proof** –  
**from** *aform-errE[OF y]*  
**have**  $|y - \text{aform-val } e (X, xs)| \leq d$  **by** *auto*  
**then show** *?thesis*  
**apply** (*intro aform-errI*)  
**apply** (*rule trunc-bound-euclE[of p X]*)  
**by** (*auto simp: aform-val-def*)  
**qed**

**lemma** *trunc-err-pdevsE*:  
**assumes**  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
**obtains** *err* **where**  
 $|err| \leq \text{tdev}' p (\text{trunc-err-pdevs } p xs)$   
 $\text{pdevs-val } e (\text{trunc-pdevs } p xs) = \text{pdevs-val } e xs + err$   
**using** *trunc-bound-pdevsE[of e p xs]*  
**by** (*auto simp: trunc-bound-pdevs-def assms*)

**lemma** *aform-err-trunc-bound-pdevsI*:  
 $y \in \text{aform-err } e ((c, \text{fst } (\text{trunc-bound-pdevs } p xs)), \text{snd } (\text{trunc-bound-pdevs } p xs) + d)$   
**if**  $y: y \in \text{aform-err } e ((c, xs), d)$

**and**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**using** *that*  
**proof** –  
**define** *exs* **where**  $exs = trunc\text{-}err\text{-}pdevs\ p\ xs$   
**from**  $aform\text{-}errE[OF\ y]$   
**have**  $|y - aform\text{-}val\ e\ (c, xs)| \leq d$  **by** *auto*  
**then show** *?thesis*  
**apply** (*intro aform-errI*)  
**apply** (*rule trunc-err-pdevsE[OF e, of p xs]*)  
**by** (*auto simp: aform-val-def trunc-bound-pdevs-def*)  
**qed**

**lemma** *aform-err-addI*:  
 $y \in aform\text{-}err\ e\ ((a + b, xs), d)$   
**if**  $y - b \in aform\text{-}err\ e\ ((a, xs), d)$   
**using** *that*  
**by** (*auto simp: aform-err-def aform-val-def*)

**theorem** *affine-unop*:  
**assumes**  $x: x \in aform\text{-}err\ e\ X$   
**assumes**  $f: |f\ x - (a * x + b)| \leq d$   
**and**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows**  $f\ x \in aform\text{-}err\ e\ (affine\text{-}unop\ p\ a\ b\ d\ X)$

**proof** –  
**show** *?thesis*  
**unfolding** *affine-unop-def Let-def*  
**apply** (*auto simp: split-beta'*)  
**apply** (*rule aform-err-sum-list'*)  
**apply** *simp*  
**apply** (*rule aform-err-eqI*)  
**apply** (*rule aform-err-trunc-bound-eucl*)  
**apply** (*rule aform-err-addI*)  
**apply** (*rule aform-err-trunc-bound-eucl*)  
**apply** (*rule aform-err-trunc-bound-pdevsI*)  
**using** *e*  
**apply** *auto*  
**apply** (*rule aform-errI*)  
**apply** (*auto simp: aform-val-def*)  
**proof** –  
**define**  $x'$  **where**  $x' = (fst\ (fst\ X) + pdevs\text{-}val\ e\ (snd\ (fst\ X)))$   
**have**  $x - x': |x - x'| \leq snd\ X$   
**using**  $aform\text{-}errE[OF\ x]$   
**by** (*auto simp: x'-def aform-val-def*)  
**have**  $|f\ x - b - (a * fst\ (fst\ X) + a * pdevs\text{-}val\ e\ (snd\ (fst\ X)))| =$   
 $|f\ x - (a * x + b) + a * (x - x')|$   
**by** (*simp add: algebra-simps x'-def*)  
**also have**  $\dots \leq |f\ x - (a * x + b)| + |a * (x - x')|$   
**by** (*rule abs-triangle-ineq*)  
**also note** *f*

```

also have  $|a * (x - x')| \leq \text{truncate-up } p (|a| * \text{snd } X)$ 
by (rule truncate-up-le)
      (auto simp: abs-mult intro!: mult-left-mono x-x')
finally show  $|f x - b - (a * \text{fst } (\text{fst } X) + a * \text{pdevs-val } e (\text{snd } (\text{fst } X)))| \leq$ 
       $\text{truncate-up } p (|a| * \text{snd } X) + d$ 
by auto
qed
qed

lemma min-range-coeffs-ge:
 $|f x - (a * x + b)| \leq d$ 
if  $l \leq x$  and  $u: x \leq u$ 
      and  $f': \bigwedge y. y \in \{l .. u\} \implies (f \text{ has-real-derivative } f' y) (at y)$ 
      and  $a: \bigwedge y. y \in \{l..u\} \implies a \leq f' y$ 
      and  $d: d \geq (f u - f l - a * (u - l)) / 2 + |(f l + f u - a * (l + u)) / 2 - b|$ 
for  $a b d::\text{real}$ 
proof (rule order-trans[OF - d])
note  $f'\text{-at} = \text{has-field-derivative-at-within}[OF f']$ 
from  $l u$  have  $lu: x \in \{l .. u\}$  and  $llu: l \in \{l .. u\}$  by simp-all

define  $m$  where  $m = (f l + f u - a * (l + u)) / 2$ 
have  $|f x - (a * x + b)| = |f x - (a * x + m) + (m - b)|$  by (simp add:
algebra-simps)
also have  $\dots \leq |f x - (a * x + m)| + |m - b|$  by (rule abs-triangle-ineq)
also have  $|f x - (a * x + m)| \leq (f u - f l - a * (u - l)) / 2$ 
proof (rule abs-leI)
have  $f x \geq f l + a * (x - l)$  (is  $?l \geq ?r$ )
apply (rule order-trans) prefer 2
apply (rule linear-lower2[OF f'-at, of l u a])
subgoal by assumption
subgoal by (rule a)
subgoal
using  $lu$ 
by (auto intro!: mult-right-mono)
subgoal using  $lu$  by auto
done
also have  $a * x + m - (f u - f l - a * (u - l)) / 2 \leq ?r$ 
by (simp add: algebra-simps m-def field-simps)
finally (xtrans) show  $-(f x - (a * x + m)) \leq (f u - f l - a * (u - l)) / 2$ 
by (simp add: algebra-simps m-def divide-simps)
next
have  $f x \leq f u + a * (x - u)$ 
apply (rule order-trans)
apply (rule linear-upper2[OF f'-at, of l u a])
subgoal by assumption
subgoal by (rule a)
subgoal
using  $lu$ 
by (auto intro!: mult-right-mono)

```

```

    subgoal using lu by auto
    done
    also have ... ≤ a * x + m + (f u - f l - a * (u - l)) / 2
      by (simp add: m-def divide-simps algebra-simps)
    finally show f x - (a * x + m) ≤ (f u - f l - a * (u - l)) / 2
      by (simp add: algebra-simps m-def divide-simps)
  qed
  also have |m - b| = abs ((f l + f u - a * (l + u)) / 2 - b)
    unfolding m-def ..
  finally show |f x - (a * x + b)| ≤ (f u - f l - a * (u - l)) / 2 + |(f l + f u
- a * (l + u)) / 2 - b|
    by (simp)
  qed

lemma min-range-coeffs-le:
  |f x - (a * x + b)| ≤ d
  if l: l ≤ x and u: x ≤ u
    and f':  $\bigwedge y. y \in \{l .. u\} \implies (f \text{ has-real-derivative } f' y) (at y)$ 
    and a:  $\bigwedge y. y \in \{l .. u\} \implies f' y \leq a$ 
    and d:  $d \geq (f l - f u + a * (u - l)) / 2 + |(f l + f u - a * (l + u)) / 2 - b|$ 
  for a b d::real
proof (rule order-trans[OF - d])
  note f'-at = has-field-derivative-at-within[OF f]
  from l u have lu: x ∈ {l .. u} and llu: l ∈ {l .. u} by simp-all

  define m where m = (f l + f u - a * (l + u)) / 2
  have |f x - (a * x + b)| = |f x - (a * x + m) + (m - b)| by (simp add:
algebra-simps)
  also have ... ≤ |f x - (a * x + m)| + |m - b| by (rule abs-triangle-ineq)
  also have |f x - (a * x + m)| ≤ (f l - f u + a * (u - l)) / 2
  proof (rule abs-leI)
    have f x ≥ f u + a * (x - u) (is ?l ≥ ?r)
      apply (rule order-trans) prefer 2
      apply (rule linear-lower[OF f'-at, of l u a])
    subgoal by assumption
    subgoal by (rule a)
    subgoal
      using lu
      by (auto intro!: mult-right-mono)
    subgoal using lu by auto
    done
  also have a * x + m - (f l - f u + a * (u - l)) / 2 ≤ ?r
    using lu
    by (auto simp add: algebra-simps m-def field-simps intro!: mult-left-mono-neg)
  finally (xtrans) show - (f x - (a * x + m)) ≤ (f l - f u + a * (u - l)) / 2
    by (simp add: algebra-simps m-def divide-simps)
next
  have f x ≤ f l + a * (x - l)
    apply (rule order-trans)

```

```

    apply (rule linear-upper[OF f'-at, of l u a])
  subgoal by assumption
  subgoal by (rule a)
  subgoal
    using lu
    by (auto intro!: mult-right-mono)
  subgoal using lu by auto
  done
  also have ... ≤ a * x + m + (f l - f u + a * (u - l)) / 2
    using lu
    by (auto simp add: algebra-simps m-def field-simps intro!: mult-left-mono-neg)
  finally show f x - (a * x + m) ≤ (f l - f u + a * (u - l)) / 2
    by (simp add: algebra-simps m-def divide-simps)
qed
  also have |m - b| = abs ((f l + f u - a * (l + u)) / 2 - b)
    unfolding m-def ..
  finally show |f x - (a * x + b)| ≤ (f l - f u + a * (u - l)) / 2 + |(f l + f u
- a * (l + u)) / 2 - b|
    by (simp)
qed

```

**context includes floatarith-syntax begin**

```

definition range-reducer p l =
  (if l < 0 ∨ l > 2 * lb-pi p
  then approx p (Pi * (Num (-2)) * (Floor (Num (l * Float 1 (-1)) / Pi))) []
  else Some 0)

```

**lemmas** approx-emptyD = approx[OF bounded-by-None[of Nil], simplified]

**lemma** range-reducerE:

```

  assumes range-reducer p l = Some ivl
  obtains n::int where n * (2 * pi) ∈r ivl
proof (cases l ≥ 0 ∧ l ≤ 2 * lb-pi p)
  case False
    with assms have - ⌊l / (2 * pi)⌋ * (2 * pi) ∈r ivl
      by (auto simp: range-reducer-def bind-eq-Some-conv inverse-eq-divide
        algebra-simps dest!: approx-emptyD)
    then show ?thesis ..
  next
    case True then have real-of-int 0 * (2 * pi) ∈r ivl using assms
      by (auto simp: range-reducer-def zero-in-float-intervalI)
    then show ?thesis ..
qed

```

```

definition range-reduce-aform-err p X = do {
  r ← range-reducer p (lower (ivl-of-aform-err p X));
  Some (add-aform' p X (ivl-err (real-interval r)))
}

```

**lemma** *range-reduce-aform-errE*:  
**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**assumes**  $x: x \in aform\text{-}err\ e\ X$   
**assumes**  $range\text{-}reduce\text{-}aform\text{-}err\ p\ X = Some\ Y$   
**obtains**  $n::int$  **where**  $x + n * (2 * pi) \in aform\text{-}err\ e\ Y$   
**proof** –  
**from** *assms* **obtain**  $r$   
**where**  $x: x \in aform\text{-}err\ e\ X$   
**and**  $r: range\text{-}reducer\ p\ (lower\ (ivl\text{-}of\text{-}aform\text{-}err\ p\ X)) = Some\ r$   
**and**  $Y: Y = add\text{-}aform'\ p\ X\ (ivl\text{-}err\ (real\text{-}interval\ r))$   
**by** (*auto simp: range-reduce-aform-err-def bind-eq-Some-conv mid-err-def split: prod.splits*)  
**from**  $range\text{-}reducerE[OF\ r]$   
**obtain**  $n::int$  **where**  $n * (2 * pi) \in_r\ r$   
**by** *auto*  
**then have**  $n * (2 * pi) \in aform\text{-}err\ e\ (ivl\text{-}err\ (real\text{-}interval\ r))$   
**by** (*auto simp: aform-val-def ac-simps divide-simps abs-real-def set-of-eq intro!: aform-errI*)  
**from**  $add\text{-}aform'[OF\ e\ x\ this,\ of\ p]$   
**have**  $x + n * (2 * pi) \in aform\text{-}err\ e\ Y$   
**by** (*auto simp: Y*)  
**then show** *?thesis ..*  
**qed**

**definition** *min-range-mono*  $p\ F\ DF\ l\ u\ X = do\ \{$   
 $let\ L = Num\ l;$   
 $let\ U = Num\ u;$   
 $aivl \leftarrow approx\ p\ (Min\ (DF\ L)\ (DF\ U))\ [];$   
 $let\ a = lower\ aivl;$   
 $let\ A = Num\ a;$   
 $bivl \leftarrow approx\ p\ (Half\ (F\ L + F\ U - A * (L + U)))\ [];$   
 $let\ (b, be) = mid\text{-}err\ bivl;$   
 $let\ (B, Be) = (Num\ (float\text{-}of\ b), Num\ (float\text{-}of\ be));$   
 $divl \leftarrow approx\ p\ ((Half\ (F\ U - F\ L - A * (U - L))) + Be)\ [];$   
 $Some\ (affine\text{-}unop\ p\ a\ b\ (real\text{-}of\text{-}float\ (upper\ divl))\ X)$   
 $\}$

**lemma** *min-range-mono*:  
**assumes**  $x: x \in aform\text{-}err\ e\ X$   
**assumes**  $l \leq x\ x \leq u$   
**assumes**  $min\text{-}range\text{-}mono\ p\ F\ DF\ l\ u\ X = Some\ Y$   
**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**assumes**  $F: \bigwedge x. x \in \{real\text{-}of\text{-}float\ l .. u\} \implies interpret\text{-}floatarith\ (F\ (Num\ x))\ [] = f\ x$   
**assumes**  $DF: \bigwedge x. x \in \{real\text{-}of\text{-}float\ l .. u\} \implies interpret\text{-}floatarith\ (DF\ (Num\ x))\ [] = f'\ x$   
**assumes**  $f': \bigwedge x. x \in \{real\text{-}of\text{-}float\ l .. u\} \implies (f\ \text{has}\text{-}real\text{-}derivative\ f'\ x)\ (at\ x)$   
**assumes**  $f'\text{-}le: \bigwedge x. x \in \{real\text{-}of\text{-}float\ l .. u\} \implies min\ (f'\ l)\ (f'\ u) \leq f'\ x$

shows  $f x \in aform\text{-}err\ e\ Y$   
**proof** –  
**from** *assms* **obtain**  $a\ b\ be\ bivl\ divl$   
**where**  $bivl: (f\ l + f\ u - a * (l + u)) / 2 \in_r\ bivl$   
**and**  $Y: Y = affine\text{-}unop\ p\ a\ b\ (upper\ divl)\ X$   
**and**  $du: (f\ u - f\ l - a * (u - l)) / 2 + be \in_r\ divl$   
**and**  $a: a \leq f'\ l\ a \leq f'\ u$   
**and**  $b\text{-}def: b = (lower\ bivl + upper\ bivl) / 2$   
**and**  $be\text{-}def: be = (upper\ bivl - lower\ bivl) / 2$   
**by** (*auto simp: min-range-mono-def Let-def bind-eq-Some-conv mid-err-def set-of-eq*  
*simp del: eq-divide-eq-numeral1*  
*split: prod.splits if-splits dest!: approx-emptyD*)  
**have**  $diff\text{-}le: real\text{-}of\text{-}float\ a \leq f'\ y$  **if**  $real\text{-}of\text{-}float\ l \leq y\ y \leq u$  **for**  $y$   
**using**  $f'\text{-}le[of\ y]$  **that**  $a$   
**by** *auto*  
**have**  $le\text{-}be: |(f\ (l) + f\ (u) - a * (real\text{-}of\text{-}float\ l + u)) / 2 - b| \leq be$   
**using**  $bivl$   
**unfolding**  $b\text{-}def\ be\text{-}def$   
**by** (*auto simp: abs-real-def divide-simps set-of-eq*)  
**have**  $|f\ x - (a * x + b)| \leq upper\ divl$   
**apply** (*rule min-range-coeffs-ge*)  
**apply** (*rule <l ≤ x>*)  
**apply** (*rule <x ≤ u>*)  
**apply** (*rule f'*) **apply** *assumption*  
**using**  $diff\text{-}le$  **apply** *force*  
**apply** (*rule order-trans[OF add-mono[OF order-refl]]*)  
**apply** (*rule le-be*)  
**using**  $bivl\ du$   
**unfolding**  $b\text{-}def[symmetric]\ be\text{-}def[symmetric]$   
**by** (*auto simp: set-of-eq*)  
**from**  $affine\text{-}unop[where\ f=f\ and\ p = p, OF\ <x \in ->\ this\ e]$   
**have**  $f\ x \in aform\text{-}err\ e\ (affine\text{-}unop\ p\ (real\text{-}of\text{-}float\ a)\ b\ (upper\ divl)\ X)$   
**by** (*auto simp: Y*)  
**then show**  $?thesis$   
**by** (*simp add: Y b-def*)  
**qed**

**definition**  $min\text{-}range\text{-}antimono\ p\ F\ DF\ l\ u\ X = do\ \{$   
 $let\ L = Num\ l;$   
 $let\ U = Num\ u;$   
 $aivl \leftarrow approx\ p\ (Max\ (DF\ L)\ (DF\ U))\ [];$   
 $let\ a = upper\ aivl;$   
 $let\ A = Num\ a;$   
 $bivl \leftarrow approx\ p\ (Half\ (F\ L + F\ U - A * (L + U)))\ [];$   
 $let\ (b, be) = mid\text{-}err\ bivl;$   
 $let\ (B, Be) = (Num\ (float\text{-}of\ b), Num\ (float\text{-}of\ be));$   
 $divl \leftarrow approx\ p\ (Add\ (Half\ (F\ L - F\ U + A * (U - L)))\ Be)\ [];$   
 $Some\ (affine\text{-}unop\ p\ a\ b\ (real\text{-}of\text{-}float\ (upper\ divl))\ X)$   
 $\}$



}

**lemma** *min-range-antimono*:

**assumes**  $x: x \in \text{aform-err } e \ X$

**assumes**  $l \leq x \leq u$

**assumes** *min-range-antimono*  $p \ F \ DF \ l \ u \ X = \text{Some } Y$

**assumes**  $e: e \in \text{UNIV} \rightarrow \{-1 .. 1\}$

**assumes**  $F: \bigwedge x. x \in \{\text{real-of-float } l .. u\} \implies \text{interpret-floatarith } (F \ (\text{Num } x)) \ \square$   
 $= f \ x$

**assumes**  $DF: \bigwedge x. x \in \{\text{real-of-float } l .. u\} \implies \text{interpret-floatarith } (DF \ (\text{Num } x)) \ \square = f' \ x$

**assumes**  $f': \bigwedge x. x \in \{\text{real-of-float } l .. u\} \implies (f \ \text{has-real-derivative } f' \ x) \ (\text{at } x)$

**assumes**  $f'\text{-le}: \bigwedge x. x \in \{\text{real-of-float } l .. u\} \implies f' \ x \leq \max (f' \ l) (f' \ u)$

**shows**  $f \ x \in \text{aform-err } e \ Y$

**proof** –

**from** *assms* **obtain**  $a \ b \ be \ aivl \ bivl \ divl$

**where**  $bivl: (f \ l + f \ u - \text{real-of-float } a * (l + u)) / 2 \in_r \ bivl$

**and**  $Y: Y = \text{affine-unop } p \ a \ b \ (\text{real-of-float } (\text{upper } divl)) \ X$

**and**  $du: (f \ l - f \ u + a * (u - l)) / 2 + be \in_r \ divl$

**and**  $a: f' \ l \leq a \ f' \ u \leq a$

**and**  $a\text{-def}: a = \text{upper } aivl$

**and**  $b\text{-def}: b = (\text{lower } bivl + \text{upper } bivl) / 2$

**and**  $be\text{-def}: be = (\text{upper } bivl - \text{lower } bivl) / 2$

**by** (*auto simp: min-range-antimono-def Let-def bind-eq-Some-conv mid-err-def set-of-eq*

*simp del: eq-divide-eq-numeral1*

*split: prod.splits if-splits dest!: approx-emptyD*)

**have**  $\text{diff-le}: f' \ y \leq \text{real-of-float } a \ \text{if } \text{real-of-float } l \leq y \leq u \ \text{for } y$

**using**  $f'\text{-le}[of \ y]$  **that**  $a$

**by** *auto*

**have**  $\text{le-be}: |(f \ (l) + f \ (u) - a * (\text{real-of-float } l + u)) / 2 - b| \leq be$

**using**  $bivl$

**unfolding**  $b\text{-def}$   $be\text{-def}$

**by** (*auto simp: abs-real-def divide-simps set-of-eq*)

**have**  $|f \ x - (a * x + b)| \leq \text{upper } divl$

**apply** (*rule min-range-coeffs-le*)

**apply** (*rule*  $\langle l \leq x \rangle$ )

**apply** (*rule*  $\langle x \leq u \rangle$ )

**apply** (*rule*  $f'$ ) **apply** *assumption*

**using**  $\text{diff-le}$  **apply** *force*

**apply** (*rule order-trans*[*OF add-mono*[*OF order-refl*]])

**apply** (*rule le-be*)

**using**  $du \ bivl$

**unfolding**  $b\text{-def}$ [*symmetric*]  $be\text{-def}$ [*symmetric*]

**by** (*auto simp: set-of-eq*)

**from** *affine-unop*[**where**  $f=f$  **and**  $p = p$ , *OF*  $\langle x \in \cdot \rangle$  *this*  $e$ ]

**have**  $f \ x \in \text{aform-err } e \ (\text{affine-unop } p \ (\text{real-of-float } a) \ b \ (\text{upper } divl)) \ X$

**by** (*auto simp: Y*)

**then show** *?thesis*

by (*simp add: Y b-def*)  
**qed**

**definition** *cos-aform-err*  $p X = do \{$   
 $X \leftarrow range-reduce-aform-err\ p\ X;$   
 $let\ ivl = ivl-of-aform-err\ p\ X;$   
 $let\ l = lower\ ivl;$   
 $let\ u = upper\ ivl;$   
 $let\ L = Num\ l;$   
 $let\ U = Num\ u;$   
 $if\ l \geq 0 \wedge u \leq lb-pi\ p\ then$   
 $\quad min-range-antimono\ p\ Cos\ (\lambda x. (Minus\ (Sin\ x)))\ l\ u\ X$   
 $else\ if\ l \geq ub-pi\ p \wedge u \leq 2 * lb-pi\ p\ then$   
 $\quad min-range-mono\ p\ Cos\ (\lambda x. (Minus\ (Sin\ x)))\ l\ u\ X$   
 $else\ do \{$   
 $\quad Some\ (ivl-err\ (real-interval\ (cos-float-interval\ p\ ivl)))$   
 $\quad \}$   
 $\}$   
 $\}$

**lemma** *abs-half-enclosure*:  
**fixes**  $r::real$   
**assumes**  $bl \leq r \leq bu$   
**shows**  $|r - (bl + bu) / 2| \leq (bu - bl) / 2$   
**using** *assms*  
**by** (*auto simp: abs-real-def divide-simps*)

**lemma** *cos-aform-err*:  
**assumes**  $x: x \in aform-err\ e\ X0$   
**assumes**  $cos-aform-err\ p\ X0 = Some\ Y$   
**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows**  $cos\ x \in aform-err\ e\ Y$   
**proof** –  
**from** *assms* **obtain**  $X\ ivl\ l\ u$  **where**  
 $X: range-reduce-aform-err\ p\ X0 = Some\ X$   
**and** *ivl-def*:  $ivl = ivl-of-aform-err\ p\ X$   
**and** *l-def*:  $l = lower\ ivl$   
**and** *u-def*:  $u = upper\ ivl$   
**by** (*auto simp: cos-aform-err-def bind-eq-Some-conv*)  
**from** *range-reduce-aform-errE*[*OF*  $e\ x\ X$ ]  
**obtain**  $n$  **where**  $xn: x + real-of-int\ n * (2 * pi) \in aform-err\ e\ X$   
**by** *auto*  
**define**  $xn$  **where**  $xn = x + n * (2 * pi)$   
**with**  $xn$  **have**  $xn \in aform-err\ e\ X$  **by** *auto*  
**from** *ivl-of-aform-err*[*OF*  $e\ xn$ , *of*  $p$ , *folded* *ivl-def*]  
**have**  $xn \in_r\ ivl$  .  
**then** **have**  $lxn: l \leq xn$  **and**  $uxn: xn \leq u$   
**by** (*auto simp: l-def u-def set-of-eq*)  
**consider**  $l \geq 0\ u \leq lb-pi\ p$   
 $|l < 0 \vee u > lb-pi\ p\ l \geq ub-pi\ p\ u \leq 2 * lb-pi\ p$

```

|  $l < 0 \vee u > lb\text{-}pi\ p\ l < ub\text{-}pi\ p \vee u > 2 * lb\text{-}pi\ p$ 
by arith
then show ?thesis
proof cases
case 1
then have min-eq-Some: min-range-antimono p Cos ( $\lambda x. Minus (Sin\ x)$ ) l u X
= Some Y
and bounds:  $0 \leq l\ u \leq (lb\text{-}pi\ p)$ 
using assms(2)
unfolding cos-aform-err-def X l-def u-def
by (auto simp: X Let-def simp flip: l-def u-def ivl-def split: prod.splits)
have bounds:  $0 \leq l\ u \leq pi$  using bounds pi-boundaries[of p] by auto
have diff-le:  $-\sin\ y \leq \max(-\sin(\text{real-of-float}\ l))(-\sin(\text{real-of-float}\ u))$ 
if  $\text{real-of-float}\ l \leq y \leq \text{real-of-float}\ u$  for y
proof -
consider  $y \leq pi / 2 \mid y \geq pi / 2$  by arith
then show ?thesis
proof cases
case 1
then have  $-\sin\ y \leq -\sin\ l$ 
using that bounds
by (auto intro!: sin-monotone-2pi-le)
then show ?thesis by auto
next
case 2
then have  $-\sin\ y \leq -\sin\ u$ 
using that bounds
unfolding sin-minus-pi[symmetric]
apply (intro sin-monotone-2pi-le)
by (auto intro!: )
then show ?thesis by auto
qed
qed
have  $\cos\ xn \in aform\text{-}err\ e\ Y$ 
apply (rule min-range-antimono[OF xn lxn uxn min-eq-Some e, where  $f' = \lambda x. -\sin\ x$ ])
subgoal by simp
subgoal by simp
subgoal by (auto intro!: derivative-eq-intros)
subgoal by (rule diff-le) auto
done
then show ?thesis
unfolding xn-def
by simp
next
case 2
then have min-eq-Some: min-range-mono p Cos ( $\lambda x. Minus (Sin\ x)$ ) l u X =
Some Y
and bounds:  $ub\text{-}pi\ p \leq l\ u \leq 2 * lb\text{-}pi\ p$ 

```

```

using assms(2)
unfolding cos-aform-err-def X
by (auto simp: X Let-def simp flip: l-def u-def ivl-def split: prod.splits)
have bounds:  $\pi \leq l \leq 2 * \pi$  using bounds pi-boundaries[of p] by auto
have diff-le:  $\min(-\sin(\text{real-of-float } l), -\sin(\text{real-of-float } u)) \leq -\sin y$ 
if  $\text{real-of-float } l \leq y \leq \text{real-of-float } u$  for y
proof -
consider  $y \leq 3 * \pi / 2 \mid y \geq 3 * \pi / 2$  by arith
then show ?thesis
proof cases
case 1
then have  $-\sin l \leq -\sin y$ 
unfolding sin-minus-pi[symmetric]
apply (intro sin-monotone-2pi-le)
using that bounds
by (auto)
then show ?thesis by auto
next
case 2
then have  $-\sin u \leq -\sin y$ 
unfolding sin-2pi-minus[symmetric]
using that bounds
apply (intro sin-monotone-2pi-le)
by (auto intro!: )
then show ?thesis by auto
qed
qed
have  $\cos xn \in \text{aform-err } e \ Y$ 
apply (rule min-range-mono[OF xn lxn uxn min-eq-Some e, where  $f'=\lambda x. -\sin x$ ])
subgoal by simp
subgoal by simp
subgoal by (auto intro!: derivative-eq-intros)
subgoal by (rule diff-le) auto
done
then show ?thesis
unfolding xn-def
by simp
next
case 3
then obtain  $ivl'$  where
 $\text{cos-float-interval } p \ ivl = ivl'$ 
 $Y = ivl\text{-err } (\text{real-interval } ivl')$ 
using assms(2)
unfolding cos-aform-err-def X l-def u-def
by (auto simp: X simp flip: l-def u-def ivl-def split: prod.splits)
with  $\text{cos-float-intervalI}[OF \langle xn \in_r \ ivl \rangle, \text{ of } p]$ 
show ?thesis
by (auto simp: xn-def)

```

**qed**  
**qed**

**definition** *sqrt-aform-err*  $p X = do \{$   
*let*  $ivl = ivl-of-aform-err\ p\ X;$   
*let*  $l = lower\ ivl;$   
*let*  $u = upper\ ivl;$   
*if*  $0 < l$  *then*  $min-range-mono\ p\ Sqrt\ (\lambda x. Half\ (Inverse\ (Sqrt\ x)))\ l\ u\ X$   
*else*  $Some\ (ivl-err\ (real-interval\ (sqrt-float-interval\ p\ ivl)))$   
 $\}$

**lemma** *sqrt-aform-err*:

**assumes**  $x: x \in aform-err\ e\ X$   
**assumes** *sqrt-aform-err*  $p\ X = Some\ Y$   
**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows**  $sqrt\ x \in aform-err\ e\ Y$

**proof** –

**obtain**  $l\ u\ ivl$   
**where**  $ivl-def: ivl = ivl-of-aform-err\ p\ X$   
**and**  $l-def: l = lower\ ivl$   
**and**  $u-def: u = upper\ ivl$   
**by** *auto*

**from** *ivl-of-aform-err*[*OF*  $e\ x$ , *of*  $p$ , *folded*  $ivl-def$ ]

**have**  $ivl: x \in_r\ ivl$ .

**then have**  $lx: l \leq x$  **and**  $ux: x \leq u$

**by** (*auto simp flip: ivl-def simp: l-def u-def set-of-eq*)

**consider**  $l > 0 \mid l \leq 0$

**by** *arith*

**then show** *?thesis*

**proof** *cases*

**case** 1

**then have** *min-eq-Some*:  $min-range-mono\ p\ Sqrt\ (\lambda x. Half\ (Inverse\ (Sqrt\ x)))$

$l\ u\ X = Some\ Y$

**and** *bounds*:  $0 < l$

**using** *assms*(2)

**unfolding** *sqrt-aform-err-def*

**by** (*auto simp: Let-def simp flip: l-def u-def ivl-def split: prod.splits*)

**have**  $sqrt\ x \in aform-err\ e\ Y$

**apply** (*rule min-range-mono*[*OF*  $x\ lx\ ux\ min-eq-Some\ e$ , **where**  $f' = \lambda x. 1 / (2 * sqrt\ x)$ ])

**subgoal by** *simp*

**subgoal by** (*simp add: divide-simps*)

**subgoal using** *bounds* **by** (*auto intro!: derivative-eq-intros simp: inverse-eq-divide*)

**subgoal using**  $\langle l > 0 \rangle$  **by** (*auto simp: inverse-eq-divide min-def divide-simps*)

**done**

**then show** *?thesis*

**by** *simp*

**next**

**case** 2

```

then have  $Y = \text{ivl-err } (\text{real-interval } (\text{sqrt-float-interval } p \text{ ivl}))$ 
  using assms(2)
  unfolding sqrt-aform-err-def
  by (auto simp: Let-def simp flip: ivl-def l-def u-def split: prod.splits)
with sqrt-float-intervalI[OF ivl]
show ?thesis
  by (auto simp: set-of-eq)
qed
qed

definition ln-aform-err  $p X = \text{do } \{$ 
  let  $\text{ivl} = \text{ivl-of-aform-err } p X;$ 
  let  $l = \text{lower ivl};$ 
  if  $0 < l$  then min-range-mono  $p Ln \text{ inverse } l (\text{upper ivl}) X$ 
  else None
 $\}$ 

lemma ln-aform-err:
  assumes  $x: x \in \text{aform-err } e X$ 
  assumes  $\text{ln-aform-err } p X = \text{Some } Y$ 
  assumes  $e: e \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
  shows  $\text{ln } x \in \text{aform-err } e Y$ 
proof  $-$ 
  obtain  $\text{ivl } l u$ 
    where  $l\text{-def}: l = \text{lower ivl}$ 
    and  $u\text{-def}: u = \text{upper ivl}$ 
    and  $\text{ivl-def}: \text{ivl} = \text{ivl-of-aform-err } p X$ 
  by auto
from ivl-of-aform-err[OF e x, of p, folded ivl-def]
have  $x \in_r \text{ivl} .$ 
then have  $lx: l \leq x$  and  $ux: x \leq u$ 
  by (auto simp: set-of-eq l-def u-def)
consider  $l > 0 \mid l \leq 0$ 
  by arith
then show ?thesis
proof cases
  case  $1$ 
then have min-eq-Some: min-range-mono  $p Ln \text{ inverse } l u X = \text{Some } Y$ 
  and  $\text{bounds}: 0 < l$ 
  using assms(2)
  unfolding ln-aform-err-def
  by (auto simp: Let-def simp flip: ivl-def l-def u-def split: prod.splits if-splits)
have  $\text{ln } x \in \text{aform-err } e Y$ 
  apply (rule min-range-mono[OF x lx ux min-eq-Some e, where f'=inverse])
  subgoal by simp
  subgoal by (simp add: divide-simps)
  subgoal using  $\text{bounds}$  by (auto intro!: derivative-eq-intros simp: inverse-eq-divide)
  subgoal using  $\langle l > 0 \rangle$  by (auto simp: inverse-eq-divide min-def divide-simps)
  done

```

```

    then show ?thesis
      by simp
  next
    case 2
    then show ?thesis using assms
      by (auto simp: ln-aform-err-def Let-def l-def ivl-def)
  qed
qed

```

```

definition exp-aform-err p X = do {
  let ivl = ivl-of-aform-err p X;
  min-range-mono p Exp Exp (lower ivl) (upper ivl) X
}

```

**lemma** exp-aform-err:

```

  assumes x: x ∈ aform-err e X
  assumes exp-aform-err p X = Some Y
  assumes e: e ∈ UNIV → {-1 .. 1}
  shows exp x ∈ aform-err e Y
proof -
  obtain l u ivl
    where l-def: l = lower ivl
      and u-def: u = upper ivl
      and ivl-def: ivl = ivl-of-aform-err p X
    by auto
  from ivl-of-aform-err[OF e x, of p, folded ivl-def]
  have x ∈r ivl .
  then have lx: l ≤ x and ux: x ≤ u
    by (auto simp: ivl-def l-def u-def set-of-eq)
  have min-eq-Some: min-range-mono p Exp Exp l u X = Some Y
    using assms(2)
    unfolding exp-aform-err-def
    by (auto simp: Let-def simp flip: ivl-def u-def l-def split: prod.splits if-splits)
  have exp x ∈ aform-err e Y
    apply (rule min-range-mono[OF x lx ux min-eq-Some e, where f'=exp])
    subgoal by simp
    subgoal by (simp add: divide-simps)
    subgoal by (auto intro!: derivative-eq-intros simp: inverse-eq-divide)
    subgoal by (auto simp: inverse-eq-divide min-def divide-simps)
    done
  then show ?thesis
    by simp
qed

```

```

definition arctan-aform-err p X = do {
  let l = Inf-aform-err p X;
  let u = Sup-aform-err p X;
  min-range-mono p Arctan (λx. 1 / (Num 1 + x * x)) l u X
}

```

```

lemma pos-add-nonneg-ne-zero:  $a > 0 \implies b \geq 0 \implies a + b \neq 0$ 
  for  $a\ b::\text{real}$ 
  by arith

lemma arctan-aform-err:
  assumes  $x: x \in \text{aform-err } e\ X$ 
  assumes arctan-aform-err  $p\ X = \text{Some } Y$ 
  assumes  $e: e \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
  shows  $\text{arctan } x \in \text{aform-err } e\ Y$ 
proof -
  obtain  $l\ u$  where  $l: l = \text{Inf-aform-err } p\ X$ 
    and  $u: u = \text{Sup-aform-err } p\ X$ 
    by auto
  from  $x\ l\ u$  have  $lx: l \leq x$  and  $ux: x \leq u$ 
    using Inf-Sup-aform-err[OF  $e$ , of  $X\ p$ ]
    by auto
  have min-eq-Some: min-range-mono  $p\ \text{Arctan } (\lambda x. 1 / (\text{Num } 1 + x * x))\ l\ u\ X$ 
    =  $\text{Some } Y$ 
    using assms(2)
    unfolding arctan-aform-err-def  $l\ u$ 
    by (auto simp: l[symmetric] u[symmetric] split: prod.splits if-splits)
  have  $\text{arctan } x \in \text{aform-err } e\ Y$ 
    apply (rule min-range-mono[OF  $x\ lx\ ux$  min-eq-Some  $e$ , where  $f' = \lambda x. \text{inverse } (1 + x^2)$ ])
    subgoal by simp
    subgoal by (simp add: power2-eq-square inverse-eq-divide)
    subgoal by (auto intro!: derivative-eq-intros simp: inverse-eq-divide)
    subgoal for  $x$ 
      apply (cases  $x \leq 0$ )
      subgoal
        apply (rule min.coboundedI1)
        apply (rule deriv-nonneg-imp-mono[of real-of-float  $l\ x$ ])
      by (auto intro!: derivative-eq-intros simp: mult-le-0-iff pos-add-nonneg-ne-zero)
      subgoal
        apply (rule min.coboundedI2)
        apply (rule le-imp-inverse-le)
        by (auto intro!: power-mono add-pos-nonneg)
      done
    done
  then show ?thesis
    by simp
qed

```

## 5.6 Power, TODO: compare with Min-range approximation?!

```

definition power-aform-err  $p\ (X::\text{aform-err})\ n =$ 
  (if  $n = 0$  then  $((1, \text{zero-pdevs}), 0)$ 
   else if  $n = 1$  then  $X$ 

```



```

else
  let x0 = float-of (fst (fst X));
      xs = snd (fst X);
      xe = float-of (snd X);
      C = the (approx p (Num x0  $\hat{e}$  n) []);
      (c, ce) = mid-err C;
      NX = the (approx p (Num (of-nat n) * (Num x0  $\hat{e}$  (n - 1))) []);
      (nx, nxe) = mid-err NX;
      Y = scaleR-pdevs nx xs;
      (Y', Y-err) = trunc-bound-pdevs p Y;
      t = tdev' p xs;
      Ye = truncate-up p (nxe * t);
      err = the (approx p
        (Num (of-nat n) * Num xe * Abs (Num x0)  $\hat{e}$  (n - 1) +
         Sume ( $\lambda$ k. Num (of-nat (n choose k)) * Abs (Num x0)  $\hat{e}$  (n - k) * (Num
  xe + Num (float-of t))  $\hat{e}$  k)
        [2.. $\text{Suc } n$ ])) []);
      ERR = upper err
  in ((c, Y'), sum-list' p [ce, Y-err, Ye, real-of-float ERR])

```

**lemma** *bounded-by-Nil*: *bounded-by* [] []  
**by** (*auto simp*: *bounded-by-def*)

**lemma** *plain-floatarith-approx*:  
**assumes** *plain-floatarith 0 f*  
**shows** *interpret-floatarith f []  $\in_r$  (the (approx p f []))*  
**proof** –  
**from** *plain-floatarith-approx-not-None[OF assms(1), of Nil p]*  
**obtain** *ivl* **where** *approx p f [] = Some ivl*  
**by** *auto*  
**from** *this approx[OF bounded-by-Nil this]*  
**show** *?thesis*  
**by** *auto*  
**qed**

**lemma** *plain-floatarith-Sum<sub>e</sub>*:  
*plain-floatarith n (Sum<sub>e</sub> f xs)  $\longleftrightarrow$  list-all ( $\lambda$ i. plain-floatarith n (f i)) xs*  
**by** (*induction xs*) (*auto simp*: *zero-floatarith-def plus-floatarith-def*)

**lemma** *sum-list'-float[simp]*: *sum-list' p xs  $\in$  float*  
**by** (*induction xs rule*: *rev-induct*) (*auto simp*: *sum-list'-def eucl-truncate-up-real-def*)

**lemma** *tdev'-float[simp]*: *tdev' p xs  $\in$  float*  
**by** (*auto simp*: *tdev'-def*)

**lemma**  
**fixes** *x y::real*  
**assumes** *abs (x - y)  $\leq$  e*  
**obtains** *err* **where** *x = y + err abs err  $\leq$  e*

```

using assms
apply atomize-elim
apply (rule exI[where  $x=x - y$ ])
by (auto simp: abs-real-def)

theorem power-aform-err:
  assumes  $x \in \text{aform-err } e \ X$ 
  assumes floats[simp]:  $\text{fst } (\text{fst } X) \in \text{float } \text{snd } X \in \text{float}$ 
  assumes  $e: e \in \text{UNIV} \rightarrow \{-1 .. 1\}$ 
  shows  $x \wedge n \in \text{aform-err } e \ (\text{power-aform-err } p \ X \ n)$ 
proof -
  consider  $n = 0 \mid n = 1 \mid n \geq 2$ 
    by arith
  then show ?thesis
  proof cases
    case 1
    then show ?thesis by (auto simp: aform-err-def power-aform-err-def aform-val-def)
  next
    case 2
    then show ?thesis
      using assms
      by (auto simp: aform-err-def power-aform-err-def aform-val-def)
  next
    case  $n: 3$ 
    define  $x0$  where  $x0 = (\text{fst } (\text{fst } X))$ 
    define  $xs$  where  $xs = \text{snd } (\text{fst } X)$ 
    define  $xe$  where  $xe = (\text{snd } X)$ 
    have [simp]:  $x0 \in \text{float } xe \in \text{float}$  using assms by (auto simp: x0-def xe-def)

    define  $xe'$  where  $xe' = x - \text{aform-val } e \ (x0, xs)$ 
    from aform-errE[OF assms(1)]
    have  $xe'$ :  $|xe'| \leq xe$ 
      by (auto simp: x0-def xs-def xe-def xe'-def)
    then have xe-nonneg:  $0 \leq xe$ 
      by auto

    define  $t$  where  $t = \text{tdev}' \ p \ xs$ 
    have  $t$ :  $\text{tdev } xs \leq t \ t \in \text{float}$  by (auto simp add: t-def tdev'-le)
    then have t-nonneg:  $0 \leq t$  using tdev-nonneg[of xs] by arith
    note  $t\text{-pdevs} = \text{abs-pdevs-val-le-tdev}$ [OF e, THEN order-trans, OF t(1)]

    have rewr1:  $\{..n\} = (\text{insert } 0 \ (\text{insert } 1 \ \{2..n\}))$  using  $n$  by auto
    have  $x = (\text{pdevs-val } e \ xs + xe') + x0$ 
      by (simp add: xe'-def aform-val-def)
    also have  $\dots \wedge n = x0 \wedge n + n * x0 \wedge (n - \text{Suc } 0) * \text{pdevs-val } e \ xs +$ 
       $(n * xe' * x0 \wedge (n - \text{Suc } 0) +$ 
       $(\sum k = 2..n. \text{real } (n \text{ choose } k) * (\text{pdevs-val } e \ xs + xe') \wedge k * x0 \wedge (n - k)))$ 
      (is - = - + ?err)
    apply (subst binomial-ring)

```

```

unfolding rewr1
using n
apply (simp add: algebra-simps)
done
also

let ?ERR = (Num (of-nat n) * Num (float-of xe) * Abs (Num (float-of x0)))  $\hat{e}$ 
(n - 1) +
(Sume (λk. Num (of-nat (n choose k)) * Abs (Num (float-of x0)))  $\hat{e}$  (n -
k) *
(Num (float-of xe) + Num (float-of t))  $\hat{e}$  k)
[2..<Suc n])
define err where err = the (approx p ?ERR [])
define ERR where ERR = upper err
have ERR: abs ?err ≤ ERR
proof -
have err-aerr: abs (?err) ≤ n * xe * abs x0  $\hat{e}$  (n - Suc 0) +
( $\sum k = 2..n.$  real (n choose k) * (t + xe)  $\hat{e}$  k * abs x0  $\hat{e}$  (n - k))
(is - ≤ ?aerr)
by (auto simp: abs-mult power-abs intro!: sum-mono mult-mono power-mono
xe'
mult-nonneg-nonneg zero-le-power t-nonneg xe-nonneg add-nonneg-nonneg
sum-abs[THEN order-trans] abs-triangle-ineq[THEN order-trans] add-mono
t-pdevs)
also
have rewr:  $\{2 .. n\} = \{2 ..<Suc n\}$ 
using n
by auto
have plain-floatarith 0 ?ERR
by (auto simp add: zero-floatarith-def plain-floatarith-Sume times-floatarith-def
plus-floatarith-def intro!: list-allI)
from plain-floatarith-approx[OF this, of p]
have ERR  $\geq$  ?aerr
using n
by (auto simp: set-of-eq err-def ERR-def sum-list-distinct-conv-sum-set rewr
t x0-def
algebra-simps)
finally show ?thesis .
qed

let ?x0n = Num (float-of x0)  $\hat{e}$  n
define C where C = the (approx p ?x0n [])
have plain-floatarith 0 ?x0n by simp
from plain-floatarith-approx[OF this, of p]
have C: x0  $\hat{e}$  n  $\in$   $\{lower\ C .. upper\ C\}$ 
by (auto simp: C-def x0-def set-of-eq)

define c where c = fst (mid-err C)
define ce where ce = snd (mid-err C)

```

```

define  $ce'$  where  $ce' = x0 \wedge n - c$ 
have  $ce'$ :  $abs (ce') \leq ce$ 
  using  $C$ 
  by (auto simp: ce'-def c-def ce-def abs-diff-le-iff mid-err-def divide-simps)
have  $x0 \wedge n = c + ce'$  by (simp add: ce'-def)
also

let  $?NX = (Num (of-nat n) * (Num (float-of x0) \wedge_e (n - 1)))$ 
define  $NX$  where  $NX = the (approx p ?NX [])$ 
have plain-floatarith 0 ?NX by (simp add: times-floatarith-def)
from plain-floatarith-approx[OF this, of p]
have  $NX$ :  $n * x0 \wedge (n - 1) \in \{lower\ NX .. upper\ NX\}$ 
  by (auto simp: NX-def x0-def set-of-eq)

define  $nx$  where  $nx = fst (mid-err NX)$ 
define  $nxe$  where  $nxe = snd (mid-err NX)$ 
define  $nx'$  where  $nx' = n * x0 \wedge (n - 1) - nx$ 
define  $Ye$  where  $Ye = truncate-up p (nxe * t)$ 
have  $Ye$ :  $Ye \geq nxe * t$  by (auto simp: Ye-def truncate-up-le)
have  $nx$ :  $abs (nx') \leq nxe\ 0 \leq nxe$ 
  using  $NX$ 
  by (auto simp: nx-def nxe-def abs-diff-le-iff mid-err-def divide-simps nx'-def)
have  $Ye$ :  $abs (nx' * pdevs-val e xs) \leq Ye$ 
  by (auto simp: Ye-def abs-mult intro!: truncate-up-le mult-mono nx t-pdevs)
have  $n * x0 \wedge (n - Suc\ 0) = nx + nx'$  by (simp add: nx'-def)
also

define  $Y$  where  $Y = scaleR-pdevs\ nx\ xs$ 
have  $Y$ :  $pdevs-val\ e\ Y = nx * pdevs-val\ e\ xs$ 
  by (simp add: Y-def)
have  $(nx + nx') * pdevs-val\ e\ xs = pdevs-val\ e\ Y + nx' * pdevs-val\ e\ xs$ 
  unfolding  $Y$  by (simp add: algebra-simps)
also

define  $Y'$  where  $Y' = fst (trunc-bound-pdevs p Y)$ 
define  $Y-err$  where  $Y-err = snd (trunc-bound-pdevs p Y)$ 
have  $Y-err$ :  $abs (- pdevs-val\ e (trunc-err-pdevs p Y)) \leq Y-err$ 
  by (auto simp: Y-err-def trunc-bound-pdevs-def abs-pdevs-val-le-tdev' e)
have  $pdevs-val\ e\ Y = pdevs-val\ e\ Y' + - pdevs-val\ e (trunc-err-pdevs p Y)$ 
  by (simp add: Y'-def trunc-bound-pdevs-def pdevs-val-trunc-err-pdevs)
finally
have  $|x \wedge n - aform-val\ e (c, Y')| =$ 
   $|ce' + - pdevs-val\ e (trunc-err-pdevs p Y) + nx' * pdevs-val\ e\ xs + ?err|$ 
  by (simp add: algebra-simps aform-val-def)
also have  $\dots \leq ce + Y-err + Ye + ERR$ 
  by (intro ERR abs-triangle-ineq[THEN order-trans] add-mono ce' Ye Y-err)
also have  $\dots \leq sum-list' p [ce, Y-err, Ye, real-of-float ERR]$ 
  by (auto intro!: sum-list'-sum-list-le)
finally show  $?thesis$ 

```

```

using n
by (intro aform-errI)
      (auto simp: power-aform-err-def c-def Y'-def C-def Y-def ERR-def x0-def
nx-def xs-def NX-def
      ce-def Y-err-def Ye-def xe-def nxe-def t-def Let-def split-beta' set-of-eq
err-def)
qed
qed

```

**definition** [*code-abbrev*]: *is-float*  $r \longleftrightarrow r \in \text{float}$

**lemma** [*code*]: *is-float* (*real-of-float*  $f$ ) = *True*

**by** (*auto simp: is-float-def*)

**definition** *powr-aform-err*  $p$   $X$   $A$  = (  
 if *Inf-aform-err*  $p$   $X$  > 0 then do {  
    $L \leftarrow \text{ln-aform-err } p$   $X$ ;  
    $\text{exp-aform-err } p$  (*mult-aform'*  $p$   $A$   $L$ )  
 }  
 else *approx-bin*  $p$  (*powr-float-interval*  $p$ )  $X$   $A$ )

**lemma** *interval-extension-powr: interval-extension2* (*powr-float-interval*  $p$ ) (*powr*)

**using** *powr-float-interval-eqI*[*of*  $p$ ]

**by** (*auto simp: interval-extension2-def*)

**theorem** *powr-aform-err*:

**assumes**  $x$ :  $x \in \text{aform-err } e$   $X$

**assumes**  $a$ :  $a \in \text{aform-err } e$   $A$

**assumes**  $e$ :  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$

**assumes**  $Y$ : *powr-aform-err*  $p$   $X$   $A$  = *Some*  $Y$

**shows**  $x$  *powr*  $a \in \text{aform-err } e$   $Y$

**proof** *cases*

**assume** *pos*: *Inf-aform-err*  $p$   $X$  > 0

**with** *Inf-Sup-aform-err*[*OF*  $e$ , *of*  $X$   $p$ ]  $x$

**have**  $x$  > 0 **by** *auto*

**then have**  $x$  *powr*  $a = \text{exp}$  ( $a * \text{ln } x$ )

**by** (*simp add: powr-def*)

**also**

**from** *pos* **obtain**  $L$  **where**  $L$ : *ln-aform-err*  $p$   $X$  = *Some*  $L$

**and**  $E$ : *exp-aform-err*  $p$  (*mult-aform'*  $p$   $A$   $L$ ) = *Some*  $Y$

**using**  $Y$

**by** (*auto simp: bind-eq-Some-conv powr-aform-err-def*)

**from** *ln-aform-err*[*OF*  $x$   $L$   $e$ ] **have**  $\text{ln } x \in \text{aform-err } e$   $L$  .

**from** *mult-aform'E*[*OF*  $e$   $a$  *this*] **have**  $a * \text{ln } x \in \text{aform-err } e$  (*mult-aform'*  $p$   $A$   $L$ ) .

**from** *exp-aform-err*[*OF* *this*  $E$   $e$ ]

**have**  $\text{exp}$  ( $a * \text{ln } x$ )  $\in \text{aform-err } e$   $Y$  .

**finally show** *?thesis* .

**next**

**from**  $x$   $a$  **have**  $xa$ :  $x \in \text{aform-err } e$  (*fst*  $X$ , *snd*  $X$ )  $a \in \text{aform-err } e$  (*fst*  $A$ , *snd*

```

A) by simp-all
  assume  $\neg$  Inf-aform-err p X > 0
  then have approx-bin p (powr-float-interval p) (fst X, snd X) (fst A, snd A) =
  Some Y
    using Y by (auto simp: powr-aform-err-def)
  from approx-binE[OF interval-extension-powr xa this e]
  show x powr a  $\in$  aform-err e Y .
qed

```

```

fun
  approx-floatarith :: nat  $\Rightarrow$  floatarith  $\Rightarrow$  aform-err list  $\Rightarrow$  (aform-err) option
where
  approx-floatarith p (Add a b) vs =
    do {
      a1  $\leftarrow$  approx-floatarith p a vs;
      a2  $\leftarrow$  approx-floatarith p b vs;
      Some (add-aform' p a1 a2)
    }
| approx-floatarith p (Mult a b) vs =
    do {
      a1  $\leftarrow$  approx-floatarith p a vs;
      a2  $\leftarrow$  approx-floatarith p b vs;
      Some (mult-aform' p a1 a2)
    }
| approx-floatarith p (Inverse a) vs =
    do {
      a  $\leftarrow$  approx-floatarith p a vs;
      inverse-aform-err p a
    }
| approx-floatarith p (Minus a) vs =
  map-option (apfst uminus-aform) (approx-floatarith p a vs)
| approx-floatarith p (Num f) vs =
  Some (num-aform (real-of-float f), 0)
| approx-floatarith p (Var i) vs =
  (if i < length vs then Some (vs ! i) else None)
| approx-floatarith p (Abs a) vs =
    do {
      r  $\leftarrow$  approx-floatarith p a vs;
      let ivl = ivl-of-aform-err p r;
          let i = lower ivl;
              let s = upper ivl;
          if i > 0 then Some r
          else if s < 0 then Some (apfst uminus-aform r)
          else do {
            Some (ivl-err (real-interval (abs-interval ivl)))
          }
    }
| approx-floatarith p (Min a b) vs =
    do {

```

```

    a1 ← approx-floatarith p a vs;
    a2 ← approx-floatarith p b vs;
    Some (min-aform-err p a1 a2)
  }
| approx-floatarith p (Max a b) vs =
  do {
    a1 ← approx-floatarith p a vs;
    a2 ← approx-floatarith p b vs;
    Some (max-aform-err p a1 a2)
  }
| approx-floatarith p (Floor a) vs =
  approx-un p (λivl. Some (floor-float-interval ivl)) (approx-floatarith p a vs)
| approx-floatarith p (Cos a) vs =
  do {
    a ← approx-floatarith p a vs;
    cos-aform-err p a
  }
| approx-floatarith p Pi vs = Some (ivl-err (real-interval (pi-float-interval p)))
| approx-floatarith p (Sqrt a) vs =
  do {
    a ← approx-floatarith p a vs;
    sqrt-aform-err p a
  }
| approx-floatarith p (Ln a) vs =
  do {
    a ← approx-floatarith p a vs;
    ln-aform-err p a
  }
| approx-floatarith p (Arctan a) vs =
  do {
    a ← approx-floatarith p a vs;
    arctan-aform-err p a
  }
| approx-floatarith p (Exp a) vs =
  do {
    a ← approx-floatarith p a vs;
    exp-aform-err p a
  }
| approx-floatarith p (Power a n) vs =
  do {
    ((a, as), e) ← approx-floatarith p a vs;
    if is-float a ∧ is-float e then Some (power-aform-err p ((a, as), e) n)
    else None
  }
| approx-floatarith p (Powr a b) vs =
  do {
    ae1 ← approx-floatarith p a vs;
    ae2 ← approx-floatarith p b vs;
    powr-aform-err p ae1 ae2
  }

```

}

**lemma** *uminus-aform-uminus-aform[simp]*:  $uminus\text{-}aform\ (uminus\text{-}aform\ z) = (z::'a::real\text{-}vector\ aform)$

**by** (*auto intro!*: *prod-eqI pdevs-eqI simp: uminus-aform-def*)

**lemma** *degree-aform-inverse-aform'*:

$degree\text{-}aform\ X \leq n \implies degree\text{-}aform\ (fst\ (inverse\text{-}aform'\ p\ X)) \leq n$

**unfolding** *inverse-aform'-def*

**by** (*auto simp: Let-def trunc-bound-pdevs-def intro!: degree-pdev-upd-le degree-trunc-pdevs-le*)

**lemma** *degree-aform-inverse-aform*:

**assumes** *inverse-aform p X = Some Y*

**assumes**  $degree\text{-}aform\ X \leq n$

**shows**  $degree\text{-}aform\ (fst\ Y) \leq n$

**using** *assms*

**by** (*auto simp: inverse-aform-def Let-def degree-aform-inverse-aform' split: if-splits*)

**lemma** *degree-aform-ivl-err[simp]*:  $degree\text{-}aform\ (fst\ (ivl\text{-}err\ a)) = 0$

**by** (*auto simp: ivl-err-def*)

**lemma** *degree-aform-approx-bin*:

**assumes** *approx-bin p ivl X Y = Some Z*

**assumes**  $degree\text{-}aform\ (fst\ X) \leq m$

**assumes**  $degree\text{-}aform\ (fst\ Y) \leq m$

**shows**  $degree\text{-}aform\ (fst\ Z) \leq m$

**using** *assms*

**by** (*auto simp: approx-bin-def bind-eq-Some-conv Basis-list-real-def*

*intro!: order-trans[OF degree-aform-independent-from]*

*order-trans[OF degree-aform-of-ivl]*)

**lemma** *degree-aform-approx-un*:

**assumes** *approx-un p ivl X = Some Y*

**assumes**  $case\ X\ of\ None \Rightarrow True \mid Some\ X \Rightarrow degree\text{-}aform\ (fst\ X) \leq d1$

**shows**  $degree\text{-}aform\ (fst\ Y) \leq d1$

**using** *assms*

**by** (*auto simp: approx-un-def bind-eq-Some-conv Basis-list-real-def*

*intro!: order-trans[OF degree-aform-independent-from]*

*order-trans[OF degree-aform-of-ivl]*)

**lemma** *degree-aform-num-aform[simp]*:  $degree\text{-}aform\ (num\text{-}aform\ x) = 0$

**by** (*auto simp: num-aform-def*)

**lemma** *degree-max-aform*:

**assumes**  $degree\text{-}aform\text{-}err\ x \leq d$

**assumes**  $degree\text{-}aform\text{-}err\ y \leq d$

**shows**  $degree\text{-}aform\text{-}err\ (max\text{-}aform\text{-}err\ p\ x\ y) \leq d$

**using** *assms*

**by** (*auto simp: max-aform-err-def Let-def Basis-list-real-def split: prod.splits*)



*intro!*: *order-trans*[*OF degree-aform-independent-from*] *order-trans*[*OF degree-aform-of-ivl*])

**lemma** *degree-min-aform*:

**assumes** *degree-aform-err*  $x \leq d$

**assumes** *degree-aform-err*  $y \leq d$

**shows** *degree-aform-err*  $((\text{min-aform-err } p \ x \ y)) \leq d$

**using** *assms*

**by** (*auto simp: min-aform-err-def Let-def Basis-list-real-def split: prod.splits*

*intro!*: *order-trans*[*OF degree-aform-independent-from*] *order-trans*[*OF degree-aform-of-ivl*])

**lemma** *degree-aform-acc-err*:

*degree-aform*  $(\text{fst } (\text{acc-err } p \ X \ e)) \leq d$

**if** *degree-aform*  $(\text{fst } X) \leq d$

**using** *that* **by** (*auto simp: acc-err-def*)

**lemma** *degree-pdev-upd-degree*:

**assumes** *degree*  $b \leq \text{Suc } n$

**assumes** *degree*  $b \leq \text{Suc } (\text{degree-aform-err } X)$

**assumes** *degree-aform-err*  $X \leq n$

**shows** *degree*  $(\text{pdev-upd } b \ (\text{degree-aform-err } X) \ 0) \leq n$

**using** *assms*

**by** (*auto intro!: degree-le*)

**lemma** *degree-aform-err-inverse-aform-err*:

**assumes** *inverse-aform-err*  $p \ X = \text{Some } Y$

**assumes** *degree-aform-err*  $X \leq n$

**shows** *degree-aform-err*  $Y \leq n$

**using** *assms*

**apply** (*auto simp: inverse-aform-err-def bind-eq-Some-conv aform-to-aform-err-def*

*acc-err-def map-aform-err-def*

*aform-err-to-aform-def*

*intro!: degree-aform-acc-err*)

**apply** (*rule degree-pdev-upd-degree*)

**apply** (*auto dest!: degree-aform-inverse-aform*)

**apply** (*meson degree-pdev-upd-le nat-le-linear not-less-eq-eq order-trans*)

**apply** (*meson degree-pdev-upd-le nat-le-linear not-less-eq-eq order-trans*)

**done**

**lemma** *degree-aform-err-affine-unop*:

*degree-aform-err*  $(\text{affine-unop } p \ a \ b \ d \ X) \leq n$

**if** *degree-aform-err*  $X \leq n$

**using** *that*

**by** (*auto simp: affine-unop-def trunc-bound-pdevs-def degree-trunc-pdevs-le split: prod.splits*)

**lemma** *degree-aform-err-min-range-mono*:

**assumes** *min-range-mono*  $p \ F \ D \ l \ u \ X = \text{Some } Y$   
**assumes** *degree-aform-err*  $X \leq n$   
**shows** *degree-aform-err*  $Y \leq n$   
**using** *assms*  
**by** (*auto simp: min-range-mono-def bind-eq-Some-conv aform-to-aform-err-def*  
*acc-err-def map-aform-err-def mid-err-def range-reduce-aform-err-def*  
*aform-err-to-aform-def Let-def split: if-splits prod.splits*  
*intro!: degree-aform-err-affine-unop*)

**lemma** *degree-aform-err-min-range-antimono*:  
**assumes** *min-range-antimono*  $p \ F \ D \ l \ u \ X = \text{Some } Y$   
**assumes** *degree-aform-err*  $X \leq n$   
**shows** *degree-aform-err*  $Y \leq n$   
**using** *assms*  
**by** (*auto simp: min-range-antimono-def bind-eq-Some-conv aform-to-aform-err-def*  
*acc-err-def map-aform-err-def mid-err-def range-reduce-aform-err-def*  
*aform-err-to-aform-def Let-def split: if-splits prod.splits*  
*intro!: degree-aform-err-affine-unop*)

**lemma** *degree-aform-err-cos-aform-err*:  
**assumes** *cos-aform-err*  $p \ X = \text{Some } Y$   
**assumes** *degree-aform-err*  $X \leq n$   
**shows** *degree-aform-err*  $Y \leq n$   
**using** *assms*  
**apply** (*auto simp: cos-aform-err-def bind-eq-Some-conv aform-to-aform-err-def*  
*acc-err-def map-aform-err-def mid-err-def range-reduce-aform-err-def*  
*aform-err-to-aform-def Let-def split: if-splits prod.splits*  
*intro!: degree-aform-err-affine-unop*)  
**apply** (*metis degree-aform-err-add-aform' degree-aform-err-min-range-antimono*  
*degree-aform-ivl-err zero-le*)  
**apply** (*metis degree-aform-err-add-aform' degree-aform-err-min-range-mono de-*  
*gree-aform-ivl-err zero-le*)  
**apply** (*metis degree-aform-err-add-aform' degree-aform-err-min-range-mono de-*  
*gree-aform-ivl-err zero-le*)  
**apply** (*metis degree-aform-err-add-aform' degree-aform-err-min-range-antimono*  
*degree-aform-ivl-err zero-le*)  
**apply** (*metis degree-aform-err-add-aform' degree-aform-err-min-range-antimono*  
*degree-aform-ivl-err zero-le*)  
**done**

**lemma** *degree-aform-err-sqrt-aform-err*:  
**assumes** *sqrt-aform-err*  $p \ X = \text{Some } Y$   
**assumes** *degree-aform-err*  $X \leq n$   
**shows** *degree-aform-err*  $Y \leq n$   
**using** *assms*  
**apply** (*auto simp: sqrt-aform-err-def Let-def split: if-splits*)  
**apply** (*metis degree-aform-err-min-range-mono*)

**done**

**lemma** *degree-aform-err-arctan-aform-err*:  
 **assumes** *arctan-aform-err*  $p$   $X = \text{Some } Y$   
 **assumes** *degree-aform-err*  $X \leq n$   
 **shows** *degree-aform-err*  $Y \leq n$   
 **using** *assms*  
 **apply** (*auto simp: arctan-aform-err-def bind-eq-Some-conv*)  
 **apply** (*metis degree-aform-err-min-range-mono*)  
 **done**

**lemma** *degree-aform-err-exp-aform-err*:  
 **assumes** *exp-aform-err*  $p$   $X = \text{Some } Y$   
 **assumes** *degree-aform-err*  $X \leq n$   
 **shows** *degree-aform-err*  $Y \leq n$   
 **using** *assms*  
 **apply** (*auto simp: exp-aform-err-def bind-eq-Some-conv*)  
 **apply** (*metis degree-aform-err-min-range-mono*)  
 **done**

**lemma** *degree-aform-err-ln-aform-err*:  
 **assumes** *ln-aform-err*  $p$   $X = \text{Some } Y$   
 **assumes** *degree-aform-err*  $X \leq n$   
 **shows** *degree-aform-err*  $Y \leq n$   
 **using** *assms*  
 **apply** (*auto simp: ln-aform-err-def Let-def split: if-splits*)  
 **apply** (*metis degree-aform-err-add-aform' degree-aform-err-min-range-mono degree-aform-ivl-err zero-le*)  
 **done**

**lemma** *degree-aform-err-power-aform-err*:  
 **assumes** *degree-aform-err*  $X \leq n$   
 **shows** *degree-aform-err* (*power-aform-err*  $p$   $X$   $m$ )  $\leq n$   
 **using** *assms*  
 **by** (*auto simp: power-aform-err-def Let-def trunc-bound-pdevs-def degree-trunc-pdevs-le split: if-splits prod.splits*)

**lemma** *degree-aform-err-powr-aform-err*:  
 **assumes** *powr-aform-err*  $p$   $X$   $Z = \text{Some } Y$   
 **assumes** *degree-aform-err*  $X \leq n$   
 **assumes** *degree-aform-err*  $Z \leq n$   
 **shows** *degree-aform-err*  $Y \leq n$   
 **using** *assms*  
 **apply** (*auto simp: powr-aform-err-def bind-eq-Some-conv degree-aform-mult-aform' dest!: degree-aform-err-ln-aform-err degree-aform-err-exp-aform-err split: if-splits*)  
 **apply** (*metis degree-aform-mult-aform' fst-conv order-trans snd-conv*)  
 **apply** (*rule degree-aform-approx-bin, assumption*)  
 **apply** *auto*

```

done

lemma approx-floatarith-degree:
  assumes approx-floatarith p ra VS = Some X
  assumes  $\bigwedge V. V \in \text{set } VS \implies \text{degree-aform-err } V \leq d$ 
  shows  $\text{degree-aform-err } X \leq d$ 
  using assms
proof (induction ra arbitrary: X)
  case (Add ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: degree-aform-err-add-aform' degree-aform-acc-err)
  next
  case (Minus ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv)
  next
  case (Mult ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: degree-aform-mult-aform' degree-aform-acc-err)
  next
  case (Inverse ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-inverse-aform-err)
  next
  case (Cos ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-cos-aform-err)
  next
  case (Arctan ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-arctan-aform-err)
  next
  case (Abs ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv Let-def Basis-list-real-def
      intro!: order-trans[OF degree-aform-independent-from] order-trans[OF de-
gree-aform-of-ivl]
      degree-aform-acc-err
      split: if-splits)
  next
  case (Max ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: degree-max-aform degree-aform-acc-err)
  next
  case (Min ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: degree-min-aform degree-aform-acc-err)
  next
  case Pi

```

```

then show ?case
  by (auto simp: bind-eq-Some-conv Let-def Basis-list-real-def
        intro!: order-trans[OF degree-aform-independent-from] order-trans[OF de-
gree-aform-of-ivl]
        degree-aform-acc-err
        split: if-splits)
next
  case (Sqrt ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-sqrt-aform-err)
next
  case (Exp ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-exp-aform-err)
next
  case (Powr ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-powr-aform-err)
next
  case (Ln ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro: degree-aform-err-ln-aform-err)
next
  case (Power ra x2a)
  then show ?case
    by (auto intro!: degree-aform-err-power-aform-err simp: bind-eq-Some-conv split:
if-splits)
next
  case (Floor ra)
  then show ?case
    apply -
    by (rule degree-aform-approx-un) (auto split: option.splits)
next
  case (Var x)
  then show ?case
    by (auto simp: max-def split: if-splits)
    (use Var.premis(2) nat-le-linear nth-mem order-trans in blast)+
next
  case (Num x)
  then show ?case by auto
qed

```

**definition** *affine-extension2* **where**

$$\begin{aligned}
& \text{affine-extension2 fcn} \text{t} \text{n-aff fcn} \text{t} \text{n} \longleftrightarrow ( \\
& \quad \forall d \ a1 \ a2 \ X \ e2. \\
& \quad \text{fcn} \text{t} \text{n-aff } d \ a1 \ a2 = \text{Some } X \longrightarrow \\
& \quad e2 \in \text{UNIV} \rightarrow \{-1..1\} \longrightarrow \\
& \quad d \geq \text{degree-aform } a1 \longrightarrow \\
& \quad d \geq \text{degree-aform } a2 \longrightarrow
\end{aligned}$$

$(\exists e3 \in UNIV \rightarrow \{-1..1\}.$   
 $(fctn (aform-val e2 a1) (aform-val e2 a2) = aform-val e3 X \wedge$   
 $(\forall n. n < d \longrightarrow e3 n = e2 n) \wedge$   
 $aform-val e2 a1 = aform-val e3 a1 \wedge aform-val e2 a2 = aform-val e3 a2)))$

**lemma** *affine-extension2E*:

**assumes** *affine-extension2 fctn-aff fctn*  
**assumes** *fctn-aff d a1 a2 = Some X*  
 $e \in UNIV \rightarrow \{-1..1\}$   
 $d \geq \text{degree-aform } a1$   
 $d \geq \text{degree-aform } a2$   
**obtains**  $e'$  **where**  $e' \in UNIV \rightarrow \{-1..1\}$   
 $fctn (aform-val e a1) (aform-val e a2) = aform-val e' X$   
 $\bigwedge n. n < d \implies e' n = e n$   
 $aform-val e a1 = aform-val e' a1$   
 $aform-val e a2 = aform-val e' a2$   
**using** *assms*  
**unfolding** *affine-extension2-def*  
**by** *metis*

**lemma** *aform-err-uminus-aform*:

$-x \in \text{aform-err } e (\text{uminus-aform } X, ba)$   
**if**  $e \in UNIV \rightarrow \{-1..1\}$   $x \in \text{aform-err } e (X, ba)$   
**using** *that* **by** *(auto simp: aform-err-def)*

**definition** *aforms-err*  $e (xs::\text{aform-err list}) = \text{listset } (\text{map } (\text{aform-err } e) xs)$

**lemma** *aforms-err-Nil[simp]*:  $\text{aforms-err } e [] = \{\}\}$

**and** *aforms-err-Cons*:  $\text{aforms-err } e (x\#xs) = \text{set-Cons } (\text{aform-err } e x) (\text{aforms-err } e xs)$   
**by** *(auto simp: aforms-err-def)*

**lemma** *in-set-ConsI*:  $a\#b \in \text{set-Cons } A B$

**if**  $a \in A$  **and**  $b \in B$   
**using** *that*  
**by** *(auto simp: set-Cons-def)*

**lemma** *mem-aforms-err-Cons-iff[simp]*:  $x\#xs \in \text{aforms-err } e (X\#XS) \iff x \in \text{aform-err } e X \wedge xs \in \text{aforms-err } e XS$

**by** *(auto simp: aforms-err-Cons set-Cons-def)*

**lemma** *mem-aforms-err-Cons-iff-Ex-conv*:  $x \in \text{aforms-err } e (X\#XS) \iff (\exists y ys. x = y\#ys \wedge y \in \text{aform-err } e X \wedge ys \in \text{aforms-err } e XS)$

**by** *(auto simp: aforms-err-Cons set-Cons-def)*

**lemma** *listset-Cons-mem-conv*:

$a \# vs \in \text{listset } AVS \iff (\exists A VS. AVS = A \# VS \wedge a \in A \wedge vs \in \text{listset } VS)$   
**by** *(induction AVS) (auto simp: set-Cons-def)*

**lemma** *listset-Nil-mem-conv[simp]*:

$\square \in \text{listset } AVS \longleftrightarrow AVS = \square$

**by** (*induction AVS*) (*auto simp: set-Cons-def*)

**lemma** *listset-nthD*:  $vs \in \text{listset } VS \implies i < \text{length } vs \implies vs ! i \in VS ! i$

**by** (*induction vs arbitrary: VS i*)

(*auto simp: nth-Cons listset-Cons-mem-conv split: nat.splits*)

**lemma** *length-listsetD*:

$vs \in \text{listset } VS \implies \text{length } vs = \text{length } VS$

**by** (*induction vs arbitrary: VS*) (*auto simp: listset-Cons-mem-conv*)

**lemma** *length-aforms-errD*:

$vs \in \text{aforms-err } e \text{ } VS \implies \text{length } vs = \text{length } VS$

**by** (*auto simp: aforms-err-def length-listsetD*)

**lemma** *nth-aforms-errI*:

$vs ! i \in \text{aform-err } e \text{ } (VS ! i)$

**if**  $vs \in \text{aforms-err } e \text{ } VS \ i < \text{length } vs$

**using** *that*

**unfolding** *aforms-err-def*

**apply**  $-$

**apply** (*frule listset-nthD, assumption*)

**by** (*auto simp: aforms-err-def length-listsetD*)

**lemma** *eucl-truncate-down-float[simp]*:  $\text{eucl-truncate-down } p \ x \in \text{float}$

**by** (*auto simp: eucl-truncate-down-def*)

**lemma** *eucl-truncate-up-float[simp]*:  $\text{eucl-truncate-up } p \ x \in \text{float}$

**by** (*auto simp: eucl-truncate-up-def*)

**lemma** *trunc-bound-eucl-float[simp]*:  $\text{fst } (\text{trunc-bound-eucl } p \ x) \in \text{float}$

$\text{snd } (\text{trunc-bound-eucl } p \ x) \in \text{float}$

**by** (*auto simp: trunc-bound-eucl-def Let-def*)

**lemma** *add-aform'-float*:

$\text{add-aform}' \ p \ x \ y = ((a, b), ba) \implies a \in \text{float}$

$\text{add-aform}' \ p \ x \ y = ((a, b), ba) \implies ba \in \text{float}$

**by** (*auto simp: add-aform'-def Let-def*)

**lemma** *uminus-aform-float*:  $\text{uminus-aform } (aa, bb) = (a, b) \implies aa \in \text{float} \implies a \in \text{float}$

**by** (*auto simp: uminus-aform-def*)

**lemma** *mult-aform'-float*:  $\text{mult-aform}' \ p \ x \ y = ((a, b), ba) \implies a \in \text{float}$

$\text{mult-aform}' \ p \ x \ y = ((a, b), ba) \implies ba \in \text{float}$

**by** (*auto simp: mult-aform'-def Let-def split-beta'*)

**lemma** *inverse-aform'-float*:  $\text{inverse-aform}' \ p \ x = ((a, bb), baa) \implies a \in \text{float}$

**using**  $[[\text{linarith-split-limit}=256]]$   
**by** (*auto simp: inverse-aform'-def Let-def*)

**lemma** *inverse-aform-float:*

*inverse-aform p x = Some ((a, bb), baa)  $\implies$  a  $\in$  float*

**by** (*auto simp: inverse-aform-def Let-def apfst-def map-prod-def uminus-aform-def  
inverse-aform'-float  
split: if-splits prod.splits*)

**lemma** *inverse-aform-err-float: inverse-aform-err p x = Some ((a, b), ba)  $\implies$  a  $\in$  float*

*inverse-aform-err p x = Some ((a, b), ba)  $\implies$  ba  $\in$  float*

**by** (*auto simp: inverse-aform-err-def map-aform-err-def acc-err-def bind-eq-Some-conv  
aform-err-to-aform-def aform-to-aform-err-def inverse-aform-float*)

**lemma** *affine-unop-float:*

*affine-unop p asdf aaa bba h = ((a, b), ba)  $\implies$  a  $\in$  float*

*affine-unop p asdf aaa bba h = ((a, b), ba)  $\implies$  ba  $\in$  float*

**by** (*auto simp: affine-unop-def trunc-bound-eucl-def Let-def split: prod.splits*)

**lemma** *min-range-antimono-float:*

*min-range-antimono p f f' i g h = Some ((a, b), ba)  $\implies$  a  $\in$  float*

*min-range-antimono p f f' i g h = Some ((a, b), ba)  $\implies$  ba  $\in$  float*

**by** (*auto simp: min-range-antimono-def Let-def bind-eq-Some-conv mid-err-def  
affine-unop-float split: prod.splits*)

**lemma** *min-range-mono-float:*

*min-range-mono p f f' i g h = Some ((a, b), ba)  $\implies$  a  $\in$  float*

*min-range-mono p f f' i g h = Some ((a, b), ba)  $\implies$  ba  $\in$  float*

**by** (*auto simp: min-range-mono-def Let-def bind-eq-Some-conv mid-err-def  
affine-unop-float split: prod.splits*)

**lemma** *in-float-timesI: a  $\in$  float if b = a \* 2 b  $\in$  float*

**proof** –

**from** *that have a = b / 2 by simp*

**also have**  $\dots \in$  float **using** *that(2) by auto*

**finally show** *?thesis .*

**qed**

**lemma** *interval-extension-floor: interval-extension1 ( $\lambda$ ivl. Some (floor-float-interval ivl)) floor*

**by** (*auto simp: interval-extension1-def floor-float-intervalI*)

**lemma** *approx-floatarith-Elem:*

**assumes** *approx-floatarith p ra VS = Some X*

**assumes** *e: e  $\in$  UNIV  $\rightarrow$   $\{-1 .. 1\}$*

**assumes** *vs  $\in$  aforms-err e VS*

**shows** *interpret-floatarith ra vs  $\in$  aform-err e X*

**using** *assms(1)*



```

proof (induction ra arbitrary: X)
  case (Add ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: add-aform'[OF e])
next
  case (Minus ra)
  then show ?case
    by (auto intro!: aform-err-uminus-aform[OF e])
next
  case (Mult ra1 ra2)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: mult-aform'E[OF e])
next
  case (Inverse ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: inverse-aform-err[OF e])
next
  case (Cos ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: cos-aform-err[OF - - e])
next
  case (Arctan ra)
  then show ?case
    by (auto simp: bind-eq-Some-conv intro!: arctan-aform-err[OF - - e])
next
  case (Abs fa)
  from Abs.prem
  obtain a where a: approx-floatarith p fa VS = Some a
    by (auto simp add: Let-def bind-eq-Some-conv)
  from Abs.IH[OF a]
  have mem: interpret-floatarith fa vs ∈ aform-err e a
    by auto
  then have mem': -interpret-floatarith fa vs ∈ aform-err e (apfst uminus-aform
a)
    by (auto simp: aform-err-def)

let ?i = lower (ivl-of-aform-err p a)
let ?s = upper (ivl-of-aform-err p a)
consider ?i > 0 | ?i ≤ 0 ?s < 0 | ?i ≤ 0 ?s ≥ 0
  by arith
then show ?case
proof cases
  case hyps: 1
  then show ?thesis
    using Abs.prem mem ivl-of-aform-err[OF e mem, of p]
    by (auto simp: a set-of-eq)
next
  case hyps: 2
  then show ?thesis

```

```

    using Abs.prems mem ivl-of-aform-err[OF e mem, of p]
           ivl-of-aform-err[OF e mem', of p]
  by (cases a) (auto simp: a abs-real-def set-of-eq intro!: aform-err-uminus-aform[OF
e])
next
  case hyps: 3
  then show ?thesis
    using Abs.prems mem ivl-of-aform-err[OF e mem, of p]
    by (auto simp: a abs-real-def max-def Let-def set-of-eq)
qed
next
  case (Max ra1 ra2)
  from Max.prems
  obtain a b where a: approx-floatarith p ra1 VS = Some a
    and b: approx-floatarith p ra2 VS = Some b
    by (auto simp add: Let-def bind-eq-Some-conv)
  from Max.IH(1)[OF a] Max.IH(2)[OF b]
  have mem: interpret-floatarith ra1 vs ∈ aform-err e a
    interpret-floatarith ra2 vs ∈ aform-err e b
    by auto
  let ?ia = lower (ivl-of-aform-err p a)
  let ?sa = upper (ivl-of-aform-err p a)
  let ?ib = lower (ivl-of-aform-err p b)
  let ?sb = upper (ivl-of-aform-err p b)
  consider ?sa < ?ib | ?sa ≥ ?ib ?sb < ?ia | ?sa ≥ ?ib ?sb ≥ ?ia
    by arith
  then show ?case
    using Max.prems mem ivl-of-aform-err[OF e mem(1), of p] ivl-of-aform-err[OF
e mem(2), of p]
    by cases (auto simp: a b max-def max-aform-err-def set-of-eq)
next
  case (Min ra1 ra2)
  from Min.prems
  obtain a b where a: approx-floatarith p ra1 VS = Some a
    and b: approx-floatarith p ra2 VS = Some b
    by (auto simp add: Let-def bind-eq-Some-conv)
  from Min.IH(1)[OF a] Min.IH(2)[OF b]
  have mem: interpret-floatarith ra1 vs ∈ aform-err e a
    interpret-floatarith ra2 vs ∈ aform-err e b
    by auto
  let ?ia = lower (ivl-of-aform-err p a)
  let ?sa = upper (ivl-of-aform-err p a)
  let ?ib = lower (ivl-of-aform-err p b)
  let ?sb = upper (ivl-of-aform-err p b)
  consider ?sa < ?ib | ?sa ≥ ?ib ?sb < ?ia | ?sa ≥ ?ib ?sb ≥ ?ia
    by arith
  then show ?case
    using Min.prems mem ivl-of-aform-err[OF e mem(1), of p] ivl-of-aform-err[OF
e mem(2), of p]

```

```

    by cases (auto simp: a b min-def min-aform-err-def set-of-eq)
next
case Pi
then show ?case using pi-float-interval
  by auto
next
case (Sqrt ra)
then show ?case
  by (auto simp: bind-eq-Some-conv intro!: sqrt-aform-err[OF - - e])
next
case (Exp ra)
then show ?case
  by (auto simp: bind-eq-Some-conv intro!: exp-aform-err[OF - - e])
next
case (Powr ra1 ra2)
then show ?case
  by (auto simp: bind-eq-Some-conv intro!: powr-aform-err[OF - - e])
next
case (Ln ra)
then show ?case
  by (auto simp: bind-eq-Some-conv intro!: ln-aform-err[OF - - e])
next
case (Power ra x2a)
then show ?case
  by (auto simp: bind-eq-Some-conv is-float-def
    intro!: power-aform-err[OF - - - e] split: if-splits)
next
case (Floor ra)
then show ?case
  by (auto simp: bind-eq-Some-conv intro!: approx-unE[OF interval-extension-floor
e]
    split: option.splits)
next
case (Var x)
then show ?case
  using assms(3)
  apply -
  apply (frule length-aforms-errD)
  by (auto split: if-splits simp: aform-err-def dest!: nth-aforms-errI[where i=x])
next
case (Num x)
then show ?case
  by (auto split: if-splits simp: aform-err-def num-aform-def aform-val-def)
qed

primrec approx-floatariths-aformerr ::
  nat  $\Rightarrow$  floatarith list  $\Rightarrow$  aform-err list  $\Rightarrow$  aform-err list option
where
  approx-floatariths-aformerr - [] - = Some []

```

```

| approx-floatariths-aformerr p (a#bs) vs =
  do {
    a ← approx-floatarith p a vs;
    r ← approx-floatariths-aformerr p bs vs;
    Some (a#r)
  }

```

**lemma** *approx-floatariths-Elem*:  
**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$   
**assumes** *approx-floatariths-aformerr p ra VS = Some X*  
**assumes**  $vs \in aforms-err\ e\ VS$   
**shows** *interpret-floatariths ra vs  $\in$  aforms-err e X*  
**using** *assms(2)*  
**proof** (*induction ra arbitrary: X*)  
**case Nil then show ?case by simp**  
**next**  
**case (Cons ra ras)**  
**from Cons.prem**  
**obtain a r where a: approx-floatarith p ra VS = Some a**  
**and r: approx-floatariths-aformerr p ras VS = Some r**  
**and X: X = a # r**  
**by (auto simp: bind-eq-Some-conv)**  
**then show ?case**  
**using assms(1)**  
**by (auto simp: X Cons.IH intro!: approx-floatarith-Elem assms)**  
**qed**

**lemma** *fold-max-mono*:  
**fixes**  $x::'a::linorder$   
**shows**  $x \leq y \implies fold\ max\ zs\ x \leq fold\ max\ zs\ y$   
**by (induct zs arbitrary: x y) (auto intro!: Cons simp: max-def)**

**lemma** *fold-max-le-self*:  
**fixes**  $y::'a::linorder$   
**shows**  $y \leq fold\ max\ ys\ y$   
**by (induct ys) (auto intro: order-trans[OF - fold-max-mono])**

**lemma** *fold-max-le*:  
**fixes**  $x::'a::linorder$   
**shows**  $x \in set\ xs \implies x \leq fold\ max\ xs\ x$   
**by (induct xs arbitrary: x z) (auto intro: order-trans[OF - fold-max-le-self])**

**abbreviation** *degree-aforms-err*  $\equiv$  *degrees o map (snd o fst)*

**definition** *aforms-err-to-aforms d xs =*  
*(map ( $\lambda(d, x). aform-err-to-aform\ x\ d$ ) (zip [d.. $d + length\ xs$ ] xs))*

**lemma** *aform-vals-empty[simp]: aform-vals e' [] = []*

**by** (*auto simp: aform-vals-def*)  
**lemma** *aforms-err-to-aforms-Nil*[*simp*]: (*aforms-err-to-aforms*  $n$  []) = []  
**by** (*auto simp: aforms-err-to-aforms-def*)

**lemma** *aforms-err-to-aforms-Cons*[*simp*]:  
*aforms-err-to-aforms*  $n$  ( $X \# XS$ ) = *aform-err-to-aform*  $X$   $n$  # *aforms-err-to-aforms*  
(*Suc*  $n$ )  $XS$   
**by** (*auto simp: aforms-err-to-aforms-def not-le nth-append nth-Cons*  
*intro!: nth-equalityI split: nat.splits*)

**lemma** *degree-aform-err-to-aform-le*:  
*degree-aform* (*aform-err-to-aform*  $X$   $n$ )  $\leq$  *max* (*degree-aform-err*  $X$ ) (*Suc*  $n$ )  
**by** (*auto simp: aform-err-to-aform-def intro!: degree-le*)

**lemma** *less-degree-aform-aform-err-to-aformD*:  $i <$  *degree-aform* (*aform-err-to-aform*  
 $X$   $n$ )  $\implies$   $i <$  *max* (*Suc*  $n$ ) (*degree-aform-err*  $X$ )  
**using** *degree-aform-err-to-aform-le*[*of*  $X$   $n$ ] **by** *auto*

**lemma** *pdevs-domain-aform-err-to-aform*:  
*pdevs-domain* (*snd* (*aform-err-to-aform*  $X$   $n$ )) = *pdevs-domain* (*snd* (*fst*  $X$ ))  $\cup$   
(*if* *snd*  $X$  = 0 *then* {} *else* { $n$ })  
**if**  $n \geq$  *degree-aform-err*  $X$   
**using** *that*  
**by** (*auto simp: aform-err-to-aform-def split: if-splits*)

**lemma** *length-aforms-err-to-aforms*[*simp*]: *length* (*aforms-err-to-aforms*  $i$   $XS$ ) =  
*length*  $XS$   
**by** (*auto simp: aforms-err-to-aforms-def*)

**lemma** *aforms-err-to-aforms-ex*:  
**assumes**  $X: x \in$  *aforms-err*  $e$   $X$   
**assumes** *deg*: *degree-aforms-err*  $X \leq n$   
**assumes**  $e: e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows**  $\exists e' \in UNIV \rightarrow \{-1 .. 1\}. x =$  *aform-vals*  $e'$  (*aforms-err-to-aforms*  $n$   $X$ )  
 $\wedge$   
 $(\forall i < n. e' i = e i)$   
**using**  $X$  *deg*  
**proof** (*induction*  $X$  *arbitrary*:  $x$   $n$ )  
**case** *Nil* **then show** ?*case* **using**  $e$   
**by** (*auto simp: o-def degrees-def intro!: bexI*[**where**  $x = \lambda i. e i$ ])  
**next**  
**case** (*Cons*  $X$   $XS$ )  
**from** *Cons.prem*s **obtain**  $y$   $ys$  **where**  $ys$ :  
*degree-aform-err*  $X \leq n$   
*degree-aforms-err*  $XS \leq n$   
 $x = y \# ys$   $y \in$  *aform-err*  $e$   $X$   $ys \in$  *aforms-err*  $e$   $XS$   
**by** (*auto simp: mem-aforms-err-Cons-iff-Ex-conv degrees-def*)  
**then have** *degree-aforms-err*  $XS \leq$  *Suc*  $n$  **by** *auto*  
**from** *Cons.IH*[*OF*  $ys$ (5) *this*]

```

obtain  $e'$  where  $e': e' \in UNIV \rightarrow \{-1..1\}$   $ys = \text{aform-vals } e' (\text{aforms-err-to-aforms } (Suc\ n)\ XS)$ 
  ( $\forall i < n. e' i = e i$ )
  by auto
from  $\text{aform-err-to-aformE}[OF\ ys(4,1)]$  obtain  $err$  where  $err:$ 
   $y = \text{aform-val } (e(n := err)) (\text{aform-err-to-aform } X\ n) - 1 \leq err \leq 1$ 
  by auto
show ?case
proof (safe intro!:  $\text{beXI}[\text{where } x=e'(n:=err)]$ , goal-cases)
  case 1
  then show ?case
    unfolding  $ys\ e'\ err$ 
    apply (auto simp: aform-vals-def aform-val-def simp del: pdevs-val-upd)
    apply (rule pdevs-val-degree-cong)
    apply simp
  subgoal
    using  $ys\ e'$ 
    by (auto dest!: less-degree-aform-aform-err-to-aformD simp: max-def split: if-splits)
  subgoal premises  $prems$  for  $a\ b$ 
  proof -
    have  $\text{pdevs-val } (\lambda a. \text{if } a = n \text{ then } err \text{ else } e' a)\ b = \text{pdevs-val } (e'(n:=err))\ b$ 
      unfolding fun-upd-def by simp
    also have  $\dots = \text{pdevs-val } e'\ b - e'\ n * \text{pdevs-apply } b\ n + err * \text{pdevs-apply } b\ n$ 
      by simp
    also
      from  $prems$ 
      obtain  $i$  where  $i: \text{aforms-err-to-aforms } (Suc\ n)\ XS ! i = (a, b)$ 
         $i < \text{length } (\text{aforms-err-to-aforms } (Suc\ n)\ XS)$ 
        by (auto simp: in-set-conv-nth)
      { note  $i(1)[\text{symmetric}]$ 
        also have  $\text{aforms-err-to-aforms } (Suc\ n)\ XS ! i = \text{aform-err-to-aform } (XS ! i)\ (Suc\ n + i)$ 
          unfolding aforms-err-to-aforms-def
          using  $i$ 
          by (simp del: upt-Suc)
          finally have  $b = \text{snd } (\text{aform-err-to-aform } (XS ! i)\ (Suc\ n + i))$  by (auto simp: prod-eq-iff)
        } note  $b = \text{this}$ 
      have  $\text{degree-aform-err } (XS ! i) \leq n$ 
        using  $ys(2)\ i$  by (auto simp: degrees-def)
      then have  $n \notin \text{pdevs-domain } b$  unfolding  $b$ 
        apply (subst pdevs-domain-aform-err-to-aform)
        by (auto intro!: degree)
      then have  $\text{pdevs-apply } b\ n = 0$  by simp
      finally
        show ?thesis by simp
    qed

```

```

done
next
case (2 i)
then show ?case
using e' by auto
next
case (3 i)
then show ?case
using e' err
by auto
qed
qed

```

**lemma** *aforms-err-to-aformsE*:

```

assumes X: x ∈ aforms-err e X
assumes deg: degree-aforms-err X ≤ n
assumes e: e ∈ UNIV → {-1 .. 1}
obtains e' where x = aform-vals e' (aforms-err-to-aforms n X) e' ∈ UNIV →
{-1 .. 1}
  ∧ i. i < n ⇒ e' i = e i
using aforms-err-to-aforms-ex[OF X deg e]
by blast

```

**definition** *approx-floatariths p ea as =*

```

do {
  let da = (degree-aforms as);
  let aes = (map (λx. (x, 0)) as);
  rs ← approx-floatariths-aformerr p ea aes;
  let d = max da (degree-aforms-err (rs));
  Some (aforms-err-to-aforms d rs)
}

```

**lemma** *listset-sings[simp]*:

```

listset (map (λx. {f x}) as) = {map f as}
by (induction as) (auto simp: set-Cons-def)

```

**lemma** *approx-floatariths-outer*:

```

assumes approx-floatariths p ea as = Some XS
assumes vs ∈ Joints as
shows (interpret-floatariths ea vs @ vs) ∈ Joints (XS @ as)

```

**proof** –

```

from assms obtain da aes rs d where
  da: da = degree-aforms as
  and aes: aes = (map (λx. (x, 0)) as)
  and rs: approx-floatariths-aformerr p ea aes = Some rs
  and d: d = max da (degree-aforms-err (rs))
  and XS: aforms-err-to-aforms d rs = XS
by (auto simp: approx-floatariths-def Let-def bind-eq-Some-conv)
have abbd: (a, b) ∈ set as ⇒ degree b ≤ degree-aforms as for a b

```

**apply** (rule *degrees-leD*[*OF order-refl*]) **by force**  
**from** *da d* **have** *i-less*:  $(a, b) \in \text{set } as \implies i < \text{degree } b \implies i < \min d$  **da for** *i*  
*a b*  
**by** (*auto dest!*: *abbd*)

**have** *abbd*:  $(a, b) \in \text{set } as \implies \text{degree } b \leq \text{degree-aforms } as$  **for** *a b*  
**apply** (rule *degrees-leD*[*OF order-refl*]) **by force**  
**from** *assms* **obtain** *e'* **where** *vs*:  $vs = (\text{map } (\text{aform-val } e') as)$  **and** *e'*:  $e' \in UNIV \rightarrow \{-1 .. 1\}$   
**by** (*auto simp*: *Joints-def valuate-def*)  
**note** *vs*  
**also**  
**have** *vs-aes*:  $vs \in \text{aforms-err } e' \text{ aes}$   
**unfolding** *aes*  
**by** (*auto simp*: *vs aforms-err-def o-def aform-err-def*)  
**from** *approx-floatariths-Elem*[*OF e' rs this*]  
**have** *iars*:  $\text{interpret-floatariths } ea (\text{map } (\text{aform-val } e') as) \in \text{aforms-err } e' \text{ rs}$   
**by** (*auto simp*: *vs*)  
**have** *degree-aforms-err rs*  $\leq d$   
**by** (*auto simp*: *d da*)  
**from** *aforms-err-to-aformsE*[*OF iars this e'*] **obtain** *e* **where**  
 $\text{interpret-floatariths } ea (\text{map } (\text{aform-val } e') as) = \text{aform-vals } e \text{ XS}$   
**and** *e*:  $e \in UNIV \rightarrow \{-1..1\} \wedge i. i < d \implies e i = e' i$   
**by** (*auto simp*: *XS*)  
**note** *this (1)*  
**finally have**  $\text{interpret-floatariths } ea \text{ vs} = \text{aform-vals } e \text{ XS}$  .

**moreover**

**from** *e* **have** *e'-eq*:  $e' i = e i$  **if**  $i < \min d$  **da for** *i*  
**using** *that*  
**by** (*auto simp*: *min-def split: if-splits*)  
**then have**  $vs = \text{aform-vals } e \text{ as}$   
**by** (*auto simp*: *vs aform-vals-def aform-val-def intro!*: *pdevs-val-degree-cong e'-eq i-less*)

**ultimately show** *?thesis*  
**using** *e(1)*  
**by** (*auto simp*: *Joints-def valuate-def aform-vals-def intro!*: *image-eqI*[**where**  $x=e$ ])  
**qed**

**lemma** *length-eq-NilI*:  $\text{length } [] = \text{length } []$   
**and** *length-eq-ConsI*:  $\text{length } xs = \text{length } ys \implies \text{length } (x\#xs) = \text{length } (y\#ys)$   
**by** *auto*

## 5.7 Generic operations on Affine Forms in Euclidean Space

**lemma** *pdevs-val-domain-cong*:



**assumes**  $b = d$   
**assumes**  $\bigwedge i. i \in \text{pdevs-domain } b \implies a \ i = c \ i$   
**shows**  $\text{pdevs-val } a \ b = \text{pdevs-val } c \ d$   
**using** *assms*  
**by** (*auto simp: pdevs-val-pdevs-domain*)

**lemma** *fresh-JointsI*:

**assumes**  $xs \in \text{Joints } XS$   
**assumes**  $\text{list-all } (\lambda Y. \text{pdevs-domain } (\text{snd } X) \cap \text{pdevs-domain } (\text{snd } Y) = \{\}) \ XS$   
**assumes**  $x \in \text{Affine } X$   
**shows**  $x \# xs \in \text{Joints } (X \# XS)$   
**using** *assms*  
**unfolding** *Joints-def Affine-def valuate-def*

**proof** *safe*

**fix**  $e \ e'::\text{nat} \Rightarrow \text{real}$   
**assume**  $H: \text{list-all } (\lambda Y. \text{pdevs-domain } (\text{snd } X) \cap \text{pdevs-domain } (\text{snd } Y) = \{\}) \ XS$   
 $XS$   
 $e \in \text{UNIV} \rightarrow \{-1..1\}$   
 $e' \in \text{UNIV} \rightarrow \{-1..1\}$   
**have**  $\bigwedge a \ b \ i. \forall Y \in \text{set } XS. \text{pdevs-domain } (\text{snd } X) \cap \text{pdevs-domain } (\text{snd } Y) = \{\}$   
 $\implies$

$\text{pdevs-apply } b \ i \neq 0 \implies$   
 $\text{pdevs-apply } (\text{snd } X) \ i \neq 0 \implies$   
 $(a, b) \notin \text{set } XS$

**by** (*metis (poly-guards-query) IntI all-not-in-conv in-pdevs-domain snd-eqD*)

**with**  $H$  **show**

$\text{aform-val } e' \ X \ \# \ \text{map } (\text{aform-val } e) \ XS \in (\lambda e. \text{map } (\text{aform-val } e) \ (X \ \# \ XS))$   
 $' \ (\text{UNIV} \rightarrow \{-1..1\})$

**by** (*intro image-eqI[where  $x = \lambda i. \text{if } i \in \text{pdevs-domain } (\text{snd } X) \text{ then } e' \ i \ \text{else } e \ i]$* )

(*auto simp: aform-val-def list-all-iff Pi-iff intro!: pdevs-val-domain-cong*)

**qed**

**primrec** *approx-slp::nat*  $\Rightarrow \text{slp} \Rightarrow \text{aform-err list} \Rightarrow \text{aform-err list option}$

**where**

$\text{approx-slp } p \ [] \ xs = \text{Some } xs$   
 $|\ \text{approx-slp } p \ (ea \ \# \ eas) \ xs =$   
 $\text{do } \{$   
 $\quad r \leftarrow \text{approx-floatarith } p \ ea \ xs;$   
 $\quad \text{approx-slp } p \ eas \ (r \ \# \ xs)$   
 $\}$

**lemma** *Nil-mem-Joints[*intro, simp*]*:  $[] \in \text{Joints } []$

**by** (*force simp: Joints-def valuate-def*)

**lemma** *map-nth-Joints*:  $xs \in \text{Joints } XS \implies (\bigwedge i. i \in \text{set } xs \implies i < \text{length } XS)$

$\implies \text{map } (\text{nth } xs) \ is \ @ \ xs \in \text{Joints } (\text{map } (\text{nth } XS) \ is \ @ \ XS)$

**by** (*auto simp: Joints-def valuate-def*)

**lemma** *map-nth-Joints'*:  $xs \in \text{Joints } XS \implies (\bigwedge i. i \in \text{set } is \implies i < \text{length } XS)$   
 $\implies \text{map } (\text{nth } xs) \text{ is} \in \text{Joints } (\text{map } (\text{nth } XS) \text{ is})$   
**by** (*rule Joints-appendD2[OF map-nth-Joints]*) *auto*

**lemma** *approx-slp-Elem*:

**assumes**  $e: e \in \text{UNIV} \rightarrow \{-1 .. 1\}$

**assumes**  $vs \in \text{aforms-err } e \text{ VS}$

**assumes**  $\text{approx-slp } p \text{ ra } VS = \text{Some } X$

**shows**  $\text{interpret-slp } \text{ra } vs \in \text{aforms-err } e \text{ X}$

**using** *assms(2-)*

**proof** (*induction ra arbitrary: X vs VS*)

**case** (*Cons ra ras*)

**from** *Cons.prem*s

**obtain**  $a$  **where**  $a: \text{approx-floatarith } p \text{ ra } VS = \text{Some } a$

**and**  $r: \text{approx-slp } p \text{ ras } (a \# VS) = \text{Some } X$

**by** (*auto simp: bind-eq-Some-conv*)

**from** *approx-floatarith-Elem[OF a e Cons.prem*s(1)]

**have**  $\text{interpret-floatarith } \text{ra } vs \in \text{aform-err } e \text{ a}$

**by** *auto*

**then have**  $1: \text{interpret-floatarith } \text{ra } vs \# vs \in \text{aforms-err } e \text{ (a \# VS)}$

**unfolding** *mem-aforms-err-Cons-iff*

**using** *Cons.prem*s(1)

**by** *auto*

**show** *?case*

**by** (*auto intro!: Cons.IH 1 r*)

**qed** *auto*

**definition** *approx-slp-outer*  $p \ n \ \text{slp } XS =$

*do* {

*let*  $d = \text{degree-aforms } XS;$

*let*  $XSe = (\text{map } (\lambda x. (x, 0)) XS);$

$rs \leftarrow \text{approx-slp } p \ \text{slp } XSe;$

*let*  $rs' = \text{take } n \ rs;$

*let*  $d' = \max d \ (\text{degree-aforms-err } rs');$

*Some* (*aforms-err-to-aforms*  $d' \ rs'$ )

}

**lemma** *take-in-listsetI*:  $xs \in \text{listset } XS \implies \text{take } n \ xs \in \text{listset } (\text{take } n \ XS)$

**by** (*induction XS arbitrary: xs n*) (*auto simp: take-Cons listset-Cons-mem-conv set-Cons-def split: nat.splits*)

**lemma** *take-in-aforms-errI*:  $\text{take } n \ xs \in \text{aforms-err } e \text{ (take } n \ XS)$

**if**  $xs \in \text{aforms-err } e \text{ XS}$

**using** *that*

**by** (*auto simp: aforms-err-def take-map[symmetric] intro!: take-in-listsetI*)

**theorem** *approx-slp-outer*:

**assumes**  $\text{approx-slp-outer } p \ n \ \text{slp } XS = \text{Some } RS$

**assumes**  $slp$ :  $slp = slp\text{-of-fas fas } n = \text{length fas}$   
**assumes**  $xs \in \text{Joints } XS$   
**shows**  $\text{interpret-floatariths fas } xs @ xs \in \text{Joints } (RS @ XS)$   
**proof** –  
**from**  $assms$  **obtain**  $d XSe rs rs' d'$  **where**  
 $d$ :  $d = \text{degree-aforms } XS$   
**and**  $XSe$ :  $XSe = (\text{map } (\lambda x. (x, 0)) XS)$   
**and**  $rs$ :  $\text{approx-slp } p (slp\text{-of-fas fas}) XSe = \text{Some } rs$   
**and**  $rs'$ :  $rs' = \text{take } (\text{length fas}) rs$   
**and**  $d'$ :  $d' = \text{max } d (\text{degree-aforms-err } rs')$   
**and**  $RS$ :  $\text{aforms-err-to-aforms } d' rs' = RS$   
**by** ( $\text{auto simp: approx-slp-outer-def Let-def bind-eq-Some-conv}$ )  
**have**  $abbd$ :  $(a, b) \in \text{set } XS \implies \text{degree } b \leq \text{degree-aforms } XS$  **for**  $a b$   
**apply** ( $\text{rule degrees-leD[OF order-refl]}$ ) **by**  $\text{force}$   
**from**  $d' d$  **have**  $i\text{-less}$ :  $(a, b) \in \text{set } XS \implies i < \text{degree } b \implies i < \text{min } d d'$  **for**  $i$   
 $a b$   
**by** ( $\text{auto dest!: abbd}$ )  
**from**  $assms$  **obtain**  $e'$  **where**  $vs$ :  $xs = (\text{map } (\text{aform-val } e') XS)$  **and**  $e'$ :  $e' \in$   
 $UNIV \rightarrow \{-1 .. 1\}$   
**by** ( $\text{auto simp: Joints-def valuate-def}$ )  
**from**  $d$  **have**  $d$ :  $V \in \text{set } XS \implies \text{degree-aform } V \leq d$  **for**  $V$   
**by** ( $\text{auto intro!: degrees-leD}$ )  
**have**  $xs\text{-XSe}$ :  $xs \in \text{aforms-err } e' XSe$   
**by** ( $\text{auto simp: vs aforms-err-def XSe o-def aform-err-def}$ )  
**from**  $\text{approx-slp-Elem[OF } e' xs\text{-XSe } rs]$   
**have**  $\text{aforms-err}$ :  $\text{interpret-slp } (slp\text{-of-fas fas}) xs \in \text{aforms-err } e' rs$  .  
**have**  $\text{interpret-floatariths fas } xs = \text{take } (\text{length fas}) (\text{interpret-slp } (slp\text{-of-fas fas})$   
 $xs)$   
**using**  $assms$  **by** ( $\text{simp add: slp-of-fas}$ )  
**also**  
**from**  $\text{aforms-err}$   
**have**  $\text{take } (\text{length fas}) (\text{interpret-slp } (slp\text{-of-fas fas}) xs) \in \text{aforms-err } e' rs'$   
**unfolding**  $rs'$   
**by** ( $\text{auto simp: take-map intro!: take-in-aforms-errI}$ )  
**finally** **have**  $\text{ier}$ :  $\text{interpret-floatariths fas } xs \in \text{aforms-err } e' rs'$  .  
**have**  $\text{degree-aforms-err } rs' \leq d'$  **using**  $d'$  **by**  $\text{auto}$   
**from**  $\text{aforms-err-to-aformsE[OF } \text{ier this } e']$  **obtain**  $e$  **where**  
 $\text{interpret-floatariths fas } xs = \text{aform-vals } e RS$   
**and**  $e$ :  $e \in UNIV \rightarrow \{-1 .. 1\} \wedge i. i < d' \implies e i = e' i$   
**unfolding**  $RS$   
**by**  $\text{auto}$   
**moreover**  
**from**  $e$  **have**  $e'\text{-eq}$ :  $e' i = e i$  **if**  $i < \text{min } d d'$  **for**  $i$   
**using**  $\text{that}$   
**by** ( $\text{auto simp: min-def split: if-splits}$ )  
**then** **have**  $xs = \text{aform-vals } e XS$   
**by** ( $\text{auto simp: vs aform-vals-def aform-val-def intro!: pdevs-val-degree-cong } e'\text{-eq}$   
 $i\text{-less}$ )

```

ultimately show ?thesis
  using e(1)
  by (auto simp: Joints-def valuate-def aform-vals-def intro!: image-eqI[where
x=e])

qed

theorem approx-slp-outer-plain:
  assumes approx-slp-outer p n slp XS = Some RS
  assumes slp: slp = slp-of-fas fas n = length fas
  assumes xs ∈ Joints XS
  shows interpret-floatariths fas xs ∈ Joints RS
proof -
  have length fas = length RS
proof -
  have f1: length xs = length XS
  using Joints-imp-length-eq assms(4) by blast
  have interpret-floatariths fas xs @ xs ∈ Joints (RS @ XS)
  using approx-slp-outer assms(1) assms(2) assms(3) assms(4) by blast
  then show ?thesis
  using f1 Joints-imp-length-eq by fastforce
qed
with Joints-appendD2[OF approx-slp-outer[OF assms]] show ?thesis by simp
qed

end

end

```

## 6 Counterclockwise

```

theory Counterclockwise
imports HOL-Analysis.Multivariate-Analysis
begin

```

### 6.1 Auxiliary Lemmas

```

lemma convex3-alt:
  fixes x y z::'a::real-vector
  assumes 0 ≤ a 0 ≤ b 0 ≤ c a + b + c = 1
  obtains u v where a *R x + b *R y + c *R z = x + u *R (y - x) + v *R (z
- x)
  and 0 ≤ u 0 ≤ v u + v ≤ 1
proof -
  from convex-hull-3[of x y z] have a *R x + b *R y + c *R z ∈ convex hull {x,
y, z}
  using assms by auto

```

**also note** *convex-hull-3-alt*  
**finally obtain**  $u v$  **where**  $a *_R x + b *_R y + c *_R z = x + u *_R (y - x) + v *_R (z - x)$   
**and**  $uv: 0 \leq u \ 0 \leq v \ u + v \leq 1$   
**by** *auto*  
**thus** *?thesis ..*  
**qed**

**lemma** (*in ordered-ab-group-add*) *add-nonpos-eq-0-iff*:  
**assumes**  $x: 0 \geq x$  **and**  $y: 0 \geq y$   
**shows**  $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$   
**proof** –  
**from** *add-nonneg-eq-0-iff*[*of -x -y*] *assms*  
**have**  $-(x + y) = 0 \longleftrightarrow -x = 0 \wedge -y = 0$   
**by** *simp*  
**also have**  $-(x + y) = 0 = (x + y = 0)$  **unfolding** *neg-equal-0-iff-equal ..*  
**finally show** *?thesis* **by** *simp*  
**qed**

**lemma** *sum-nonpos-eq-0-iff*:  
**fixes**  $f :: 'a \Rightarrow 'b::ordered-ab-group-add$   
**shows**  $\llbracket \text{finite } A; \forall x \in A. f\ x \leq 0 \rrbracket \Longrightarrow \text{sum } f\ A = 0 \longleftrightarrow (\forall x \in A. f\ x = 0)$   
**by** (*induct set: finite*) (*simp-all add: add-nonpos-eq-0-iff sum-nonpos*)

**lemma** *fold-if-in-set*:  
*fold*  $(\lambda x\ m. \text{if } P\ x\ m\ \text{then } x\ \text{else } m)$   $xs\ x \in \text{set } (x \# xs)$   
**by** (*induct xs arbitrary: x*) *auto*

## 6.2 Sort Elements of a List

**locale** *linorder-list0* = **fixes**  $le::'a \Rightarrow 'a \Rightarrow \text{bool}$   
**begin**

**definition** *min-for*  $a\ b = (\text{if } le\ a\ b\ \text{then } a\ \text{else } b)$

**lemma** *min-for-in*[*simp*]:  $x \in S \Longrightarrow y \in S \Longrightarrow \text{min-for } x\ y \in S$   
**by** (*auto simp: min-for-def*)

**lemma** *fold-min-eqI1*:  $\text{fold } \text{min-for } ys\ y \notin \text{set } ys \Longrightarrow \text{fold } \text{min-for } ys\ y = y$   
**using** *fold-if-in-set*[*of - ys y*]  
**by** (*auto simp: min-for-def*[*abs-def*])

**function** *selsort* **where**  
*selsort*  $[] = []$   
| *selsort*  $(y \# ys) = (\text{let}$   
     $xm = \text{fold } \text{min-for } ys\ y;$   
     $xs' = \text{List.remove1 } xm\ (y \# ys)$   
     $\text{in } (xm \# \text{selsort } xs'))$   
**by** *pat-completeness auto*

**termination**

**by** (*relation* *Wellfounded.measure length*)  
*(auto simp: length-remove1 intro!: fold-min-eqI1 dest!: length-pos-if-in-set)*

**lemma** *in-set-selsort-eq*:  $x \in \text{set } (\text{selsort } xs) \longleftrightarrow x \in (\text{set } xs)$   
**by** (*induct rule: selsort.induct*) (*auto simp: Let-def intro!: fold-min-eqI1*)

**lemma** *set-selsort[simp]*:  $\text{set } (\text{selsort } xs) = \text{set } xs$   
**using** *in-set-selsort-eq* **by** *blast*

**lemma** *length-selsort[simp]*:  $\text{length } (\text{selsort } xs) = \text{length } xs$   
**proof** (*induct xs rule: selsort.induct*)

**case** ( $2\ x\ xs$ )  
**from**  $2[OF\ refl\ refl]$   
**show** *?case*  
**unfolding** *selsort.simps*  
**by** (*auto simp: Let-def length-remove1*  
*simp del: selsort.simps split: if-split-asm*  
*intro!: Suc-pred*  
*dest!: fold-min-eqI1*)

**qed** *simp*

**lemma** *distinct-selsort[simp]*:  $\text{distinct } (\text{selsort } xs) = \text{distinct } xs$   
**by** (*auto intro!: card-distinct dest!: distinct-card*)

**lemma** *selsort-eq-empty-iff[simp]*:  $\text{selsort } xs = [] \longleftrightarrow xs = []$   
**by** (*cases xs*) (*auto simp: Let-def*)

**inductive** *sortedP* :: '*a list*  $\Rightarrow$  *bool* **where**

*Nil: sortedP []*  
| *Cons:  $\forall y \in \text{set } ys. le\ x\ y \Longrightarrow \text{sortedP } ys \Longrightarrow \text{sortedP } (x \# ys)$*

**inductive-cases**

*sortedP-Nil: sortedP []* **and**  
*sortedP-Cons: sortedP (x#xs)*

**inductive-simps**

*sortedP-Nil-iff: sortedP Nil* **and**  
*sortedP-Cons-iff: sortedP (Cons x xs)*

**lemma** *sortedP-append-iff*:

*sortedP (xs @ ys) = (sortedP xs & sortedP ys & ( $\forall x \in \text{set } xs. \forall y \in \text{set } ys. le\ x\ y$ ))*  
**by** (*induct xs*) (*auto intro!: Nil Cons elim!: sortedP-Cons*)

**lemma** *sortedP-appendI*:

*sortedP xs  $\Longrightarrow$  sortedP ys  $\Longrightarrow$  ( $\bigwedge x\ y. x \in \text{set } xs \Longrightarrow y \in \text{set } ys \Longrightarrow le\ x\ y$ )  $\Longrightarrow$  sortedP (xs @ ys)*  
**by** (*induct xs*) (*auto intro!: Nil Cons elim!: sortedP-Cons*)

```

lemma sorted-nth-less: sortedP xs  $\implies$   $i < j \implies j < \text{length } xs \implies \text{le } (xs ! i) (xs ! j)$ 
  by (induct xs arbitrary: i j) (auto simp: nth-Cons split: nat.split elim!: sortedP-Cons)

lemma sorted-butlastI[intro, simp]: sortedP xs  $\implies$  sortedP (butlast xs)
  by (induct xs) (auto simp: elim!: sortedP-Cons intro!: sortedP.Cons dest!: in-set-butlastD)

lemma sortedP-right-of-append1:
  assumes sortedP (zs@[z])
  assumes  $y \in \text{set } zs$ 
  shows  $\text{le } y z$ 
  using assms
  by (induct zs arbitrary: y z) (auto elim!: sortedP-Cons)

lemma sortedP-right-of-last:
  assumes sortedP zs
  assumes  $y \in \text{set } zs \ y \neq \text{last } zs$ 
  shows  $\text{le } y (\text{last } zs)$ 
  using assms
  apply (intro sortedP-right-of-append1 [of butlast zs last zs y])
  subgoal by (metis append-is-Nil-conv list.distinct(1) snoc-eq-iff-butlast split-list)
  subgoal by (metis List.insert-def append-butlast-last-id insert-Nil list.distinct(1) rotate1.simps(2) set-ConsD set-rotate1)
  done

lemma selsort-singleton-iff: selsort xs = [x]  $\longleftrightarrow$   $xs = [x]$ 
  by (induct xs) (auto simp: Let-def)

lemma hd-last-sorted:
  assumes sortedP xs length xs > 1
  shows  $\text{le } (\text{hd } xs) (\text{last } xs)$ 
proof (cases xs)
  case (Cons y ys)
  note  $ys = \text{this}$ 
  thus ?thesis
  using ys assms
  by (auto elim!: sortedP-Cons)
qed (insert assms, simp)

end

lemma (in comm-monoid-add) sum-list-distinct-selsort:
  assumes distinct xs
  shows  $\text{sum-list } (\text{linorder-list0.selsort } \text{le } xs) = \text{sum-list } xs$ 
  using assms
  apply (simp add: distinct-sum-list-conv-Sum linorder-list0.distinct-selsort)

```

```

apply (rule sum.cong)
subgoal by (simp add: linorder-list0.set-selsort)
subgoal by simp
done

declare linorder-list0.sortedP-Nil-iff[code]
linorder-list0.sortedP-Cons-iff[code]
linorder-list0.selsort.simps[code]
linorder-list0.min-for-def[code]

locale linorder-list = linorder-list0 le for le::'a::ab-group-add  $\Rightarrow$  - +
fixes S
assumes order-refl:  $a \in S \Rightarrow le\ a\ a$ 
assumes trans':  $a \in S \Rightarrow b \in S \Rightarrow c \in S \Rightarrow a \neq b \Rightarrow b \neq c \Rightarrow a \neq c$ 
 $\Rightarrow$ 
le a b  $\Rightarrow$  le b c  $\Rightarrow$  le a c
assumes antisym:  $a \in S \Rightarrow b \in S \Rightarrow le\ a\ b \Rightarrow le\ b\ a \Rightarrow a = b$ 
assumes linear':  $a \in S \Rightarrow b \in S \Rightarrow a \neq b \Rightarrow le\ a\ b \vee le\ b\ a$ 
begin

lemma trans:  $a \in S \Rightarrow b \in S \Rightarrow c \in S \Rightarrow le\ a\ b \Rightarrow le\ b\ c \Rightarrow le\ a\ c$ 
by (cases a = b b = c a = c
rule: bool.exhaust[case-product bool.exhaust[case-product bool.exhaust]])
(auto simp: order-refl intro: trans')

lemma linear:  $a \in S \Rightarrow b \in S \Rightarrow le\ a\ b \vee le\ b\ a$ 
by (cases a = b) (auto simp: linear' order-refl)

lemma min-le1:  $w \in S \Rightarrow y \in S \Rightarrow le\ (min\text{-for}\ w\ y)\ y$ 
and min-le2:  $w \in S \Rightarrow y \in S \Rightarrow le\ (min\text{-for}\ w\ y)\ w$ 
using linear
by (auto simp: min-for-def refl)

lemma fold-min:
assumes set xs  $\subseteq$  S
shows list-all ( $\lambda y. le\ (fold\ min\text{-for}\ (tl\ xs)\ (hd\ xs))\ y$ ) xs
proof (cases xs)
case (Cons y ys)
hence subset: set (y#ys)  $\subseteq$  S using assms
by auto
show ?thesis
unfolding Cons list.sel
using subset
proof (induct ys arbitrary: y)
case (Cons z zs)
hence IH:  $\bigwedge y. y \in S \Rightarrow list\text{-all}\ (le\ (fold\ min\text{-for}\ zs\ y))\ (y\ \#)\ zs$ 
by simp
let ?f = fold min-for zs (min-for z y)
have ?f  $\in$  set ((min-for z y)#zs)

```



```

    unfolding min-for-def[abs-def]
    by (rule fold-if-in-set)
  also have ...  $\subseteq S$  using Cons.prem $s$  by auto
  finally have  $?f \in S$  .

  have le ?f (min-for z y)
    using IH[of min-for z y] Cons.prem $s$ 
    by auto
  moreover have le (min-for z y) y le (min-for z y) z using Cons.prem $s$ 
    by (auto intro!: min-le1 min-le2)
  ultimately have le ?f y le ?f z using Cons.prem $s$   $\langle ?f \in S \rangle$ 
    by (auto intro!: trans[of ?f min-for z y])
  thus ?case
    using IH[of min-for z y]
    using Cons.prem $s$ 
    by auto
  qed (simp add: order-refl)
qed simp

```

```

lemma
  sortedP-selsort:
  assumes set xs  $\subseteq S$ 
  shows sortedP (selsort xs)
  using assms
proof (induction xs rule: selsort.induct)
  case (2 z zs)
  from this fold-min[of z#zs]
  show ?case
    by (fastforce simp: list-all-iff Let-def
        simp del: remove1.simps
        intro: Cons intro!: 2(1)[OF refl refl]
        dest!: rev-subsetD[OF - set-remove1-subset])+)
qed (auto intro!: Nil)

```

end

### 6.3 Abstract CCW Systems

```

locale ccw-system0 =
  fixes ccw::'a  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  and S::'a set
begin

  abbreviation indelta t p q r  $\equiv$  ccw t q r  $\wedge$  ccw p t r  $\wedge$  ccw p q t
  abbreviation insquare p q r s  $\equiv$  ccw p q r  $\wedge$  ccw q r s  $\wedge$  ccw r s p  $\wedge$  ccw s p q

end

abbreviation distinct3 p q r  $\equiv$   $\neg(p = q \vee p = r \vee q = r)$ 

```

**abbreviation**  $distinct4\ p\ q\ r\ s \equiv \neg(p = q \vee p = r \vee p = s \vee \neg distinct3\ q\ r\ s)$   
**abbreviation**  $distinct5\ p\ q\ r\ s\ t \equiv \neg(p = q \vee p = r \vee p = s \vee p = t \vee \neg distinct4\ q\ r\ s\ t)$

**abbreviation**  $in3\ S\ p\ q\ r \equiv p \in S \wedge q \in S \wedge r \in S$   
**abbreviation**  $in4\ S\ p\ q\ r\ s \equiv in3\ S\ p\ q\ r \wedge s \in S$   
**abbreviation**  $in5\ S\ p\ q\ r\ s\ t \equiv in4\ S\ p\ q\ r\ s \wedge t \in S$

**locale**  $ccw\text{-}system12 = ccw\text{-}system0 +$   
**assumes**  $cyclic: ccw\ p\ q\ r \implies ccw\ q\ r\ p$   
**assumes**  $ccw\text{-}antisym: distinct3\ p\ q\ r \implies in3\ S\ p\ q\ r \implies ccw\ p\ q\ r \implies \neg ccw\ p\ r\ q$

**locale**  $ccw\text{-}system123 = ccw\text{-}system12 +$   
**assumes**  $nondegenerate: distinct3\ p\ q\ r \implies in3\ S\ p\ q\ r \implies ccw\ p\ q\ r \vee ccw\ p\ r\ q$   
**begin**

**lemma**  $not\text{-}ccw\text{-}eq: distinct3\ p\ q\ r \implies in3\ S\ p\ q\ r \implies \neg ccw\ p\ q\ r \longleftrightarrow ccw\ p\ r\ q$   
**using**  $ccw\text{-}antisym\ nondegenerate$  **by**  $blast$

**end**

**locale**  $ccw\text{-}system4 = ccw\text{-}system123 +$   
**assumes**  $interior:$   
 $distinct4\ p\ q\ r\ t \implies in4\ S\ p\ q\ r\ t \implies ccw\ t\ q\ r \implies ccw\ p\ t\ r \implies ccw\ p\ q\ t$   
 $\implies ccw\ p\ q\ r$   
**begin**

**lemma**  $interior':$   
 $distinct4\ p\ q\ r\ t \implies in4\ S\ p\ q\ r\ t \implies ccw\ p\ q\ t \implies ccw\ q\ r\ t \implies ccw\ r\ p\ t \implies$   
 $ccw\ p\ q\ r$   
**by**  $(metis\ ccw\text{-}antisym\ cyclic\ interior\ nondegenerate)$

**end**

**locale**  $ccw\text{-}system1235' = ccw\text{-}system123 +$   
**assumes**  $dual\text{-}transitive:$   
 $distinct5\ p\ q\ r\ s\ t \implies in5\ S\ p\ q\ r\ s\ t \implies$   
 $ccw\ s\ t\ p \implies ccw\ s\ t\ q \implies ccw\ s\ t\ r \implies ccw\ t\ p\ q \implies ccw\ t\ q\ r \implies ccw\ t\ p\ r$

**locale**  $ccw\text{-}system1235 = ccw\text{-}system123 +$   
**assumes**  $transitive: distinct5\ p\ q\ r\ s\ t \implies in5\ S\ p\ q\ r\ s\ t \implies$   
 $ccw\ t\ s\ p \implies ccw\ t\ s\ q \implies ccw\ t\ s\ r \implies ccw\ t\ p\ q \implies ccw\ t\ q\ r \implies ccw\ t\ p\ r$   
**begin**

**lemmas**  $ccw\text{-}axioms = cyclic\ nondegenerate\ ccw\text{-}antisym\ transitive$

**sublocale**  $ccw\text{-}system1235'$

```

proof (unfold-locates, rule ccontr, goal-cases)
  case prems: (1 p q r s t)
  hence  $ccw\ s\ p\ q \implies ccw\ s\ r\ p$ 
    by (metis ccw-axioms prems)
  moreover
  have  $ccw\ s\ r\ p \implies ccw\ s\ q\ r$ 
    by (metis ccw-axioms prems)
  moreover
  have  $ccw\ s\ q\ r \implies ccw\ s\ p\ q$ 
    by (metis ccw-axioms prems)
  ultimately
  have  $ccw\ s\ p\ q \wedge ccw\ s\ r\ p \wedge ccw\ s\ q\ r \vee ccw\ s\ q\ p \wedge ccw\ s\ p\ r \wedge ccw\ s\ r\ q$ 
    by (metis ccw-axioms prems)
  thus False
    by (metis ccw-axioms prems)
qed

end

locale ccw-system = ccw-system1235 + ccw-system4

end

```

## 7 CCW Vector Space

```

theory Counterclockwise-Vector
imports Counterclockwise
begin

```

```

locale ccw-vector-space = ccw-system12 ccw S for ccw::'a::real-vector  $\Rightarrow 'a \Rightarrow 'a$ 
 $\Rightarrow \text{bool}$  and S +
  assumes translate-plus[simp]:  $ccw\ (a + x)\ (b + x)\ (c + x) \longleftrightarrow ccw\ a\ b\ c$ 
  assumes scaleR1-eq[simp]:  $0 < e \implies ccw\ 0\ (e*_R a)\ b = ccw\ 0\ a\ b$ 
  assumes uminus1[simp]:  $ccw\ 0\ (-a)\ b = ccw\ 0\ b\ a$ 
  assumes add1:  $ccw\ 0\ a\ b \implies ccw\ 0\ c\ b \implies ccw\ 0\ (a + c)\ b$ 
begin

```

```

lemma translate-plus'[simp]:
   $ccw\ (x + a)\ (x + b)\ (x + c) \longleftrightarrow ccw\ a\ b\ c$ 
  by (auto simp: ac-simps)

```

```

lemma uminus2[simp]:  $ccw\ 0\ a\ (-b) = ccw\ 0\ b\ a$ 
  by (metis minus-minus uminus1)

```

```

lemma uminus-all[simp]:  $ccw\ (-a)\ (-b)\ (-c) \longleftrightarrow ccw\ a\ b\ c$ 
proof -

```

```

  have  $ccw\ (-a)\ (-b)\ (-c) \longleftrightarrow ccw\ 0\ (- (b - a))\ (- (c - a))$ 
    using translate-plus[of -a a -b -c]
    by simp

```

**also have**  $\dots \longleftrightarrow ccw\ 0\ (b - a)\ (c - a)$   
**by** *(simp del: minus-diff-eq)*  
**also have**  $\dots \longleftrightarrow ccw\ a\ b\ c$   
**using** *translate-plus[of a -a b c]*  
**by** *simp*  
**finally show** *?thesis* .  
**qed**

**lemma** *translate-origin: NO-MATCH*  $0\ p \implies ccw\ p\ q\ r \longleftrightarrow ccw\ 0\ (q - p)\ (r - p)$   
**using** *translate-plus[of p - p q r]*  
**by** *simp*

**lemma** *translate[simp]:*  $ccw\ a\ (a + b)\ (a + c) \longleftrightarrow ccw\ 0\ b\ c$   
**by** *(simp add: translate-origin)*

**lemma** *translate-plus3:*  $ccw\ (a - x)\ (b - x)\ c \longleftrightarrow ccw\ a\ b\ (c + x)$   
**using** *translate-plus[of a -x b c + x]* **by** *simp*

**lemma** *renormalize:*  
 $ccw\ 0\ (a - b)\ (c - a) \implies ccw\ b\ a\ c$   
**by** *(metis diff-add-cancel diff-self cyclic minus-diff-eq translate-plus3 uminus1)*

**lemma** *cyclicI:*  $ccw\ p\ q\ r \implies ccw\ q\ r\ p$   
**by** *(metis cyclic)*

**lemma**  
*scaleR2-eq[simp]:*  
 $0 < e \implies ccw\ 0\ xr\ (e *_R P) \longleftrightarrow ccw\ 0\ xr\ P$   
**using** *scaleR1-eq[of e -P xr]*  
**by** *simp*

**lemma** *scaleR1-nonzero-eq:*  
 $e \neq 0 \implies ccw\ 0\ (e *_R a)\ b = (if\ e > 0\ then\ ccw\ 0\ a\ b\ else\ ccw\ 0\ b\ a)$   
**proof** *cases*  
**assume**  $e < 0$   
**define**  $e'$  **where**  $e' = -e$   
**hence**  $e = -e'\ e' > 0$  **using**  $\langle e < 0 \rangle$  **by** *simp-all*  
**thus** *?thesis* **by** *simp*  
**qed** *simp*

**lemma** *neg-scaleR[simp]:*  $x < 0 \implies ccw\ 0\ (x *_R b)\ c \longleftrightarrow ccw\ 0\ c\ b$   
**using** *scaleR1-nonzero-eq* **by** *auto*

**lemma**  
*scaleR1:*  
 $0 < e \implies ccw\ 0\ xr\ P \implies ccw\ 0\ (e *_R xr)\ P$   
**by** *simp*

**lemma**

*add3*:  $ccw\ 0\ a\ b \wedge ccw\ 0\ a\ c \implies ccw\ 0\ a\ (b + c)$   
**using** *add1*[of  $-b\ a\ -c$ ] *uminus1*[of  $b + c\ a$ ]  
**by** *simp*

**lemma** *add3-self*[*simp*]:  $ccw\ 0\ p\ (p + q) \longleftrightarrow ccw\ 0\ p\ q$

**using** *translate*[of  $-p\ p\ p + q$ ]  
**apply** (*simp add: cyclic*)  
**apply** (*metis cyclic uminus2*)  
**done**

**lemma** *add2-self*[*simp*]:  $ccw\ 0\ (p + q)\ p \longleftrightarrow ccw\ 0\ q\ p$

**using** *translate*[of  $-p\ p + q\ p$ ]  
**apply** *simp*  
**apply** (*metis cyclic uminus1*)  
**done**

**lemma** *scale-add3*[*simp*]:  $ccw\ 0\ a\ (x *_{R}\ a + b) \longleftrightarrow ccw\ 0\ a\ b$

**proof** –

{  
  **assume**  $x = 0$   
  **hence** *?thesis* **by** *simp*  
} **moreover** {  
  **assume**  $x > 0$   
  **hence** *?thesis* **using** *add3-self scaleR1-eq* **by** *blast*  
} **moreover** {  
  **assume**  $x < 0$   
  **define**  $x'$  **where**  $x' = -x$   
  **hence**  $x = -x'\ x' > 0$  **using**  $\langle x < 0 \rangle$  **by** *simp-all*  
  **hence**  $ccw\ 0\ a\ (x *_{R}\ a + b) = ccw\ 0\ (x' *_{R}\ a + -b)\ (x' *_{R}\ a)$   
  **by** (*subst uminus1[symmetric]*) *simp*  
  **also have**  $\dots = ccw\ 0\ (-b)\ a$   
  **unfolding** *add2-self* **by** (*simp add:  $\langle x' > 0 \rangle$* )  
  **also have**  $\dots = ccw\ 0\ a\ b$   
  **by** *simp*  
  **finally have** *?thesis* .  
} **ultimately show** *?thesis* **by** *arith*

**qed**

**lemma** *scale-add3'*[*simp*]:  $ccw\ 0\ a\ (b + x *_{R}\ a) \longleftrightarrow ccw\ 0\ a\ b$

**and** *scale-minus3*[*simp*]:  $ccw\ 0\ a\ (x *_{R}\ a - b) \longleftrightarrow ccw\ 0\ b\ a$

**and** *scale-minus3'*[*simp*]:  $ccw\ 0\ a\ (b - x *_{R}\ a) \longleftrightarrow ccw\ 0\ a\ b$

**using**

*scale-add3*[of  $a\ x\ b$ ]

*scale-add3*[of  $a\ -x\ b$ ]

*scale-add3*[of  $a\ x\ -b$ ]

**by** (*simp-all add: ac-simps*)

**lemma** *sum*:

```

assumes fin: finite X
assumes ne:  $X \neq \{\}$ 
assumes ncoll:  $(\bigwedge x. x \in X \implies \text{ccw } 0 \ a \ (f \ x))$ 
shows  $\text{ccw } 0 \ a \ (\text{sum } f \ X)$ 
proof -
from ne obtain x where  $x \in X$  insert x  $X = X$  by auto
have  $\text{ccw } 0 \ a \ (\text{sum } f \ (\text{insert } x \ X))$ 
using fin ncoll
proof (induction X)
case empty thus ?case using  $\langle x \in X \rangle$  ncoll
by auto
next
case (insert y F)
hence  $\text{ccw } 0 \ a \ (\text{sum } f \ (\text{insert } y \ (\text{insert } x \ F)))$ 
by (cases y = x) (auto intro!: add3)
thus ?case
by (simp add: insert-commute)
qed
thus ?thesis using  $\langle \text{insert } x \ X = X \rangle$  by simp
qed

```

```

lemma sum2:
assumes fin: finite X
assumes ne:  $X \neq \{\}$ 
assumes ncoll:  $(\bigwedge x. x \in X \implies \text{ccw } 0 \ (f \ x) \ a)$ 
shows  $\text{ccw } 0 \ (\text{sum } f \ X) \ a$ 
using sum[OF assms(1,2), of -a f] ncoll
by simp

```

```

lemma translate-minus[simp]:
 $\text{ccw } (x - a) \ (x - b) \ (x - c) = \text{ccw } (-a) \ (-b) \ (-c)$ 
using translate-plus[of -a x -b -c]
by simp

```

**end**

```

locale ccw-convex = ccw-system ccw S for ccw and  $S::'a::\text{real-vector set} +$ 
fixes oriented
assumes convex2:
 $u \geq 0 \implies v \geq 0 \implies u + v = 1 \implies \text{ccw } a \ b \ c \implies \text{ccw } a \ b \ d \implies \text{oriented } a$ 
 $b \implies$ 
 $\text{ccw } a \ b \ (u *_R c + v *_R d)$ 
begin

```

```

lemma convex-hull:
assumes [intro, simp]: finite C
assumes ccw:  $\bigwedge c. c \in C \implies \text{ccw } a \ b \ c$ 
assumes ch:  $x \in \text{convex hull } C$ 
assumes oriented: oriented a b

```

```

shows ccw a b x
proof -
  define D where D = C
  have D: C ⊆ D ∧ c. c ∈ D ⇒ ccw a b c by (simp-all add: D-def ccw)
  show ccw a b x
    using ⟨finite C⟩ D ch
  proof (induct arbitrary: x)
    case empty thus ?case by simp
  next
    case (insert c C)
    hence C ⊆ D by simp
    {
      assume C = {}
      hence ?case
        using insert
        by simp
    } moreover {
      assume C ≠ {}
      from convex-hull-insert[OF this, of c] insert(6)
      obtain u v d where u ≥ 0 v ≥ 0 d ∈ convex hull C u + v = 1
        and x: x = u *R c + v *R d
        by blast
      have ccw a b d
        by (auto intro: insert.hyps(3)[OF ⟨C ⊆ D⟩] insert.prem1 ⟨d ∈ convex hull
C⟩)
      from insert
      have ccw a b c
        by simp
      from convex2[OF ⟨0 ≤ u⟩ ⟨0 ≤ v⟩ ⟨u + v = 1⟩ ⟨ccw a b c⟩ ⟨ccw a b d⟩
⟨oriented a b⟩]
      have ?case by (simp add: x)
    } ultimately show ?case by blast
  qed
qed
end
end

```

## 8 CCW for Nonaligned Points in the Plane

```

theory Counterclockwise-2D-Strict
  imports
    Counterclockwise-Vector
    Affine-Arithmetic-Auxiliarities
begin

```

## 8.1 Determinant

**type-synonym** *point* = *real\*real*

**fun** *det3*::*point*  $\Rightarrow$  *point*  $\Rightarrow$  *point*  $\Rightarrow$  *real* **where** *det3* (*xp*, *yp*) (*xq*, *yq*) (*xr*, *yr*) =  
 $xp * yq + yp * xr + xq * yr - yq * xr - yp * xq - xp * yr$

**lemma** *det3-def'*:

$det3\ p\ q\ r = fst\ p * snd\ q + snd\ p * fst\ r + fst\ q * snd\ r -$   
 $snd\ q * fst\ r - snd\ p * fst\ q - fst\ p * snd\ r$

**by** (*cases p q r rule: prod.exhaust[case-product prod.exhaust[case-product prod.exhaust]]*)  
*auto*

**lemma** *det3-eq-det*: *det3* (*xa*, *ya*) (*xb*, *yb*) (*xc*, *yc*) =

*det* (*vector* [*vector* [*xa*, *ya*, 1], *vector* [*xb*, *yb*, 1], *vector* [*xc*, *yc*, 1]]::*real*<sup>3</sup>)

**unfolding** *Determinants.det-def UNIV-3*

**by** (*auto simp: sum-over-permutations-insert*  
*vector-3 sign-swap-id permutation-swap-id sign-compose*)

**declare** *det3.simps[simp del]*

**lemma** *det3-self23[simp]*: *det3* *a b b* = 0

**and** *det3-self12[simp]*: *det3* *b b a* = 0

**by** (*auto simp: det3-def'*)

**lemma**

*coll-ex-scaling*:

**assumes** *b*  $\neq$  *c*

**assumes** *d*: *det3* *a b c* = 0

**shows**  $\exists r. a = b + r *_R (c - b)$

**proof** –

**from** *assms* **have** *fst b*  $\neq$  *fst c*  $\vee$  *snd b*  $\neq$  *snd c* **by** (*auto simp: prod-eq-iff*)

**thus** *?thesis*

**proof**

**assume** *neq*: *fst b*  $\neq$  *fst c*

**with** *d* **have**  $snd\ a = ((fst\ a - fst\ b) * snd\ c + (fst\ c - fst\ a) * snd\ b) / (fst\ c - fst\ b)$

**by** (*auto simp: det3-def' field-simps*)

**hence**  $snd\ a = ((fst\ a - fst\ b) / (fst\ c - fst\ b)) * snd\ c +$

$((fst\ c - fst\ a) / (fst\ c - fst\ b)) * snd\ b$

**by** (*simp add: add-divide-distrib*)

**hence**  $snd\ a = snd\ b + (fst\ a - fst\ b) * snd\ c / (fst\ c - fst\ b) +$

$((fst\ c - fst\ a) - (fst\ c - fst\ b)) * snd\ b / (fst\ c - fst\ b)$

**using** *neq*

**by** (*simp add: field-simps*)

**hence**  $snd\ a = snd\ b + ((fst\ a - fst\ b) * snd\ c + (-fst\ a + fst\ b) * snd\ b) / (fst\ c - fst\ b)$

**unfolding** *add-divide-distrib*

**by** (*simp add: algebra-simps*)

**also**



**have**  $(fst\ a - fst\ b) * snd\ c + (-\ fst\ a + fst\ b) * snd\ b = (fst\ a - fst\ b) * (snd\ c - snd\ b)$   
**by** *(simp add: algebra-simps)*  
**finally have**  $snd\ a = snd\ b + (fst\ a - fst\ b) / (fst\ c - fst\ b) * (snd\ c - snd\ b)$   
**by** *simp*  
**moreover**  
**hence**  $fst\ a = fst\ b + (fst\ a - fst\ b) / (fst\ c - fst\ b) * (fst\ c - fst\ b)$   
**using** *neq by simp*  
**ultimately have**  $a = b + ((fst\ a - fst\ b) / (fst\ c - fst\ b)) *_R (c - b)$   
**by** *(auto simp: prod-eq-iff)*  
**thus** *?thesis by blast*  
**next**  
**assume** *neq: snd\ b  $\neq$  snd\ c*  
**with** *d* **have**  $fst\ a = ((snd\ a - snd\ b) * fst\ c + (snd\ c - snd\ a) * fst\ b) / (snd\ c - snd\ b)$   
**by** *(auto simp: det3-def' field-simps)*  
**hence**  $fst\ a = ((snd\ a - snd\ b) / (snd\ c - snd\ b)) * fst\ c + ((snd\ c - snd\ a) / (snd\ c - snd\ b)) * fst\ b$   
**by** *(simp add: add-divide-distrib)*  
**hence**  $fst\ a = fst\ b + (snd\ a - snd\ b) * fst\ c / (snd\ c - snd\ b) + ((snd\ c - snd\ a) - (snd\ c - snd\ b)) * fst\ b / (snd\ c - snd\ b)$   
**using** *neq*  
**by** *(simp add: field-simps)*  
**hence**  $fst\ a = fst\ b + ((snd\ a - snd\ b) * fst\ c + (-\ snd\ a + snd\ b) * fst\ b) / (snd\ c - snd\ b)$   
**unfolding** *add-divide-distrib*  
**by** *(simp add: algebra-simps)*  
**also**  
**have**  $(snd\ a - snd\ b) * fst\ c + (-\ snd\ a + snd\ b) * fst\ b = (snd\ a - snd\ b) * (fst\ c - fst\ b)$   
**by** *(simp add: algebra-simps)*  
**finally have**  $fst\ a = fst\ b + (snd\ a - snd\ b) / (snd\ c - snd\ b) * (fst\ c - fst\ b)$   
**by** *simp*  
**moreover**  
**hence**  $snd\ a = snd\ b + (snd\ a - snd\ b) / (snd\ c - snd\ b) * (snd\ c - snd\ b)$   
**using** *neq by simp*  
**ultimately have**  $a = b + ((snd\ a - snd\ b) / (snd\ c - snd\ b)) *_R (c - b)$   
**by** *(auto simp: prod-eq-iff)*  
**thus** *?thesis by blast*  
**qed**  
**qed**

**lemma** *cramer:  $\neg det3\ s\ t\ q = 0 \implies (det3\ t\ p\ r) = ((det3\ t\ q\ r) * (det3\ s\ t\ p) + (det3\ t\ p\ q) * (det3\ s\ t\ r)) / (det3\ s\ t\ q)$*   
**by** *(auto simp: det3-def' field-simps)*

**lemma** *convex-comb-dets:*  
**assumes**  $det3\ p\ q\ r > 0$

**shows**  $s = (\det3\ s\ q\ r / \det3\ p\ q\ r) *_{\mathbb{R}} p + (\det3\ p\ s\ r / \det3\ p\ q\ r) *_{\mathbb{R}} q +$   
 $(\det3\ p\ q\ s / \det3\ p\ q\ r) *_{\mathbb{R}} r$   
**(is ?lhs = ?rhs)**

**proof** –  
**from** *assms* **have**  $\det3\ p\ q\ r *_{\mathbb{R}} ?lhs = \det3\ p\ q\ r *_{\mathbb{R}} ?rhs$   
**by** (*simp add: field-simps prod-eq-iff scaleR-add-right*) (*simp add: algebra-simps det3-def'*)  
**thus** *?thesis* **using** *assms* **by** *simp*

**qed**

**lemma** *four-points-aligned*:  
**assumes** *c*:  $\det3\ t\ p\ q = 0$   $\det3\ t\ q\ r = 0$   
**assumes** *distinct*:  $\text{distinct5}\ t\ s\ p\ q\ r$   
**shows**  $\det3\ t\ r\ p = 0$   $\det3\ p\ q\ r = 0$

**proof** –  
**from** *distinct* **have**  $d: p \neq q \wedge q \neq r$  **by** (*auto*)  
**from** *coll-ex-scaling*[*OF d(1) c(1)*] **obtain** *s1* **where**  $s1: t = p + s1 *_{\mathbb{R}} (q - p)$  **by** *auto*  
**from** *coll-ex-scaling*[*OF d(2) c(2)*] **obtain** *s2* **where**  $s2: t = q + s2 *_{\mathbb{R}} (r - q)$  **by** *auto*  
**from** *distinct s1* **have**  $ne: 1 - s1 \neq 0$  **by** *auto*  
**from** *s1 s2* **have**  $(1 - s1) *_{\mathbb{R}} p = (1 - s1 - s2) *_{\mathbb{R}} q + s2 *_{\mathbb{R}} r$   
**by** (*simp add: algebra-simps*)  
**hence**  $(1 - s1) *_{\mathbb{R}} p /_{\mathbb{R}} (1 - s1) = ((1 - s1 - s2) *_{\mathbb{R}} q + s2 *_{\mathbb{R}} r) /_{\mathbb{R}} (1 - s1)$   
**by** *simp*  
**with** *ne* **have**  $p = ((1 - s1 - s2) / (1 - s1)) *_{\mathbb{R}} q + (s2 / (1 - s1)) *_{\mathbb{R}} r$   
**using** *ne*  
**by** (*simp add: prod-eq-iff inverse-eq-divide add-divide-distrib*)  
**define** *k1* **where**  $k1 = (1 - s1 - s2) / (1 - s1)$   
**define** *k2* **where**  $k2 = s2 / (1 - s1)$   
**have**  $\det3\ t\ r\ p = \det3\ 0\ (k1 *_{\mathbb{R}} q + (k2 - 1) *_{\mathbb{R}} r)$   
 $(k1 *_{\mathbb{R}} q + (k2 - 1) *_{\mathbb{R}} r + (-s1 * (k1 - 1)) *_{\mathbb{R}} q - (s1 * k2) *_{\mathbb{R}} r)$   
**unfolding** *s1 p k1-def*[*symmetric*] *k2-def*[*symmetric*]  
**by** (*simp add: algebra-simps det3-def'*)  
**also** **have**  $-s1 * (k1 - 1) = s1 * k2$   
**using** *ne* **by** (*auto simp: k1-def field-simps k2-def*)  
**also**  
**have**  $1 - k1 = k2$   
**using** *ne*  
**by** (*auto simp: k2-def k1-def field-simps*)  
**have** *k21*:  $k2 - 1 = -k1$   
**using** *ne*  
**by** (*auto simp: k2-def k1-def field-simps*)  
**finally** **have**  $\det3\ t\ r\ p = \det3\ 0\ (k1 *_{\mathbb{R}} (q - r)) ((k1 + (s1 * k2)) *_{\mathbb{R}} (q - r))$   
**by** (*auto simp: algebra-simps*)  
**also** **have**  $\dots = 0$   
**by** (*simp add: algebra-simps det3-def'*)  
**finally** **show**  $\det3\ t\ r\ p = 0$  .

**have**  $\det3\ p\ q\ r = \det3\ (k1\ *_R\ q + k2\ *_R\ r)\ q\ r$   
**unfolding**  $p\ k1\text{-def}[\textit{symmetric}]\ k2\text{-def}[\textit{symmetric}]\ ..$   
**also have**  $\dots = \det3\ 0\ (r - q)\ (k1\ *_R\ q + (-k1)\ *_R\ r)$   
**unfolding**  $k21[\textit{symmetric}]$   
**by**  $(\textit{auto simp: algebra-simps det3-def'})$   
**also have**  $\dots = \det3\ 0\ (r - q)\ (-k1\ *_R\ (r - q))$   
**by**  $(\textit{auto simp: det3-def' algebra-simps})$   
**also have**  $\dots = 0$   
**by**  $(\textit{auto simp: det3-def'})$   
**finally show**  $\det3\ p\ q\ r = 0$  .  
**qed**

**lemma**  $\textit{det-identity}$ :

$\det3\ t\ p\ q * \det3\ t\ s\ r + \det3\ t\ q\ r * \det3\ t\ s\ p + \det3\ t\ r\ p * \det3\ t\ s\ q = 0$   
**by**  $(\textit{auto simp: det3-def' algebra-simps})$

**lemma**  $\textit{det3-eq-zeroI}$ :

**assumes**  $p = q + x *_R (t - q)$   
**shows**  $\det3\ q\ t\ p = 0$   
**unfolding**  $\textit{assms}$   
**by**  $(\textit{auto simp: det3-def' algebra-simps})$

**lemma**  $\textit{det3-rotate}$ :  $\det3\ a\ b\ c = \det3\ c\ a\ b$

**by**  $(\textit{auto simp: det3-def'})$

**lemma**  $\textit{det3-switch}$ :  $\det3\ a\ b\ c = - \det3\ a\ c\ b$

**by**  $(\textit{auto simp: det3-def'})$

**lemma**  $\textit{det3-switch'}$ :  $\det3\ a\ b\ c = - \det3\ b\ a\ c$

**by**  $(\textit{auto simp: det3-def'})$

**lemma**  $\textit{det3-pos-transitive-coll}$ :

$\det3\ t\ s\ p > 0 \implies \det3\ t\ s\ r \geq 0 \implies \det3\ t\ p\ q \geq 0 \implies$

$\det3\ t\ q\ r > 0 \implies \det3\ t\ s\ q = 0 \implies \det3\ t\ p\ r > 0$

**using**  $\textit{det-identity}[\textit{of } t\ p\ q\ s\ r]$

**by**  $(\textit{metis add.commute add-less-same-cancel1 det3-switch det3-switch' less-eq-real-def less-not-sym monoid-add-class.add.left-neutral mult-pos-pos mult-zero-left mult-zero-right})$

**lemma**  $\textit{det3-pos-transitive}$ :

$\det3\ t\ s\ p > 0 \implies \det3\ t\ s\ q \geq 0 \implies \det3\ t\ s\ r \geq 0 \implies \det3\ t\ p\ q \geq 0 \implies$

$\det3\ t\ q\ r > 0 \implies \det3\ t\ p\ r > 0$

**apply**  $(\textit{cases } \det3\ t\ s\ q \neq 0)$

**using**  $\textit{cramer}[\textit{of } q\ t\ s\ p\ r]$

**apply**  $(\textit{force simp: det3-rotate}[\textit{of } q\ t\ p]\ \textit{det3-rotate}[\textit{of } p\ q\ t]\ \textit{det3-switch}[\textit{of } t\ p\ s]$

$\textit{det3-switch'}[\textit{of } q\ t\ r]\ \textit{det3-rotate}[\textit{of } q\ t\ s]\ \textit{det3-rotate}[\textit{of } s\ q\ t]$

$\textit{intro!}:\ \textit{divide-pos-pos add-nonneg-pos})$

**apply**  $(\textit{metis det3-pos-transitive-coll})$

**done**

**lemma** *det3-zero-translate-plus[simp]*:  $\det3 (a + x) (b + x) (c + x) = 0 \iff \det3 a b c = 0$

**by** (*auto simp: algebra-simps det3-def'*)

**lemma** *det3-zero-translate-plus'[simp]*:  $\det3 (a) (a + b) (a + c) = 0 \iff \det3 0 b c = 0$

**by** (*auto simp: algebra-simps det3-def'*)

**lemma**

*det30-zero-scaleR1*:

$0 < e \implies \det3 0 x r P = 0 \implies \det3 0 (e *_{\mathbb{R}} x r) P = 0$

**by** (*auto simp: zero-prod-def algebra-simps det3-def'*)

**lemma** *det3-same[simp]*:  $\det3 a x x = 0$

**by** (*auto simp: det3-def'*)

**lemma**

*det30-zero-scaleR2*:

$0 < e \implies \det3 0 P x r = 0 \implies \det3 0 P (e *_{\mathbb{R}} x r) = 0$

**by** (*auto simp: zero-prod-def algebra-simps det3-def'*)

**lemma** *det3-eq-zero*:  $e \neq 0 \implies \det3 0 x r (e *_{\mathbb{R}} Q) = 0 \iff \det3 0 x r Q = 0$

**by** (*auto simp: det3-def'*)

**lemma** *det30-plus-scaled3[simp]*:  $\det3 0 a (b + x *_{\mathbb{R}} a) = 0 \iff \det3 0 a b = 0$

**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *det30-plus-scaled2[simp]*:

**shows**  $\det3 0 (a + x *_{\mathbb{R}} a) b = 0 \iff (\text{if } x = -1 \text{ then True else } \det3 0 a b = 0)$

(*is ?lhs = ?rhs*)

**proof**

**assume**  $\det3 0 (a + x *_{\mathbb{R}} a) b = 0$

**hence**  $\text{fst } a * \text{snd } b * (1 + x) = \text{fst } b * \text{snd } a * (1 + x)$

**by** (*simp add: algebra-simps det3-def'*)

**thus** *?rhs*

**by** (*auto simp add: det3-def'*)

**qed** (*auto simp: det3-def' algebra-simps split: if-split-asm*)

**lemma** *det30-uminus2[simp]*:  $\det3 0 (-a) (b) = 0 \iff \det3 0 a b = 0$

**and** *det30-uminus3[simp]*:  $\det3 0 a (-b) = 0 \iff \det3 0 a b = 0$

**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *det30-minus-scaled3[simp]*:  $\det3 0 a (b - x *_{\mathbb{R}} a) = 0 \iff \det3 0 a b = 0$

**using** *det30-plus-scaled3[of a b -x]* **by** *simp*

**lemma** *det30-scaled-minus3[simp]*:  $\det3 0 a (e *_{\mathbb{R}} a - b) = 0 \iff \det3 0 a b = 0$

**using** *det30-plus-scaled3[of a -b e]*

**by** (*simp add: algebra-simps*)

**lemma** *det30-minus-scaled2*[simp]:

$\text{det3 } 0 (a - x *_R a) b = 0 \longleftrightarrow (\text{if } x = 1 \text{ then True else } \text{det3 } 0 a b = 0)$   
using *det30-plus-scaled2*[of  $a -x b$ ] by *simp*

**lemma** *det3-nonneg-scaleR1*:

$0 < e \implies \text{det3 } 0 xr P \geq 0 \implies \text{det3 } 0 (e *_R xr) P \geq 0$   
by (*auto simp add: det3-def' algebra-simps*)

**lemma** *det3-nonneg-scaleR1-eq*:

$0 < e \implies \text{det3 } 0 (e *_R xr) P \geq 0 \longleftrightarrow \text{det3 } 0 xr P \geq 0$   
by (*auto simp add: det3-def' algebra-simps*)

**lemma** *det3-translate-origin*: *NO-MATCH*  $0 p \implies \text{det3 } p q r = \text{det3 } 0 (q - p) (r - p)$

by (*auto simp: det3-def' algebra-simps*)

**lemma** *det3-nonneg-scaleR-segment2*:

assumes  $\text{det3 } x y z \geq 0$   
assumes  $a > 0$   
shows  $\text{det3 } x ((1 - a) *_R x + a *_R y) z \geq 0$

**proof** -

from *assms* have  $0 \leq \text{det3 } 0 (a *_R (y - x)) (z - x)$

by (*intro det3-nonneg-scaleR1*) (*simp-all add: det3-translate-origin*)

thus *?thesis*

by (*simp add: algebra-simps det3-translate-origin*)

**qed**

**lemma** *det3-nonneg-scaleR-segment1*:

assumes  $\text{det3 } x y z \geq 0$   
assumes  $0 \leq a < 1$   
shows  $\text{det3 } ((1 - a) *_R x + a *_R y) y z \geq 0$

**proof** -

from *assms* have  $\text{det3 } 0 ((1 - a) *_R (y - x)) (z - x + (-a) *_R (y - x)) \geq 0$

by (*subst det3-nonneg-scaleR1-eq*) (*auto simp add: det3-def' algebra-simps*)

thus *?thesis*

by (*auto simp: algebra-simps det3-translate-origin*)

**qed**

## 8.2 Strict CCW Predicate

**definition**  $\text{ccw}' p q r \longleftrightarrow 0 < \text{det3 } p q r$

**interpretation** *ccw'*: *ccw-vector-space ccw'*

by *unfold-locales* (*auto simp: ccw'-def det3-def' algebra-simps*)

**interpretation** *ccw'*: *linorder-list0 ccw' x for x* .

**lemma** *ccw'-contra*:  $\text{ccw}' t r q \implies \text{ccw}' t q r = \text{False}$

by (auto simp: ccw'-def det3-def' algebra-simps)

**lemma not-ccw'-eq:**  $\neg ccw' t p s \longleftrightarrow ccw' t s p \vee det3 t s p = 0$   
 by (auto simp: ccw'-def det3-def' algebra-simps)

**lemma neq-left-right-of:**  $ccw' a b c \implies ccw' a c d \implies b \neq d$   
 by (auto simp: ccw'-def det3-def' algebra-simps)

**lemma ccw'-subst-collinear:**

assumes  $det3 t r s = 0$

assumes  $s \neq t$

assumes  $ccw' t r p$

shows  $ccw' t s p \vee ccw' t p s$

**proof** cases

assume  $r \neq s$

from *assms* have  $det3 r s t = 0$

by (auto simp: algebra-simps det3-def')

from *coll-ex-scaling*[*OF* *assms*(2)] *this*

**obtain**  $x$  **where**  $s: r = s + x *_R (t - s)$  **by** *auto*

**from** *assms*(3)[*simplified ccw'-def s*]

**have**  $0 < det3 0 (s + x *_R (t - s) - t) (p - t)$

by (auto simp: algebra-simps det3-def')

**also have**  $s + x *_R (t - s) - t = (1 - x) *_R (s - t)$

by (*simp add: algebra-simps*)

**finally have**  $ccw': ccw' 0 ((1 - x) *_R (s - t)) (p - t)$

by (*simp add: ccw'-def*)

**hence**  $x \neq 1$  **by** (*auto simp add: det3-def' ccw'-def*)

{

assume  $x < 1$

**hence** *?thesis* **using** *ccw'*

by (*auto simp: not-ccw'-eq ccw'.translate-origin*)

} **moreover** {

assume  $x > 1$

**hence** *?thesis* **using** *ccw'*

by (*auto simp: not-ccw'-eq ccw'.translate-origin*)

} **ultimately show** *?thesis* **using**  $\langle x \neq 1 \rangle$  **by** *arith*

**qed** (*insert assms, simp*)

**lemma ccw'-sorted-scaleR:**  $ccw'.sortedP 0 xs \implies r > 0 \implies ccw'.sortedP 0 (map ((*_R) r) xs)$

by (*induct xs*) (*auto intro!: ccw'.sortedP.Cons elim!: ccw'.sortedP-Cons simp del: scaleR-Pair*)

### 8.3 Collinearity

**abbreviation**  $coll a b c \equiv det3 a b c = 0$

**lemma coll-zero**[*intro, simp*]:  $coll 0 z 0$

by (*auto simp: det3-def'*)

**lemma** *coll-zero1*[*intro, simp*]:  $\text{coll } 0 \ 0 \ z$   
**by** (*auto simp: det3-def'*)

**lemma** *coll-self*[*intro, simp*]:  $\text{coll } 0 \ z \ z$   
**by** *auto*

**lemma** *ccw'-not-coll*:  
 $\text{ccw}' \ a \ b \ c \implies \neg \text{coll } a \ b \ c$   
 $\text{ccw}' \ a \ b \ c \implies \neg \text{coll } a \ c \ b$   
 $\text{ccw}' \ a \ b \ c \implies \neg \text{coll } b \ a \ c$   
 $\text{ccw}' \ a \ b \ c \implies \neg \text{coll } b \ c \ a$   
 $\text{ccw}' \ a \ b \ c \implies \neg \text{coll } c \ a \ b$   
 $\text{ccw}' \ a \ b \ c \implies \neg \text{coll } c \ b \ a$   
**by** (*auto simp: det3-def' ccw'-def algebra-simps*)

**lemma** *coll-add*:  $\text{coll } 0 \ x \ y \implies \text{coll } 0 \ x \ z \implies \text{coll } 0 \ x \ (y + z)$   
**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *coll-scaleR-left-eq*[*simp*]:  $\text{coll } 0 \ (r *_R x) \ y \longleftrightarrow r = 0 \vee \text{coll } 0 \ x \ y$   
**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *coll-scaleR-right-eq*[*simp*]:  $\text{coll } 0 \ y \ (r *_R x) \longleftrightarrow r = 0 \vee \text{coll } 0 \ y \ x$   
**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *coll-scaleR*:  $\text{coll } 0 \ x \ y \implies \text{coll } 0 \ (r *_R x) \ y$   
**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *coll-sum-list*:  $(\bigwedge y. y \in \text{set } ys \implies \text{coll } 0 \ x \ y) \implies \text{coll } 0 \ x \ (\text{sum-list } ys)$   
**by** (*induct ys*) (*auto intro!: coll-add*)

**lemma** *scaleR-left-normalize*:  
**fixes**  $a :: \text{real}$  **and**  $b \ c :: 'a :: \text{real-vector}$   
**shows**  $a *_R b = c \longleftrightarrow (\text{if } a = 0 \text{ then } c = 0 \text{ else } b = c /_R a)$   
**by** (*auto simp: field-simps*)

**lemma** *coll-scale-pair*:  $\text{coll } 0 \ (a, b) \ (c, d) \implies (a, b) \neq 0 \implies (\exists x. (c, d) = x *_R (a, b))$   
**by** (*auto intro: exI[where x=c/a] exI[where x=d/b] simp: det3-def' field-simps prod-eq-iff*)

**lemma** *coll-scale*:  $\text{coll } 0 \ r \ q \implies r \neq 0 \implies (\exists x. q = x *_R r)$   
**using** *coll-scale-pair*[*of fst r snd r fst q snd q*]  
**by** *simp*

**lemma** *coll-add-trans*:  
**assumes**  $\text{coll } 0 \ x \ (y + z)$   
**assumes**  $\text{coll } 0 \ y \ z$   
**assumes**  $x \neq 0$

```

assumes  $y \neq 0$ 
assumes  $z \neq 0$ 
assumes  $y + z \neq 0$ 
shows  $\text{coll } 0 \ x \ z$ 
proof (cases  $\text{snd } z = 0$ )
  case True
    hence  $\text{snd } y = 0$ 
      using assms
      by (cases  $z$ ) (auto simp add: zero-prod-def det3-def')
    with True assms have  $\text{snd } x = 0$ 
      by (cases  $y$ , cases  $z$ ) (auto simp add: zero-prod-def det3-def')
    from  $\langle \text{snd } x = 0 \rangle \langle \text{snd } y = 0 \rangle \langle \text{snd } z = 0 \rangle$ 
    show ?thesis
      by (auto simp add: zero-prod-def det3-def')
  next
    case False
    note  $z = \text{False}$ 
    hence  $\text{snd } y \neq 0$ 
      using assms
      by (cases  $y$ ) (auto simp add: zero-prod-def det3-def')
    with False assms have  $\text{snd } x \neq 0$ 
      apply (cases  $x$ )
      apply (cases  $y$ )
      apply (cases  $z$ )
      apply (auto simp add: zero-prod-def det3-def')
      apply (metis mult.commute mult-eq-0-iff ring-class.ring-distrib(1))
    done
    with False assms  $\langle \text{snd } y \neq 0 \rangle$  have  $yz: \text{snd } (y + z) \neq 0$ 
      by (cases  $x$ ; cases  $y$ ; cases  $z$ ) (auto simp add: det3-def' zero-prod-def)
    from coll-scale[OF assms(1) assms(3)] coll-scale[OF assms(2) assms(4)]
    obtain  $r \ s$  where  $rs: y + z = r *_R x \ z = s *_R y$ 
      by auto
    with  $z$  have  $s \neq 0$ 
      by (cases  $x$ ; cases  $y$ ; cases  $z$ ) (auto simp: zero-prod-def)
    with  $rs \ z \ yz$  have  $r \neq 0$ 
      by (cases  $x$ ; cases  $y$ ; cases  $z$ ) (auto simp: zero-prod-def)
    from  $\langle s \neq 0 \rangle \ rs$  have  $y = r *_R x - z \ y = z /_R s$ 
      by (auto simp: inverse-eq-divide algebra-simps)
    hence  $r *_R x - z = z /_R s$  by simp
    hence  $r *_R x = (1 + \text{inverse } s) *_R z$ 
      by (auto simp: inverse-eq-divide algebra-simps)
    hence  $x = (\text{inverse } r * (1 + \text{inverse } s)) *_R z$ 
      using  $\langle r \neq 0 \rangle \langle s \neq 0 \rangle$ 
      by (auto simp: field-simps scaleR-left-normalize)
    from this
    show ?thesis
      by (auto intro: coll-scaleR)
qed

```



**lemma** *coll-commute*:  $coll\ 0\ a\ b \longleftrightarrow coll\ 0\ b\ a$   
**by** (*metis det3-rotate det3-switch' diff-0 diff-self*)

**lemma** *coll-add-cancel*:  $coll\ 0\ a\ (a + b) \implies coll\ 0\ a\ b$   
**by** (*cases a, cases b*) (*auto simp: det3-def' algebra-simps*)

**lemma** *coll-trans*:  
 $coll\ 0\ a\ b \implies coll\ 0\ a\ c \implies a \neq 0 \implies coll\ 0\ b\ c$   
**by** (*metis coll-scale coll-scaleR*)

**lemma** *sum-list-posI*:  
**fixes**  $xs::'a::ordered-comm-monoid-add\ list$   
**shows**  $(\bigwedge x. x \in set\ xs \implies x > 0) \implies xs \neq [] \implies sum-list\ xs > 0$   
**proof** (*induct xs*)  
**case** (*Cons x xs*)  
**thus** ?*case*  
**by** (*cases xs = []*) (*auto intro!: add-pos-pos*)  
**qed** *simp*

**lemma** *nonzero-fstI*[*intro, simp*]:  $fst\ x \neq 0 \implies x \neq 0$   
**and** *nonzero-sndI*[*intro, simp*]:  $snd\ x \neq 0 \implies x \neq 0$   
**by** *auto*

**lemma** *coll-sum-list-trans*:  
 $xs \neq [] \implies coll\ 0\ a\ (sum-list\ xs) \implies (\bigwedge x. x \in set\ xs \implies coll\ 0\ x\ y) \implies$   
 $(\bigwedge x. x \in set\ xs \implies coll\ 0\ x\ (sum-list\ xs)) \implies$   
 $(\bigwedge x. x \in set\ xs \implies snd\ x > 0) \implies a \neq 0 \implies coll\ 0\ a\ y$   
**proof** (*induct xs rule: list-nonempty-induct*)  
**case** (*single x*)  
**from** *single(1) single(2)[of x] single(4)[of x]* **have**  $coll\ 0\ x\ a\ coll\ 0\ x\ y\ x \neq 0$   
**by** (*auto simp: coll-commute*)  
**thus** ?*case* **by** (*rule coll-trans*)  
**next**  
**case** (*cons x xs*)  
**from** *cons(5)[of x] <a ≠ 0> cons(6)[of x]*  
**have**  $*$ :  $coll\ 0\ x\ (sum-list\ xs)\ a \neq 0\ x \neq 0$  **by** (*force simp add: coll-add-cancel*)  
**have**  $0 < snd\ (sum-list\ (x\#\ xs))$   
**unfolding** *snd-sum-list*  
**by** (*rule sum-list-posI*) (*auto intro!: add-pos-pos cons simp: snd-sum-list*)  
**hence**  $x + sum-list\ xs \neq 0$  **by** *simp*  
**from** *coll-add-trans[OF cons(3)[simplified] \* - this]*  
**have** *cH*:  $coll\ 0\ a\ (sum-list\ xs)$   
**by** (*cases sum-list xs = 0*) *auto*  
**from** *cons(4)* **have** *cy*:  $(\bigwedge x. x \in set\ xs \implies coll\ 0\ x\ y)$  **by** *simp*  
**{**  
**fix** *y* **assume**  $y \in set\ xs$   
**hence**  $snd\ (sum-list\ xs) > 0$   
**unfolding** *snd-sum-list*  
**by** (*intro sum-list-posI*) (*auto intro!: add-pos-pos cons simp: snd-sum-list*)  
**}**

```

hence sum-list xs ≠ 0 by simp
from cons(5)[of x] have coll 0 x (sum-list xs)
  by (simp add: coll-add-cancel)
from cons(5)[of y]
have coll 0 y (sum-list xs)
  using ⟨y ∈ set xs⟩ cons(6)[of y] ⟨x + sum-list xs ≠ 0⟩
  apply (cases y = x)
  subgoal by (force simp add: coll-add-cancel)
  subgoal by (force simp: dest!: coll-add-trans[OF - *(1) - *(3)])
  done
} note cl = this
show ?case
  by (rule cons(2)[OF cH cy cl cons(6) ⟨a ≠ 0⟩]) auto
qed

```

```

lemma sum-list-coll-ex-scale:
  assumes coll:  $\bigwedge x. x \in \text{set } xs \implies \text{coll } 0 \ z \ x$ 
  assumes nz:  $z \neq 0$ 
  shows  $\exists r. \text{sum-list } xs = r \ *_R \ z$ 
proof -
  {
    fix i assume i:  $i < \text{length } xs$ 
    hence nth:  $xs ! i \in \text{set } xs$  by simp
    note coll-scale[OF coll[OF nth] ⟨ $z \neq 0$ ⟩]
  } then obtain r where  $r: \bigwedge i. i < \text{length } xs \implies r \ i \ *_R \ z = xs ! i$ 
    by metis
  have  $xs = \text{map } (!) \ xs \ [0..<\text{length } xs]$  by (simp add: map-nth)
  also have  $\dots = \text{map } (\lambda i. r \ i \ *_R \ z) \ [0..<\text{length } xs]$ 
    by (auto simp: r)
  also have  $\text{sum-list } \dots = (\sum i \leftarrow [0..<\text{length } xs]. r \ i) \ *_R \ z$ 
    by (simp add: sum-list-sum-nth scaleR-sum-left)
  finally show ?thesis ..
qed

```

```

lemma sum-list-filter-coll-ex-scale:  $z \neq 0 \implies \exists r. \text{sum-list } (\text{filter } (\text{coll } 0 \ z) \ zs) =$ 
   $r \ *_R \ z$ 
  by (rule sum-list-coll-ex-scale) simp

```

```

end
theory Polygon
imports Counterclockwise-2D-Strict
begin

```

## 8.4 Polygonal chains

```

definition polychain  $xs = (\forall i. \text{Suc } i < \text{length } xs \longrightarrow \text{snd } (xs ! i) = (\text{fst } (xs ! \text{Suc } i)))$ 

```

```

lemma polychainI:

```

**assumes**  $\bigwedge i. \text{Suc } i < \text{length } xs \implies \text{snd } (xs ! i) = \text{fst } (xs ! \text{Suc } i)$   
**shows** *polychain*  $xs$   
**by** (*auto intro!*: *assms simp: polychain-def*)

**lemma** *polychain-Nil*[*simp*]: *polychain*  $[] = \text{True}$   
**and** *polychain-singleton*[*simp*]: *polychain*  $[x] = \text{True}$   
**by** (*auto simp: polychain-def*)

**lemma** *polychain-Cons*:  
*polychain*  $(y \# ys) = (\text{if } ys = [] \text{ then } \text{True} \text{ else } \text{snd } y = \text{fst } (ys ! 0) \wedge \text{polychain } ys)$   
**by** (*auto simp: polychain-def nth-Cons split: nat.split*)

**lemma** *polychain-appendI*:  
*polychain*  $xs \implies \text{polychain } ys \implies (xs \neq [] \implies ys \neq [] \implies \text{snd } (\text{last } xs) = \text{fst } (\text{hd } ys)) \implies$   
*polychain*  $(xs @ ys)$   
**by** (*induct xs arbitrary: ys*)  
*(auto simp add: polychain-Cons nth-append hd-conv-nth split: if-split-asm)*

**fun** *pairself* **where** *pairself*  $f (x, y) = (f x, f y)$

**lemma** *pairself-apply*: *pairself*  $f x = (f (\text{fst } x), f (\text{snd } x))$   
**by** (*cases x, simp*)

**lemma** *polychain-map-pairself*: *polychain*  $xs \implies \text{polychain } (\text{map } (\text{pairself } f) xs)$   
**by** (*auto simp: polychain-def pairself-apply*)

**definition** *convex-polychain*  $xs \longleftrightarrow$   
*(polychain*  $xs \wedge$   
 $(\forall i. \text{Suc } i < \text{length } xs \longrightarrow \text{det3 } (\text{fst } (xs ! i)) (\text{snd } (xs ! i)) (\text{snd } (xs ! \text{Suc } i)) > 0))$ )

**lemma** *convex-polychain-Cons2*[*simp*]:  
*convex-polychain*  $(x \# y \# zs) \longleftrightarrow$   
 $\text{snd } x = \text{fst } y \wedge \text{det3 } (\text{fst } x) (\text{fst } y) (\text{snd } y) > 0 \wedge \text{convex-polychain } (y \# zs)$   
**by** (*auto simp add: convex-polychain-def polychain-def nth-Cons split: nat.split*)

**lemma** *convex-polychain-ConsD*:  
**assumes** *convex-polychain*  $(x \# xs)$   
**shows** *convex-polychain*  $xs$   
**using** *assms* **by** (*auto simp: convex-polychain-def polychain-def nth-Cons split: nat.split*)

**definition**  
*convex-polygon*  $xs \longleftrightarrow (\text{convex-polychain } xs \wedge (xs \neq [] \longrightarrow \text{fst } (\text{hd } xs) = \text{snd } (\text{last } xs)))$

**lemma** *convex-polychain-Nil*[*simp*]: *convex-polychain*  $[] = \text{True}$

```

and convex-polychain-Cons[simp]: convex-polychain [x] = True
by (auto simp: convex-polychain-def)

lemma convex-polygon-Cons2[simp]:
  convex-polygon (x#y#zs)  $\longleftrightarrow$  fst x = snd (last (y#zs))  $\wedge$  convex-polychain
  (x#y#zs)
  by (auto simp: convex-polygon-def convex-polychain-def polychain-def nth-Cons)

lemma polychain-append-connected:
  polychain (xs @ ys)  $\implies$  xs  $\neq$  []  $\implies$  ys  $\neq$  []  $\implies$  fst (hd ys) = snd (last xs)
  by (auto simp: convex-polychain-def nth-append not-less polychain-def last-conv-nth
  hd-conv-nth
  dest!: spec[where x = length xs - 1])

lemma convex-polychain-appendI:
  assumes cxs: convex-polychain xs
  assumes cys: convex-polychain ys
  assumes pxy: polychain (xs @ ys)
  assumes xs  $\neq$  []  $\implies$  ys  $\neq$  []  $\implies$  det3 (fst (last xs)) (snd (last xs)) (snd (hd ys))
  > 0
  shows convex-polychain (xs @ ys)
proof -
  {
    fix i
    assume i < length xs length xs  $\leq$  Suc i Suc i < length xs + length ys
    hence xs  $\neq$  [] ys  $\neq$  [] i = length xs - 1 by auto
  }
  thus ?thesis
  using assms
  by (auto simp: hd-conv-nth convex-polychain-def nth-append Suc-diff-le last-conv-nth
  )
qed

lemma convex-polychainI:
  assumes polychain xs
  assumes  $\bigwedge i. \text{Suc } i < \text{length } xs \implies \text{det3 } (\text{fst } (xs ! i)) (\text{snd } (xs ! i)) (\text{snd } (xs ! \text{Suc } i)) > 0$ 
  shows convex-polychain xs
  by (auto intro!: assms simp: convex-polychain-def ccw'-def)

lemma convex-polygon-skip:
  assumes convex-polygon (x # y # z # w # ws)
  assumes ccw'.sortedP (fst x) (map snd (butlast (x # y # z # w # ws)))
  shows convex-polygon ((fst x, snd y) # z # w # ws)
  using assms by (auto elim!: ccw'.sortedP-Cons simp: ccw'-def[symmetric])

primrec polychain-of::'a::ab-group-add  $\Rightarrow$  'a list  $\Rightarrow$  ('a*'a) list where
  polychain-of xc [] = []

```

|  $\text{polychain-of } xc \ (xm\#xs) = (xc, xc + xm)\#\text{polychain-of } (xc + xm) \ xs$

**lemma** *in-set-polychain-ofD*:  $ab \in \text{set } (\text{polychain-of } x \ xs) \implies (\text{snd } ab - \text{fst } ab) \in \text{set } xs$

**by** (*induct xs arbitrary: x*) *auto*

**lemma** *fst-polychain-of-nth-0[simp]*:  $xs \neq [] \implies \text{fst } ((\text{polychain-of } p \ xs) ! 0) = p$   
**by** (*cases xs*) (*auto simp: Let-def*)

**lemma** *fst-hd-polychain-of*:  $xs \neq [] \implies \text{fst } (\text{hd } (\text{polychain-of } x \ xs)) = x$   
**by** (*cases xs*) *auto*

**lemma** *length-polychain-of-eq[simp]*:  
**shows**  $\text{length } (\text{polychain-of } p \ qs) = \text{length } qs$   
**by** (*induct qs arbitrary: p*) *simp-all*

**lemma**  
*polychain-of-subsequent-eq*:  
**assumes**  $\text{Suc } i < \text{length } qs$   
**shows**  $\text{snd } (\text{polychain-of } p \ qs ! i) = \text{fst } (\text{polychain-of } p \ qs ! \text{Suc } i)$   
**using** *assms*  
**by** (*induct qs arbitrary: p i*) (*auto simp add: nth-Cons split: nat.split*)

**lemma** *polychain-of-eq-empty-iff[simp]*:  $\text{polychain-of } p \ xs = [] \longleftrightarrow xs = []$   
**by** (*cases xs*) (*auto simp: Let-def*)

**lemma** *in-set-polychain-of-imp-sum-list*:  
**assumes**  $z \in \text{set } (\text{polychain-of } Pc \ Ps)$   
**obtains**  $d$  **where**  $z = (Pc + \text{sum-list } (\text{take } d \ Ps), Pc + \text{sum-list } (\text{take } (\text{Suc } d) \ Ps))$   
**using** *assms*  
**apply** *atomize-elim*  
**proof** (*induction Ps arbitrary: Pc z*)  
**case Nil** **thus** *?case by simp*  
**next**  
**case** (*Cons P Ps*)  
**hence**  $z = (Pc, Pc + P) \vee z \in \text{set } (\text{polychain-of } (Pc + P) \ Ps)$   
**by** *auto*  
**thus** *?case*  
**proof**  
**assume**  $z \in \text{set } ((\text{polychain-of } (Pc + P) \ Ps))$   
**from** *Cons.IH[OF this]*  
**obtain**  $d$   
**where**  $z = (Pc + P + \text{sum-list } (\text{take } d \ Ps), Pc + P + \text{sum-list } (\text{take } (\text{Suc } d) \ Ps))$   
**by** *auto*  
**thus** *?case*  
**by** (*auto intro!: exI[where x=Suc d]*)  
**qed** (*auto intro!: exI[where x=0]*)

qed

**lemma** *last-polychain-of*:  $\text{length } xs > 0 \implies \text{snd } (\text{last } (\text{polychain-of } p \ xs)) = p + \text{sum-list } xs$   
by (induct xs arbitrary: p) simp-all

**lemma** *polychain-of-singleton-iff*:  $\text{polychain-of } p \ xs = [a] \iff \text{fst } a = p \wedge xs = [(\text{snd } a - p)]$   
by (induct xs) auto

**lemma** *polychain-of-add*:  $\text{polychain-of } (x + y) \ xs = \text{map } (((+) (y, y))) (\text{polychain-of } x \ xs)$   
by (induct xs arbitrary: x y) (auto simp: algebra-simps)

## 8.5 Dirvec: Inverse of Polychain

**primrec** *dirvec* where  $\text{dirvec } (x, y) = (y - x)$

**lemma** *dirvec-minus*:  $\text{dirvec } x = \text{snd } x - \text{fst } x$   
by (cases x) simp

**lemma** *dirvec-nth-polychain-of*:  $n < \text{length } xs \implies \text{dirvec } ((\text{polychain-of } p \ xs) ! n) = (xs ! n)$   
by (induct xs arbitrary: p n) (auto simp: nth-Cons split: nat.split)

**lemma** *dirvec-hd-polychain-of*:  $xs \neq [] \implies \text{dirvec } (\text{hd } (\text{polychain-of } p \ xs)) = (\text{hd } xs)$   
by (simp add: hd-conv-nth dirvec-nth-polychain-of)

**lemma** *dirvec-last-polychain-of*:  $xs \neq [] \implies \text{dirvec } (\text{last } (\text{polychain-of } p \ xs)) = (\text{last } xs)$   
by (simp add: last-conv-nth dirvec-nth-polychain-of)

**lemma** *map-dirvec-polychain-of*[simp]:  $\text{map } \text{dirvec } (\text{polychain-of } x \ xs) = xs$   
by (induct xs arbitrary: x) simp-all

## 8.6 Polychain of Sorted (*polychain-of*, *ccw'.sortedP*)

**lemma** *ccw'-sortedP-translateD*:  
 $\text{linorder-list0.sortedP } (ccw' \ x0) (\text{map } ((+) \ x \circ g) \ xs) \implies$   
 $\text{linorder-list0.sortedP } (ccw' (x0 - x)) (\text{map } g \ xs)$

**proof** (induct xs arbitrary: x0 x)

case Nil thus ?case by (auto simp: linorder-list0.sortedP.Nil)

next

case (Cons a xs x0 x)

hence  $\forall y \in \text{set } xs. ccw' (x0 - x) (g \ a) (g \ y)$

by (auto elim!: linorder-list0.sortedP-Cons simp: ccw'.translate-origin algebra-simps)

thus ?case

using Cons.premis(1)

**by** (*auto elim!*: *linorder-list0.sortedP-Cons intro!*: *linorder-list0.sortedP.Cons simp: Cons.hyps*)

**qed**

**lemma** *ccw'-sortedP-translateI*:

*linorder-list0.sortedP* (*ccw'* (*x0 - x*)) (*map g xs*)  $\implies$   
*linorder-list0.sortedP* (*ccw'* *x0*) (*map* ((*+*) *x*  $\circ$  *g*) *xs*)  
**using** *ccw'-sortedP-translateD*[*of x0 - x -x (+) x o g xs*]  
**by** (*simp add: o-def*)

**lemma** *ccw'-sortedP-translate-comp*[*simp*]:

*linorder-list0.sortedP* (*ccw'* *x0*) (*map* ((*+*) *x*  $\circ$  *g*) *xs*)  $\longleftrightarrow$   
*linorder-list0.sortedP* (*ccw'* (*x0 - x*)) (*map g xs*)  
**by** (*metis ccw'-sortedP-translateD ccw'-sortedP-translateI*)

**lemma** *snd-plus-commute*: *snd*  $\circ$  (*+*) (*x0*, *x0*) = (*+*) *x0* *o* *snd*

**by** *auto*

**lemma** *ccw'-sortedP-renormalize*:

*ccw'.sortedP* *a* (*map snd* (*polychain-of* (*x0 + x*) *xs*))  $\longleftrightarrow$   
*ccw'.sortedP* (*a - x0*) (*map snd* (*polychain-of* *x* *xs*))  
**by** (*simp add: polychain-of-add add.commute snd-plus-commute*)

**lemma** *ccw'-sortedP-polychain-of01*:

**shows** *ccw'.sortedP* 0 [*u*]  $\implies$  *ccw'.sortedP* *x0* (*map snd* (*polychain-of* *x0* [*u*]))  
**and** *ccw'.sortedP* 0 []  $\implies$  *ccw'.sortedP* *x0* (*map snd* (*polychain-of* *x0* []))  
**by** (*auto intro!*: *linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons simp: ac-simps*)

**lemma** *ccw'-sortedP-polychain-of2*:

**assumes** *ccw'.sortedP* 0 [*u*, *v*]  
**shows** *ccw'.sortedP* *x0* (*map snd* (*polychain-of* *x0* [*u*, *v*]))  
**using** *assms*  
**by** (*auto intro!*: *linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons elim!*: *linorder-list0.sortedP-Cons simp: ac-simps ccw'.translate-origin*)

**lemma** *ccw'-sortedP-polychain-of3*:

**assumes** *ccw'.sortedP* 0 (*u#v#w#xs*)  
**shows** *ccw'.sortedP* *x0* (*map snd* (*polychain-of* *x0* (*u#v#w#xs*)))  
**using** *assms*

**proof** (*induct xs arbitrary: x0 u v w*)

**case** *Nil*

**then have** \*: *ccw'* 0 *u v ccw'* 0 *v w ccw'* 0 *u w*

**by** (*auto intro!*: *linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons elim!*: *linorder-list0.sortedP-Cons simp: ac-simps*)

**moreover have** *ccw'* 0 (*u + v*) (*u + (v + w)*)

**by** (*metis add.assoc ccw'.add1 ccw'.add3-self \*(2-)*)

**ultimately show** ?*case*

**by** (*auto intro!*: *linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons*

*elim!*: *linorder-list0.sortedP-Cons simp: ac-simps ccw'.translate-origin ccw'.add3*)

```

next
  case (Cons y ys)
  have s1: linorder-list0.sortedP (ccw' 0) ((u + v)#w#y#ys) using Cons.premis
    by (auto intro!: linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons
        elim!: linorder-list0.sortedP-Cons simp: ccw'.add1)
  have s2: linorder-list0.sortedP (ccw' 0) (u#(v + w)#y#ys) using Cons.premis
    by (auto intro!: linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons
        elim!: linorder-list0.sortedP-Cons simp: ccw'.add3 ccw'.add1)
  have s3: linorder-list0.sortedP (ccw' 0) (u#v#(w + y)#ys) using Cons.premis
    by (auto intro!: linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons
        elim!: linorder-list0.sortedP-Cons simp: ccw'.add3 ccw'.add1)
  show ?case
    using Cons.hyps[OF s1, of x0] Cons.hyps[OF s2, of x0] Cons.hyps[OF s3, of
x0] Cons.premis
    by (auto intro!: linorder-list0.sortedP.Nil linorder-list0.sortedP.Cons
        elim!: linorder-list0.sortedP-Cons simp: ac-simps)
qed

```

```

lemma ccw'-sortedP-polychain-of-snd:
  assumes ccw'.sortedP 0 xs
  shows ccw'.sortedP x0 (map snd (polychain-of x0 xs))
  using assms
  by (metis ccw'-sortedP-polychain-of01 ccw'-sortedP-polychain-of2 ccw'-sortedP-polychain-of3
      list.exhaust)

```

```

lemma ccw'-sortedP-implies-distinct:
  assumes ccw'.sortedP x qs
  shows distinct qs
  using assms
proof induct
  case Cons thus ?case by (meson ccw'-contra distinct.simps(2))
qed simp

```

```

lemma ccw'-sortedP-implies-nonaligned:
  assumes ccw'.sortedP x qs
  assumes y ∈ set qs z ∈ set qs y ≠ z
  shows ¬ coll x y z
  using assms
  by induct (force simp: ccw'-def det3-def' algebra-simps)+

```

```

lemma list-all-mp: list-all P xs ⇒ (∧x. x ∈ set xs ⇒ P x ⇒ Q x) ⇒ list-all
Q xs
  by (auto simp: list-all-iff)

```

```

lemma
  ccw'-scale-origin:
  assumes e ∈ UNIV → {0<..<1}
  assumes x ∈ set (polychain-of Pc (P # QRRs))
  assumes ccw'.sortedP 0 (P # QRRs)

```



**assumes**  $ccw' (fst\ x) (snd\ x) (P + (Pc + (\sum_{P \in set\ QRRs} e\ P *_R\ P)))$   
**shows**  $ccw' (fst\ x) (snd\ x) (e\ P *_R\ P + (Pc + (\sum_{P \in set\ QRRs} e\ P *_R\ P)))$   
**proof** –  
**from**  $assms(2)$  **have**  $fst\ x = Pc \wedge snd\ x = Pc + P \vee x \in set\ (polychain-of\ (Pc + P)\ QRRs)$  **by**  $auto$   
**thus**  $?thesis$   
**proof**  
**assume**  $x: x \in set\ (polychain-of\ (Pc + P)\ QRRs)$   
**define**  $q$  **where**  $q = snd\ x - fst\ x$   
**from**  $Polygon.in-set-polychain-of-imp-sum-list[OF\ x]$   
**obtain**  $d$  **where**  $d: fst\ x = (Pc + P + sum-list\ (take\ d\ QRRs))$  **by**  $(auto\ simp:\ prod-eq-iff)$   
**from**  $in-set-polychain-ofD[OF\ x]$   
**have**  $q-in: q \in set\ QRRs$  **by**  $(simp\ add: q-def)$   
**define**  $R$  **where**  $R = set\ QRRs - \{q\}$   
**hence**  $QRRs: set\ QRRs = R \cup \{q\}$   $q \notin R$  **finite**  $R$  **using**  $q-in$  **by**  $auto$   
**have**  $ccw' 0\ q\ (-P)$   
**using**  $assms(3)$   
**by**  $(auto\ simp: ccw'.sortedP-Cons-iff\ q-in)$   
**hence**  $ccw' 0\ q\ ((1 - e\ P) *_R\ (-P))$   
**using**  $assms(1)$  **by**  $(subst\ ccw'.scaleR2-eq)\ (auto\ simp: algebra-simps)$   
**moreover**  
**from**  $assms(4)$  **have**  $ccw' 0\ q\ ((\sum_{P \in set\ QRRs} e\ P *_R\ P) - sum-list\ (take\ d\ QRRs))$   
**by**  $(auto\ simp: q-def\ ccw'.translate-origin\ d)$   
**ultimately**  
**have**  $ccw' 0\ q\ ((1 - e\ P) *_R\ (-P) + ((\sum_{P \in set\ QRRs} e\ P *_R\ P) - sum-list\ (take\ d\ QRRs)))$   
**by**  $(intro\ ccw'.add3)\ auto$   
**thus**  $?thesis$   
**by**  $(auto\ simp: ccw'.translate-origin\ q-def\ algebra-simps\ d)$   
**qed**  $(metis\ (no-types,\ lifting)\ add.left-commute\ assms(4)\ ccw'.add3-self\ ccw'.scale-add3\ ccw'.translate)$   
**qed**

**lemma**  $polychain-of-ccw-convex:$

**assumes**  $e \in UNIV \rightarrow \{0 <..< 1\}$   
**assumes**  $sorted: linorder-list0.sortedP\ (ccw' 0)\ (P\#\ Q\#\ Ps)$   
**shows**  $list-all$   
 $(\lambda(xi, xj). ccw' xi\ xj\ (Pc + (\sum_{P \in set\ (P\#\ Q\#\ Ps)} e\ P *_R\ P)))$   
 $(polychain-of\ Pc\ (P\#\ Q\#\ Ps))$   
**using**  $assms(1)\ assms(2)$   
**proof**  $(induct\ Ps\ arbitrary: P\ Q\ Pc)$   
**case**  $Nil$   
**have**  $eq: e\ P *_R\ P + e\ Q *_R\ Q - P = (1 - e\ P) *_R\ (-P) + e\ Q *_R\ Q$   
**using**  $\langle e \in \cdot \rangle$   
**by**  $(auto\ simp\ add: algebra-simps)$   
**from**  $Nil\ ccw'.sortedP-imply-distinct[OF\ Nil(2)]$   
**have**  $P \neq Q\ e\ P < 1\ 0 < e\ Q\ ccw' 0\ P\ Q$

```

    by (auto elim!: linorder-list0.sortedP-Cons)
  thus ?case
    by (auto simp: ccw'-not-coll ccw'.translate-origin eq)
next
  case (Cons R Rs)
  hence ccw' 0 P Q P ≠ Q using ccw'-sortedP-implies-distinct[OF Cons(3)]
    by (auto elim!: linorder-list0.sortedP-Cons)
  have list-all (λ(xi, xj). ccw' xi xj ((Pc + P) + (∑ P∈set (Q # R # Rs). e P
*_R P)))
    (polychain-of (Pc + P) (Q # R # Rs))
    using Cons(2-)
    by (intro Cons(1)) (auto elim!: linorder-list0.sortedP-Cons)
  also have polychain-of (Pc + P) (Q # R # Rs) = tl (polychain-of Pc (P # Q
# R # Rs))
    by simp
  finally have list-all (λ(xi, xj). ccw' xi xj (Pc + P + (∑ P∈set (Q # R # Rs).
e P *_R P)))
    (tl (polychain-of Pc (P # Q # R # Rs))) .
  moreover
  have list-all
    (λ(xi, xj). ccw' xi xj (P + (∑ P∈set (Q # R # Rs). e P *_R P)))
    (polychain-of P (Q # R # Rs))
    using Cons(2-)
    by (intro Cons(1)) (auto elim!: linorder-list0.sortedP-Cons)
  have (λ(xi, xj). ccw' xi xj (Pc + P + (∑ P∈set (Q # R # Rs). e P *_R P)))
    (hd (polychain-of Pc (P # Q # R # Rs)))
    using ccw'-sortedP-implies-nonaligned[OF Cons(3), of P Q]
      ccw'-sortedP-implies-nonaligned[OF Cons(3), of Q R]
      ccw'-sortedP-implies-nonaligned[OF Cons(3), of P R]
      Cons(2,3)
    by (auto simp add: Pi-iff add.assoc simp del: scaleR-Pair intro!: ccw'.sum
elim!: linorder-list0.sortedP-Cons)
  ultimately
  have list-all
    (λ(xi, xj). ccw' xi xj (P + (Pc + (∑ P∈set (Q # R # Rs). e P *_R P))))
    (polychain-of Pc (P # Q # R # Rs))
    by (simp add: ac-simps)
  hence list-all
    (λ(xi, xj). ccw' xi xj (e P *_R P + (Pc + (∑ P∈set (Q # R # Rs). e P *_R
P))))
    (polychain-of Pc (P # Q # R # Rs))
    unfolding split-beta'
    by (rule list-all-mp, intro ccw'-scale-origin[OF assms(1)])
    (auto intro!: ccw'-scale-origin Cons(3))
  thus ?case
    using ccw'-sortedP-implies-distinct[OF Cons(3)] Cons
    by (simp add: ac-simps)
qed

```

**lemma** *polychain-of-ccw*:  
**assumes**  $e \in UNIV \rightarrow \{0 < .. < 1\}$   
**assumes** *sorted*:  $ccw'.sortedP\ 0\ qs$   
**assumes** *qs*:  $length\ qs \neq 1$   
**shows** *list-all*  $(\lambda(xi, xj). ccw'\ xi\ xj\ (Pc + (\sum P \in set\ qs. e\ P *_{R}\ P)))\ (polychain-of\ Pc\ qs)$   
**using** *assms*  
**proof** (*cases qs*)  
**case** (*Cons Q Qs*)  
**note**  $CQ = this$   
**show** *?thesis* **using** *assms*  
**proof** (*cases Qs*)  
**case** (*Cons R Rs*)  
**thus** *?thesis* **using** *assms*  
**unfolding**  $CQ\ Cons$   
**by** (*intro polychain-of-ccw-convex*) (*auto simp: CQ Cons intro!: polychain-of-ccw-convex*)  
**qed** (*auto simp: CQ*)  
**qed** *simp*

**lemma** *in-polychain-of-ccw*:  
**assumes**  $e \in UNIV \rightarrow \{0 < .. < 1\}$   
**assumes**  $ccw'.sortedP\ 0\ qs$   
**assumes**  $length\ qs \neq 1$   
**assumes**  $seg \in set\ (polychain-of\ Pc\ qs)$   
**shows**  $ccw'\ (fst\ seg)\ (snd\ seg)\ (Pc + (\sum P \in set\ qs. e\ P *_{R}\ P))$   
**using** *polychain-of-ccw[OF assms(1,2,3)] assms(4)*  
**by** (*simp add: list-all-iff split-beta*)

**lemma** *distinct-butlast-ne-last*:  $distinct\ xs \implies x \in set\ (butlast\ xs) \implies x \neq last\ xs$   
**by** (*metis append-butlast-last-id distinct-butlast empty-iff in-set-butlastD list.set(1) not-distinct-conv-prefix*)

**lemma**  
*ccw'-sortedP-convex-rotate-aux*:  
**assumes**  $ccw'.sortedP\ 0\ (zs)\ ccw'.sortedP\ x\ (map\ snd\ (polychain-of\ x\ (zs)))$   
**shows**  $ccw'.sortedP\ (snd\ (last\ (polychain-of\ x\ (zs))))\ (map\ snd\ (butlast\ (polychain-of\ x\ (zs))))$   
**using** *assms*  
**proof** (*induct zs arbitrary: x rule: list.induct*)  
**case** (*Cons z zs*)  
**{**  
**assume**  $zs \neq []$   
**have**  $ccw'.sortedP\ (snd\ (last\ (polychain-of\ (x + z)\ zs)))$   
 $(map\ snd\ (butlast\ (polychain-of\ (x + z)\ zs)))$   
**using** *Cons.prem*  
**by** (*auto elim!: ccw'.sortedP-Cons intro!: ccw'-sortedP-polychain-of-snd Cons.hyps*)  
**from** - *this*  
**have**  $linorder-list0.sortedP\ (ccw'\ (snd\ (last\ (polychain-of\ (x + z)\ zs))))$   
 $((x + z) \# map\ snd\ (butlast\ (polychain-of\ (x + z)\ zs)))$   
**}**

```

proof (rule ccw'.sortedP.Cons, safe)
  fix c d
  assume cd: (c, d) ∈ set (map snd (butlast (polychain-of (x + z) zs)))
  then obtain a b where ab: ((a, b), c, d) ∈ set (butlast (polychain-of (x +
z) zs))
    by auto
  have cd': (c, d) ∈ set (butlast (map snd (polychain-of (x + z) zs))) using cd
    by (auto simp: map-butlast)
  have ccw' (x + z) (c, d) (last (map snd (polychain-of (x + z) zs)))
  proof (rule ccw'.sortedP-right-of-last)
    show ccw'.sortedP (x + z) (map snd (polychain-of (x + z) zs))
      using Cons
    by (force intro!: ccw'.sortedP.Cons ccw'.sortedP.Nil ccw'-sortedP-polychain-of-snd
      elim!: ccw'.sortedP-Cons)
    show (c, d) ∈ set (map snd (polychain-of (x + z) zs))
      using in-set-butlastD[OF ab]
      by force
    from Cons(3) cd'
    show (c, d) ≠ last (map snd (polychain-of (x + z) zs))
      by (intro distinct-butlast-ne-last ccw'-sortedP-implies-distinct[where x=x])
        (auto elim!: ccw'.sortedP-Cons)
  qed
  thus ccw' (snd (last (polychain-of (x + z) zs))) (x + z) (c, d)
    by (auto simp: last-map[symmetric, where f= snd] ‹zs ≠ []› intro:
ccw'.cyclicI)
  qed
}
thus ?case
  by (auto simp: ccw'.sortedP.Nil)
qed (simp add: ccw'.sortedP.Nil)

lemma ccw'-polychain-of-sorted-center-last:
  assumes set-butlast: (c, d) ∈ set (butlast (polychain-of x0 xs))
  assumes sorted: ccw'.sortedP 0 xs
  assumes ne: xs ≠ []
  shows ccw' x0 d (snd (last (polychain-of x0 xs)))
proof –
  from ccw'-sortedP-polychain-of-snd[OF sorted, of x0]
  have ccw'.sortedP x0 (map snd (polychain-of x0 xs)) .
  also
  from set-butlast obtain ys zs where butlast (polychain-of x0 xs) = ys@((c,
d)#zs)
    by (auto simp add: in-set-conv-decomp)
  hence polychain-of x0 xs = ys @ (c, d) # zs @ [last (polychain-of x0 xs)]
  by (metis append-Cons append-assoc append-butlast-last-id ne polychain-of-eq-empty-iff)
  finally show ?thesis by (auto elim!: ccw'.sortedP-Cons simp: ccw'.sortedP-append-iff)
qed

end

```

## 9 CCW for Arbitrary Points in the Plane

**theory** *Counterclockwise-2D-Arbitrary*  
**imports** *Counterclockwise-2D-Strict*  
**begin**

### 9.1 Interpretation of Knuth's axioms in the plane

**definition** *lex::point*  $\Rightarrow$  *point*  $\Rightarrow$  *bool* **where**

$$\text{lex } p \ q \ \longleftrightarrow \ (\text{fst } p < \text{fst } q \ \vee \ \text{fst } p = \text{fst } q \ \wedge \ \text{snd } p < \text{snd } q \ \vee \ p = q)$$

**definition** *psi::point*  $\Rightarrow$  *point*  $\Rightarrow$  *point*  $\Rightarrow$  *bool* **where**

$$\text{psi } p \ q \ r \ \longleftrightarrow \ (\text{lex } p \ q \ \wedge \ \text{lex } q \ r)$$

**definition** *ccw::point*  $\Rightarrow$  *point*  $\Rightarrow$  *point*  $\Rightarrow$  *bool* **where**

$$\text{ccw } p \ q \ r \ \longleftrightarrow \ \text{ccw}' \ p \ q \ r \ \vee \ (\text{det3 } p \ q \ r = 0 \ \wedge \ (\text{psi } p \ q \ r \ \vee \ \text{psi } q \ r \ p \ \vee \ \text{psi } r \ p \ q))$$

**interpretation** *ccw*: *linorder-list0* *ccw* *x* **for** *x* .

**lemma** *ccw'-imp-ccw*: *ccw'* *a* *b* *c*  $\Longrightarrow$  *ccw* *a* *b* *c*

**by** (*simp* *add*: *ccw-def*)

**lemma** *ccw-ncoll-imp-ccw*: *ccw* *a* *b* *c*  $\Longrightarrow$   $\neg$ *coll* *a* *b* *c*  $\Longrightarrow$  *ccw'* *a* *b* *c*

**by** (*simp* *add*: *ccw-def*)

**lemma** *ccw-translate*: *ccw* *p* (*p* + *q*) (*p* + *r*) = *ccw* 0 *q* *r*

**by** (*auto* *simp*: *ccw-def* *psi-def* *lex-def*)

**lemma** *ccw-translate-origin*: *NO-MATCH* 0 *p*  $\Longrightarrow$  *ccw* *p* *q* *r* = *ccw* 0 (*q* - *p*) (*r* - *p*)

**using** *ccw-translate*[*of* *p* *q* - *p* *r* - *p*]

**by** *simp*

**lemma** *psi-scale*:

*psi* (*r* \*<sub>R</sub> *a*) (*r* \*<sub>R</sub> *b*) 0 = (if *r* > 0 then *psi* *a* *b* 0 else if *r* < 0 then *psi* 0 *b* *a* else *True*)

*psi* (*r* \*<sub>R</sub> *a*) 0 (*r* \*<sub>R</sub> *b*) = (if *r* > 0 then *psi* *a* 0 *b* else if *r* < 0 then *psi* *b* 0 *a* else *True*)

*psi* 0 (*r* \*<sub>R</sub> *a*) (*r* \*<sub>R</sub> *b*) = (if *r* > 0 then *psi* 0 *a* *b* else if *r* < 0 then *psi* *b* *a* 0 else *True*)

**by** (*auto* *simp*: *psi-def* *lex-def* *det3-def'* *not-less* *algebra-split-simps*)

**lemma** *ccw-scale23*: *ccw* 0 *a* *b*  $\Longrightarrow$  *r* > 0  $\Longrightarrow$  *ccw* 0 (*r* \*<sub>R</sub> *a*) (*r* \*<sub>R</sub> *b*)

**by** (*auto* *simp*: *ccw-def* *psi-scale*)

**lemma** *psi-notI*: *distinct3* *p* *q* *r*  $\Longrightarrow$  *psi* *p* *q* *r*  $\Longrightarrow$   $\neg$  *psi* *q* *p* *r*

**by** (*auto* *simp*: *algebra-simps* *psi-def* *lex-def*)

**lemma** *not-lex-eq*:  $\neg$  *lex* *a* *b*  $\longleftrightarrow$  *lex* *b* *a*  $\wedge$  *b*  $\neq$  *a*

**by** (*auto* *simp*: *algebra-simps* *lex-def* *prod-eq-iff*)

**lemma** *lex-trans*:  $\text{lex } a \ b \implies \text{lex } b \ c \implies \text{lex } a \ c$   
**by** (*auto simp: lex-def*)

**lemma** *lex-sym-eqI*:  $\text{lex } a \ b \implies \text{lex } b \ a \implies a = b$   
**and** *lex-sym-eq-iff*:  $\text{lex } a \ b \implies \text{lex } b \ a \longleftrightarrow a = b$   
**by** (*auto simp: lex-def*)

**lemma** *lex-refl[simp]*:  $\text{lex } p \ p$   
**by** (*metis not-lex-eq*)

**lemma** *psi-disjuncts*:  
 $\text{distinct3 } p \ q \ r \implies \text{psi } p \ q \ r \vee \text{psi } p \ r \ q \vee \text{psi } q \ r \ p \vee \text{psi } q \ p \ r \vee \text{psi } r \ p \ q \vee \text{psi } r \ q \ p$   
**by** (*auto simp: psi-def not-lex-eq*)

**lemma** *nlex-ccw-left*:  $\text{lex } x \ 0 \implies \text{ccw } 0 \ (0, 1) \ x$   
**by** (*auto simp: ccw-def lex-def psi-def ccw'-def det3-def'*)

**interpretation** *ccw-system123* *ccw*  
**apply** *unfold-locales*  
**subgoal by** (*force simp: ccw-def ccw'-def det3-def' algebra-simps*)  
**subgoal by** (*force simp: ccw-def ccw'-def det3-def' psi-def algebra-simps lex-sym-eq-iff*)  
**subgoal by** (*drule psi-disjuncts*) (*force simp: ccw-def ccw'-def det3-def' algebra-simps*)  
**done**

**lemma** *lex-scaleR-nonneg*:  $\text{lex } a \ b \implies r \geq 0 \implies \text{lex } a \ (a + r *_{\mathbb{R}} (b - a))$   
**by** (*auto simp: lex-def*)

**lemma** *lex-scale1-zero*:  
 $\text{lex } (v *_{\mathbb{R}} u) \ 0 = (\text{if } v > 0 \text{ then } \text{lex } u \ 0 \text{ else if } v < 0 \text{ then } \text{lex } 0 \ u \text{ else } \text{True})$   
**and** *lex-scale2-zero*:  
 $\text{lex } 0 \ (v *_{\mathbb{R}} u) = (\text{if } v > 0 \text{ then } \text{lex } 0 \ u \text{ else if } v < 0 \text{ then } \text{lex } u \ 0 \text{ else } \text{True})$   
**by** (*auto simp: lex-def prod-eq-iff less-eq-prod-def algebra-split-simps*)

**lemma** *nlex-add*:  
**assumes**  $\text{lex } a \ 0 \ \text{lex } b \ 0$   
**shows**  $\text{lex } (a + b) \ 0$   
**using** *assms* **by** (*auto simp: lex-def*)

**lemma** *nlex-sum*:  
**assumes** *finite*  $X$   
**assumes**  $\bigwedge x. x \in X \implies \text{lex } (f \ x) \ 0$   
**shows**  $\text{lex } (\text{sum } f \ X) \ 0$   
**using** *assms*  
**by** *induction* (*auto intro!: nlex-add*)

**lemma** *abs-add-nlex*:

```

assumes coll 0 a b
assumes lex a 0
assumes lex b 0
shows abs (a + b) = abs a + abs b
proof (rule antisym[OF abs-triangle-ineq])
have fst (|a| + |b|) ≤ fst |a + b|
  using assms
  by (auto simp add: det3-def' abs-prod-def lex-def)
moreover
{
  assume H: fst a < 0 fst b < 0
  hence snd b ≤ 0 ↔ snd a ≤ 0
  using assms
  by (auto simp: lex-def det3-def' mult.commute)
  (metis mult-le-cancel-left-neg mult-zero-right)+
  hence |snd a| + |snd b| ≤ |snd a + snd b|
  using H by auto
} hence snd (|a| + |b|) ≤ snd |a + b|
  using assms
  by (auto simp add: det3-def' abs-prod-def lex-def)
ultimately
show |a| + |b| ≤ |a + b| unfolding less-eq-prod-def ..
qed

```

```

lemma lex-sum-list: (∧x. x ∈ set xs ⇒ lex x 0) ⇒ lex (sum-list xs) 0
  by (induct xs) (auto simp: nlex-add)

```

```

lemma
  abs-sum-list-coll:
  assumes coll: list-all (coll 0 x) xs
  assumes x ≠ 0
  assumes up: list-all (λx. lex x 0) xs
  shows abs (sum-list xs) = sum-list (map abs xs)
  using assms
proof (induct xs)
  case (Cons y ys)
  hence coll 0 x y coll 0 x (sum-list ys)
  by (auto simp: list-all-iff intro!: coll-sum-list)
  hence coll 0 y (sum-list ys) using ⟨x ≠ 0⟩
  by (rule coll-trans)
  hence |y + sum-list ys| = abs y + abs (sum-list ys) using Cons
  by (subst abs-add-nlex) (auto simp: list-all-iff lex-sum-list)
  thus ?case using Cons by simp
qed simp

```

```

lemma lex-diff1: lex (a - b) c = lex a (c + b)
and lex-diff2: lex c (a - b) = lex (c + b) a
  by (auto simp: lex-def)

```

**lemma** *sum-list-eq-0-iff-nonpos*:

**fixes** *xs::'a::ordered-ab-group-add list*

**shows**  $\text{list-all } (\lambda x. x \leq 0) \text{ } xs \implies \text{sum-list } xs = 0 \longleftrightarrow (\forall n \in \text{set } xs. n = 0)$

**by** (*auto simp: list-all-iff sum-list-sum-nth sum-nonpos-eq-0-iff*)

(*auto simp add: in-set-conv-nth*)

**lemma** *sum-list-nlex-eq-zeroI*:

**assumes** *nlex: list-all*  $(\lambda x. \text{lex } x \ 0) \text{ } xs$

**assumes**  $\text{sum-list } xs = 0$

**assumes**  $x \in \text{set } xs$

**shows**  $x = 0$

**proof** –

**from** *assms(2)* **have**  $z1: \text{sum-list } (\text{map } \text{fst } xs) = 0$  **and**  $z2: \text{sum-list } (\text{map } \text{snd } xs) = 0$

**by** (*auto simp: prod-eq-iff fst-sum-list snd-sum-list*)

**from** *nlex* **have**  $\text{list-all } (\lambda x. x \leq 0) (\text{map } \text{fst } xs)$

**by** (*auto simp: lex-def list-all-iff*)

**from** *sum-list-eq-0-iff-nonpos[OF this] z1 nlex*

**have**

$z1': \text{list-all } (\lambda x. x = 0) (\text{map } \text{fst } xs)$

**and**  $\text{list-all } (\lambda x. x \leq 0) (\text{map } \text{snd } xs)$

**by** (*auto simp: list-all-iff lex-def*)

**from** *sum-list-eq-0-iff-nonpos[OF this(2)] z2*

**have**  $\text{list-all } (\lambda x. x = 0) (\text{map } \text{snd } xs)$  **by** (*simp add: list-all-iff*)

**with**  $z1'$  **show**  $x = 0$  **by** (*auto simp: list-all-iff zero-prod-def assms prod-eq-iff*)

**qed**

**lemma** *sum-list-eq0I*:  $(\forall x \in \text{set } xs. x = 0) \implies \text{sum-list } xs = 0$

**by** (*induct xs*) *auto*

**lemma** *sum-list-nlex-eq-zero-iff*:

**assumes** *nlex: list-all*  $(\lambda x. \text{lex } x \ 0) \text{ } xs$

**shows**  $\text{sum-list } xs = 0 \longleftrightarrow \text{list-all } ((=) \ 0) \text{ } xs$

**using** *assms*

**by** (*auto intro: sum-list-nlex-eq-zeroI sum-list-eq0I simp: list-all-iff*)

**lemma**

**assumes**  $\text{lex } p \ q \ \text{lex } q \ r \ 0 \leq a \ 0 \leq b \ 0 \leq c \ a + b + c = 1$

**assumes** *comb-def*:  $\text{comb} = a *_R p + b *_R q + c *_R r$

**shows** *lex-convex3*:  $\text{lex } p \ \text{comb} \ \text{lex } \text{comb} \ r$

**proof** –

**from** *convex3-alt[OF assms(3–6), of p q r]*

**obtain**  $u \ v$  **where**

$uv: a *_R p + b *_R q + c *_R r = p + u *_R (q - p) + v *_R (r - p) \ 0 \leq u \ 0 \leq v \ u + v \leq 1$  .

**have**  $\text{lex } p \ r$

**using** *assms* **by** (*metis lex-trans*)

**hence**  $\text{lex } (v *_R (p - r)) \ 0$  **using** *uv*

**by** (*simp add: lex-scale1-zero lex-diff1*)



**also**  
**have**  $\text{lex } 0 (u *_{\mathbb{R}} (q - p))$  **using**  $\langle \text{lex } p \ q \rangle \ uv$   
**by** (*simp add: lex-scale2-zero lex-diff2*)  
**finally** (*lex-trans*)  
**show**  $\text{lex } p \ \text{comb}$   
**unfolding** *comb-def uv*  
**by** (*simp add: lex-def prod-eq-iff algebra-simps*)  
**from** *comb-def* **have**  $\text{comb-def}'$ :  $\text{comb} = c *_{\mathbb{R}} r + b *_{\mathbb{R}} q + a *_{\mathbb{R}} p$  **by** *simp*  
**from** *assms* **have**  $c + b + a = 1$  **by** *simp*  
**from** *convex3-alt[OF assms(5,4,3) this, of r q p]*  
**obtain**  $u \ v$  **where**  $uv$ :  $c *_{\mathbb{R}} r + b *_{\mathbb{R}} q + a *_{\mathbb{R}} p = r + u *_{\mathbb{R}} (q - r) + v *_{\mathbb{R}}$   
 $(p - r)$   
 $0 \leq u \ 0 \leq v \ u + v \leq 1$   
**by** *auto*  
**have**  $\text{lex } (u *_{\mathbb{R}} (q - r)) \ 0$   
**using**  $uv \ \langle \text{lex } q \ r \rangle$   
**by** (*simp add: lex-scale1-zero lex-diff1*)  
**also have**  $\text{lex } 0 (v *_{\mathbb{R}} (r - p))$   
**using**  $uv \ \langle \text{lex } p \ r \rangle$   
**by** (*simp add: lex-scale2-zero lex-diff2*)  
**finally** (*lex-trans*) **show**  $\text{lex } \text{comb} \ r$   
**unfolding** *comb-def' uv*  
**by** (*simp add: lex-def prod-eq-iff algebra-simps*)  
**qed**

**lemma** *lex-convex-self2*:  
**assumes**  $\text{lex } p \ q \ 0 \leq a \ a \leq 1$   
**defines**  $r \equiv a *_{\mathbb{R}} p + (1 - a) *_{\mathbb{R}} q$   
**shows**  $\text{lex } p \ r$  (**is** *?th1*)  
**and**  $\text{lex } r \ q$  (**is** *?th2*)  
**using** *lex-convex3[OF  $\langle \text{lex } p \ q \rangle$ , of q a 1 - a 0 r]*  
*assms*  
**by** (*simp-all add: r-def*)

**lemma** *lex-uminus0[simp]*:  $\text{lex } (-a) \ 0 = \text{lex } 0 \ a$   
**by** (*auto simp: lex-def*)

**lemma**  
*lex-fst-zero-imp*:  
 $\text{fst } x = 0 \implies \text{lex } x \ 0 \implies \text{lex } y \ 0 \implies \neg \text{coll } 0 \ x \ y \implies \text{ccw}' \ 0 \ y \ x$   
**by** (*auto simp: ccw'-def det3-def' lex-def algebra-split-simps*)

**lemma** *lex-ccw-left*:  $\text{lex } x \ y \implies r > 0 \implies \text{ccw } y (y + (0, r)) \ x$   
**by** (*auto simp: ccw-def ccw'-def det3-def' algebra-simps lex-def psi-def*)

**lemma** *lex-translate-origin*: *NO-MATCH*  $0 \ a \implies \text{lex } a \ b = \text{lex } 0 (b - a)$   
**by** (*auto simp: lex-def*)

## 9.2 Order prover setup

**definition**  $lexs\ p\ q \longleftrightarrow (lex\ p\ q \wedge p \neq q)$

**lemma**  $lexs\text{-irrefl}$ :  $\neg\ lexs\ p\ p$   
**and**  $lexs\text{-imp-lex}$ :  $lexs\ x\ y \implies lex\ x\ y$   
**and**  $not\text{-lexs}$ :  $(\neg\ lexs\ x\ y) = (lex\ y\ x)$   
**and**  $not\text{-lex}$ :  $(\neg\ lex\ x\ y) = (lexs\ y\ x)$   
**and**  $eq\text{-lex-refl}$ :  $x = y \implies lex\ x\ y$   
**by** (*auto simp: lexs-def lex-def prod-eq-iff*)

**lemma**  $lexs\text{-trans}$ :  $lexs\ x\ y \implies lexs\ y\ z \implies lexs\ x\ z$   
**and**  $lexs\text{-lex-trans}$ :  $lexs\ x\ y \implies lex\ y\ z \implies lexs\ x\ z$   
**and**  $lex\text{-lexs-trans}$ :  $lex\ x\ y \implies lexs\ y\ z \implies lexs\ x\ z$   
**and**  $lex\text{-neq-trans}$ :  $lex\ a\ b \implies a \neq b \implies lexs\ a\ b$   
**and**  $neq\text{-lex-trans}$ :  $a \neq b \implies lex\ a\ b \implies lexs\ a\ b$   
**and**  $lexs\text{-imp-neq}$ :  $lexs\ a\ b \implies a \neq b$   
**by** (*auto simp: lexs-def lex-def prod-eq-iff*)

**local-setup**  $\langle$

*HOL-Order-Tac.declare-linorder* {  
 $ops = \{eq = @\{term\ \langle(=) :: point \Rightarrow point \Rightarrow bool\rangle\}, le = @\{term\ \langlelex\rangle\}, lt = @\{term\ \langlelexs\rangle\},$   
 $thms = \{trans = @\{thm\ lex-trans\}, refl = @\{thm\ lex-refl\}, eqD1 = @\{thm\ eq-lex-refl\},$   
 $eqD2 = @\{thm\ eq-lex-refl[OF\ sym]\}, antisym = @\{thm\ lex-sym-eqI\},$   
 $contr = @\{thm\ notE\}\},$   
 $conv\text{-thms} = \{less\text{-le} = @\{thm\ eq-reflection[OF\ lexs-def]\},$   
 $nless\text{-le} = @\{thm\ eq-reflection[OF\ not-lexs]\},$   
 $nle\text{-le} = @\{thm\ eq-reflection[OF\ not-lex-eq]\}\}$   
 $\}$   
 $\rangle$

## 9.3 Contradictions

**lemma**

**assumes**  $d$ :  $distinct4\ s\ p\ q\ r$   
**shows**  $contra1$ :  $\neg(lex\ p\ q \wedge lex\ q\ r \wedge lex\ r\ s \wedge indelta\ s\ p\ q\ r)$  (**is**  $?th1$ )  
**and**  $contra2$ :  $\neg(lex\ s\ p \wedge lex\ p\ q \wedge lex\ q\ r \wedge indelta\ s\ p\ q\ r)$  (**is**  $?th2$ )  
**and**  $contra3$ :  $\neg(lex\ p\ r \wedge lex\ p\ s \wedge lex\ q\ r \wedge lex\ q\ s \wedge insquare\ p\ r\ q\ s)$  (**is**  $?th3$ )

**proof** –

{  
**assume**  $det3\ s\ p\ q = 0\ det3\ s\ q\ r = 0\ det3\ s\ r\ p = 0\ det3\ p\ q\ r = 0$   
**hence**  $?th1\ ?th2\ ?th3$  **using**  $d$   
**by** (*auto simp add: det3-def' ccw'-def ccw-def psi-def algebra-simps*)  
**} moreover** {  
**assume**  $*$ :  $\neg(det3\ s\ p\ q = 0 \wedge det3\ s\ q\ r = 0 \wedge det3\ s\ r\ p = 0 \wedge det3\ p\ q\ r = 0)$   
{  
**assume**  $d0$ :  $det3\ p\ q\ r = 0$

```

with d have ?th1 ∧ ?th2
  by (force simp add: det3-def' ccw'-def ccw-def psi-def algebra-simps)
} moreover {
  assume dp: det3 p q r ≠ 0
  have ?th1 ∧ ?th2
    unfolding de-Morgan-disj[symmetric]
  proof (rule notI, goal-cases)
    case prems: 1
    hence **: indelta s p q r by auto
    hence nonnegs: det3 p q r ≥ 0 0 ≤ det3 s q r 0 ≤ det3 p s r 0 ≤ det3 p q s
      by (auto simp: ccw-def ccw'-def det3-def' algebra-simps)
    hence det-pos: det3 p q r > 0 using dp by simp
    have det-eq: det3 s q r + det3 p s r + det3 p q s = det3 p q r
      by (auto simp: ccw-def det3-def' algebra-simps)
    hence det-div-eq:
      det3 s q r / det3 p q r + det3 p s r / det3 p q r + det3 p q s / det3 p q r
= 1
      using det-pos by (auto simp: field-simps)
    from lex-convex3[OF - - - - det-div-eq convex-comb-dets[OF det-pos, of s]]
    have lex p s lex s r
      using prems by (auto simp: nonnegs)
    with prems d show False by (simp add: lex-sym-eq-iff)
  qed
} moreover have ?th3
proof (safe, goal-cases)
  case prems: 1
  have nonnegs: det3 p r q ≥ 0 det3 r q s ≥ 0 det3 s p r ≥ 0 det3 q s p ≥ 0
    using prems
  by (auto simp add: ccw-def ccw'-def less-eq-real-def)
  have dets-eq: det3 p r q + det3 q s p = det3 r q s + det3 s p r
    by (auto simp: det3-def')
  hence **: det3 p r q = 0 ∧ det3 q s p = 0 ⇒ det3 r q s = 0 ∧ det3 s p r
= 0
    using prems
  by (auto simp: ccw-def ccw'-def)
moreover
{
  fix p r q s
  assume det-pos: det3 p r q > 0
  assume dets-eq: det3 p r q + det3 q s p = det3 r q s + det3 s p r
  assume nonnegs: det3 r q s ≥ 0 det3 s p r ≥ 0 det3 q s p ≥ 0
  assume g14: lex p r lex p s lex q r lex q s
  assume d: distinct4 s p q r

  let ?sum = (det3 p r q + det3 q s p) / det3 p r q
  have eqs: det3 s p r = det3 p r s det3 r q s = det3 s r q det3 q s p = - det3
p s q
    by (auto simp: det3-def' algebra-simps)
  from convex-comb-dets[OF det-pos, of s]

```

```

      have ((det3 p r q / det3 p r q) *R s + (det3 q s p / det3 p r q) *R r) /R
?sum =
      ((det3 r q s / det3 p r q) *R p + (det3 s p r / det3 p r q) *R q) /R ?sum
      unfolding eqs
      by (simp add: algebra-simps prod-eq-iff)
      hence srpq: (det3 p r q / det3 p r q / ?sum) *R s + (det3 q s p / det3 p r
q / ?sum) *R r =
      (det3 r q s / det3 p r q / ?sum) *R p + (det3 s p r / det3 p r q / ?sum)
*R q
      (is ?s *R s + ?r *R r = ?p *R p + ?q *R q)
      using det-pos
      by (simp add: algebra-simps inverse-eq-divide)
      have eqs: ?s + ?r = 1 ?p + ?q = 1
      and s: ?s ≥ 0 ?s ≤ 1
      and r: ?r ≥ 0 ?r ≤ 1
      and p: ?p ≥ 0 ?p ≤ 1
      and q: ?q ≥ 0 ?q ≤ 1
      unfolding add-divide-distrib[symmetric]
      using det-pos nonnegs dets-eq
      by (auto)
      from eqs have eqs': 1 - ?s = ?r 1 - ?r = ?s 1 - ?p = ?q 1 - ?q = ?p
      by auto
      have comm: ?r *R r + ?s *R s = ?s *R s + ?r *R r
      ?q *R q + ?p *R p = ?p *R p + ?q *R q
      by simp-all
      define K
      where K = (det3 r q s / det3 p r q / ?sum) *R p + (det3 s p r / det3 p
r q / ?sum) *R q
      note rewr = eqs' comm srpq K-def[symmetric]
      from lex-convex-self2[OF - s, of s r, unfolded rewr]
      lex-convex-self2[OF - r, of r s, unfolded rewr]
      lex-convex-self2[OF - p, of p q, unfolded rewr]
      lex-convex-self2[OF - q, of q p, unfolded rewr]
      have False using g14 d det-pos
      by (metis lex-trans not-lex-eq)
    } note wlog = this
    from dets-eq have 1: det3 q s p + det3 p r q = det3 s p r + det3 r q s
      by simp
    from d have d': distinct4 r q p s by auto
    note wlog[of q s p r, OF - 1 nonnegs(3,2,1) prems(4,3,2,1) d']
      wlog[of p r q s, OF - dets-eq nonnegs(2,3,4) prems(1-4) d]
    ultimately show False using nonnegs d *
      by (auto simp: less-eq-real-def det3-def' algebra-simps)
  qed
  ultimately have ?th1 ?th2 ?th3 by blast+
} ultimately show ?th1 ?th2 ?th3 by force+
qed

```

lemma ccw'-subst-psi-disj:

**assumes**  $\det3\ t\ r\ s = 0$   
**assumes**  $\psi\ t\ r\ s \vee \psi\ t\ s\ r \vee \psi\ s\ r\ t$   
**assumes**  $s \neq t$   
**assumes**  $ccw'\ t\ r\ p$   
**shows**  $ccw'\ t\ s\ p$   
**proof cases**  
**assume**  $r \neq s$   
**from** *assms* **have**  $r \neq t$  **by** (*auto simp: det3-def' ccw'-def algebra-simps*)  
**from** *assms* **have**  $\det3\ r\ s\ t = 0$   
**by** (*auto simp: algebra-simps det3-def'*)  
**from** *coll-ex-scaling[OF assms(3) this]*  
**obtain**  $x$  **where**  $s: r = s + x *_{\mathbb{R}} (t - s)$  **by** *auto*  
**from** *assms(4)[simplified s]*  
**have**  $0 < \det3\ 0\ (s + x *_{\mathbb{R}} (t - s) - t)\ (p - t)$   
**by** (*auto simp: algebra-simps det3-def' ccw'-def*)  
**also have**  $s + x *_{\mathbb{R}} (t - s) - t = (1 - x) *_{\mathbb{R}} (s - t)$   
**by** (*simp add: algebra-simps*)  
**finally have**  $ccw': ccw'\ 0\ ((1 - x) *_{\mathbb{R}} (s - t))\ (p - t)$   
**by** (*simp add: ccw'-def*)  
**hence**  $neg: x \neq 1$  **by** (*auto simp add: det3-def' ccw'-def*)  
**have**  $tr: fst\ s < fst\ r \implies fst\ t = fst\ s \implies snd\ t \leq snd\ r$   
**by** (*simp add: s*)  
**from**  $s$  **have**  $fst\ (r - s) = fst\ (x *_{\mathbb{R}} (t - s))\ snd\ (r - s) = snd\ (x *_{\mathbb{R}} (t - s))$   
**by** *auto*  
**hence**  $x = (if\ fst\ (t - s) = 0\ then\ snd\ (r - s) / snd\ (t - s)\ else\ fst\ (r - s) /$   
 $fst\ (t - s))$   
**using**  $\langle s \neq t \rangle$   
**by** (*auto simp add: field-simps prod-eq-iff*)  
**also have**  $\dots \leq 1$   
**using** *assms*  
**by** (*auto simp: lex-def psi-def tr*)  
**finally have**  $x < 1$  **using** *neg* **by** *simp*  
**thus** *?thesis* **using** *ccw'*  
**by** (*auto simp: ccw'.translate-origin*)  
**qed** (*insert assms, simp*)

**lemma** *lex-contr*:

**assumes**  $distinct4\ t\ s\ q\ r$   
**assumes**  $lex\ t\ s\ lex\ s\ r$   
**assumes**  $\det3\ t\ s\ r = 0$   
**assumes**  $ccw'\ t\ s\ q$   
**assumes**  $ccw'\ t\ q\ r$   
**shows** *False*  
**using** *ccw'-subst-psi-disj[of t s r q] assms*  
**by** (*cases r = t*) (*auto simp: det3-def' algebra-simps psi-def ccw'-def*)

**lemma** *contra4*:

**assumes**  $distinct4\ s\ r\ q\ p$   
**assumes**  $lex: lex\ q\ p\ lex\ p\ r\ lex\ r\ s$

```

assumes ccw:  $ccw\ r\ q\ s\ ccw\ r\ s\ p\ ccw\ r\ q\ p$ 
shows False
proof cases
  assume c:  $ccw\ s\ q\ p$ 
  from c have *:  $indelta\ s\ r\ q\ p$ 
    using assms by simp
  with contra1[OF assms(1)]
  have  $\neg (lex\ r\ q \wedge lex\ q\ p \wedge lex\ p\ s)$  by blast
  hence  $\neg lex\ q\ p$ 
    using  $\langle ccw\ s\ q\ p \rangle$  contra1 cyclic assms nondegenerate by blast
  thus False using assms by simp
next
  assume  $\neg ccw\ s\ q\ p$ 
  with ccw have  $ccw\ q\ s\ p \wedge ccw\ s\ p\ r \wedge ccw\ p\ r\ q \wedge ccw\ r\ q\ s$ 
    by (metis assms(1) ccw'.cyclic ccw-def not-ccw'-eq psi-disjuncts)
  moreover
  from lex have  $lex\ q\ r\ lex\ q\ s\ lex\ p\ r\ lex\ p\ s$  by order+
  ultimately show False using contra3[of r q p s]  $\langle distinct4\ s\ r\ q\ p \rangle$  by blast
qed

```

**lemma** *not-coll-ordered-lexI*:

```

assumes  $l \neq x0$ 
  and  $lex\ x1\ r$ 
  and  $lex\ x1\ l$ 
  and  $lex\ r\ x0$ 
  and  $lex\ l\ x0$ 
  and  $ccw'\ x0\ l\ x1$ 
  and  $ccw'\ x0\ x1\ r$ 
shows  $det3\ x0\ l\ r \neq 0$ 
proof
  assume  $coll\ x0\ l\ r$ 
  from  $\langle coll\ x0\ l\ r \rangle$  have 1:  $coll\ 0\ (l - x0)\ (r - x0)$ 
    by (simp add: det3-def' algebra-simps)
  from  $\langle lex\ r\ x0 \rangle$  have 2:  $lex\ (r - x0)\ 0$  by (auto simp add: lex-def)
  from  $\langle lex\ l\ x0 \rangle$  have 3:  $lex\ (l - x0)\ 0$  by (auto simp add: lex-def)
  from  $\langle ccw'\ x0\ l\ x1 \rangle$  have 4:  $ccw'\ 0\ (l - x0)\ (x1 - x0)$ 
    by (simp add: det3-def' ccw'-def algebra-simps)
  from  $\langle ccw'\ x0\ x1\ r \rangle$  have 5:  $ccw'\ 0\ (x1 - x0)\ (r - x0)$ 
    by (simp add: det3-def' ccw'-def algebra-simps)
  from  $\langle lex\ x1\ r \rangle$  have 6:  $lex\ 0\ (r - x0 - (x1 - x0))$  by (auto simp: lex-def)
  from  $\langle lex\ x1\ l \rangle$  have 7:  $lex\ 0\ (l - x0 - (x1 - x0))$  by (auto simp: lex-def)
  define r' where  $r' = r - x0$ 
  define l' where  $l' = l - x0$ 
  define x0' where  $x0' = x1 - x0$ 
  from 1 2 3 4 5 6 7
  have rs:  $coll\ 0\ l'\ r'\ lex\ r'\ 0$ 
     $lex\ l'\ 0$ 
     $ccw'\ 0\ l'\ x0'$ 
     $ccw'\ 0\ x0'\ r'$ 

```

```

lex 0 (r' - x0')
lex 0 (l' - x0')
unfolding r'-def[symmetric] l'-def[symmetric] x0'-def[symmetric]
by auto
from assms have l' ≠ 0
  by (auto simp: l'-def)
from coll-scale[OF ‹coll 0 l' -› this]
obtain y where y: r' = y *R l' by auto
{
  assume y > 0
  with rs have False
    by (auto simp: det3-def' algebra-simps y ccw'-def)
} moreover {
  assume y < 0
  with rs have False
    by (auto simp: lex-def not-less algebra-simps algebra-split-simps y ccw'-def)
} moreover {
  assume y = 0
  from this rs have False
    by (simp add: ccw'-def y)
} ultimately show False by arith
qed

```

```

interpretation ccw-system4 ccw
proof unfold-locales
  fix p q r t
  assume ccw: ccw t q r ccw p t r ccw p q t
  show ccw p q r
  proof (cases det3 t q r = 0 ∧ det3 p t r = 0 ∧ det3 p q t = 0)
    case True
    {
      assume psi t q r ∨ psi q r t ∨ psi r t q
        psi p t r ∨ psi t r p ∨ psi r p t
        psi p q t ∨ psi q t p ∨ psi t p q
      hence psi p q r ∨ psi q r p ∨ psi r p q
        using lex-sym-eq-iff psi-def by blast
    }
  with True ccw show ?thesis
    by (simp add: det3-def' algebra-simps ccw-def ccw'-def)
  next
  case False
  hence 0 ≤ det3 t q r 0 ≤ det3 p t r 0 ≤ det3 p q t
    using ccw by (auto simp: less-eq-real-def ccw-def ccw'-def)
  with False show ?thesis
    by (auto simp: ccw-def det3-def' algebra-simps ccw'-def intro!: disjI1)
  qed
qed

```

**lemma** lex-total: lex t q ∧ t ≠ q ∨ lex q t ∧ t ≠ q ∨ t = q

by *auto*

**lemma**  
*ccw-two-up-contr*:  
**assumes** *c*:  $ccw' t p q ccw' t q r$   
**assumes** *ccws*:  $ccw t s p ccw t s q ccw t s r ccw t p q ccw t q r ccw t r p$   
**assumes** *distinct*:  $distinct5 t s p q r$   
**shows** *False*

**proof** –  
**from** *ccws*  
**have** *nn*:  $det3 t s p \geq 0 \ det3 t s q \geq 0 \ det3 t s r \geq 0 \ det3 t r p \geq 0$   
**by** (*auto simp add: less-eq-real-def ccw-def ccw'-def*)  
**with** *c det-identity*[*of t p q s r*]  
**have** *tsr*:  $coll t s r$  **and** *tsp*:  $coll t s p$   
**by** (*auto simp: add-nonneg-eq-0-iff ccw'-def*)  
**moreover**  
**have** *trp*:  $coll t r p$   
**by** (*metis ccw'-subst-collinear distinct not-ccw'-eq tsr tsp*)  
**ultimately have** *tpr*:  $coll t p r$   
**by** (*auto simp: det3-def' algebra-simps*)  
**moreover**  
**have** *psi*:  $psi t p r \vee psi t r p \vee psi r p t$   
**unfolding** *psi-def*

**proof** –  
**have** *ntsr*:  $\neg ccw' t s r \rightarrow \neg ccw' t r s$   
**using** *tsr*  
**by** (*auto simp: not-ccw'-eq det3-def' algebra-simps*)  
**have** *f8*:  $\neg ccw' t r s$   
**using** *tsr not-ccw'-eq* **by** *blast*  
**have** *f9*:  $\neg ccw' t r p$   
**using** *tpr* **by** (*simp add: not-ccw'-eq*)  
**have** *f10*:  $(lex t r \wedge lex r p \vee lex r p \wedge lex p t \vee lex p t \wedge lex t r)$   
**using** *ccw-def ccws(6) psi-def f9* **by** *auto*

**have**  $\neg ccw' t r q$   
**using** *c(2) not-ccw'-eq* **by** *blast*

**moreover**  
**have**  $\neg coll t q s$   
**using** *ntsr ccw'-subst-collinear distinct c(2)* **by** *blast*  
**hence**  $ccw' t s q$   
**by** (*meson ccw-def ccws(2) not-ccw'-eq*)

**moreover**  
**from** *tsr tsp <coll t r p>* **have**  $coll t p s \ coll t p r \ coll t r s$   
**by** (*auto simp add: det3-def' algebra-simps*)

**ultimately**  
**show**  $lex t p \wedge lex p r \vee lex t r \wedge lex r p \vee lex r p \wedge lex p t$   
**by** (*metis ccw'-subst-psi-disj distinct ccw-def ccws(3) contra4 tsp ntsr(1) f10*)

*lex-total*  
*psi-def trp*)



**qed**  
**moreover**  
**from** *distinct* **have**  $r \neq t$  **by** *auto*  
**ultimately**  
**have**  $ccw' t r q$  **using**  $c(1)$   
**by** (*rule ccw'-subst-psi-disj*)  
**thus** *False*  
**using**  $c(2)$  **by** (*simp add: ccw'-contra*)  
**qed**

**lemma**

*ccw-transitive-contr:*

**fixes**  $t s p q r$

**assumes** *ccws*:  $ccw t s p \ ccw t s q \ ccw t s r \ ccw t p q \ ccw t q r \ ccw t r p$

**assumes** *distinct*:  $distinct5 t s p q r$

**shows** *False*

**proof** –

**from** *ccws distinct* **have**  $*$ :  $ccw p t r \ ccw p q t$  **by** (*metis cyclic*) $+$

**with** *distinct* **have**  $ccw r p q$  **using** *interior[OF - - ccws(5) \*, of UNIV]*

**by** (*auto intro: cyclic*)

**from** *ccws* **have** *nonnegs*:  $det3 t s p \geq 0 \ det3 t s q \geq 0 \ det3 t s r \geq 0 \ det3 t p q$   
 $\geq 0$

$det3 t q r \geq 0 \ det3 t r p \geq 0$

**by** (*auto simp add: less-eq-real-def ccw-def ccw'-def*)

{

**assume**  $ccw' t p q \ ccw' t q r \ ccw' t r p$

**hence** *False*

**using** *ccw-two-up-contra ccws distinct* **by** *blast*

} **moreover** {

**assume**  $c$ :  $coll t q r \ coll t r p$

**with** *distinct four-points-aligned(1)[OF c, of s]*

**have**  $coll t p q$

**by** *auto*

**hence**  $(psi t p q \vee psi p q t \vee psi q t p)$

$psi t q r \vee psi q r t \vee psi r t q$

$psi t r p \vee psi r p t \vee psi p t r$

**using**  $ccws(4,5,6) c$

**by** (*simp-all add: ccw-def ccw'-def*)

**hence** *False*

**using** *distinct*

**by** (*auto simp: psi-def ccw'-def*)

} **moreover** {

**assume**  $c$ :  $det3 t p q = 0 \ det3 t q r > 0 \ det3 t r p = 0$

**have**  $\bigwedge x. det3 t q r = 0 \vee t = x \vee r = q \vee q = x \vee r = p \vee p = x \vee r = x$

**by** (*meson c(1) c(3) distinct four-points-aligned(1)*)

**hence** *False*

**by** (*metis (full-types) c(2) distinct less-irrefl*)

} **moreover** {

**assume**  $c$ :  $\det3\ t\ p\ q = 0\ \det3\ t\ q\ r = 0\ \det3\ t\ r\ p > 0$   
**have**  $\bigwedge x. \det3\ t\ r\ p = 0 \vee t = x \vee r = x \vee q = x \vee p = x$   
**by** (*meson*  $c(1)\ c(2)$  *distinct four-points-aligned(1)*)  
**hence** *False*  
**by** (*metis* (*no-types*)  $c(3)$  *distinct less-numeral-extra(3)*)  
**}** **moreover** {  
**assume**  $c$ :  $ccw'\ t\ p\ q\ ccw'\ t\ q\ r$   
**from** *ccw-two-up-contr*[*OF this ccws distinct*]  
**have** *False* .  
**}** **moreover** {  
**assume**  $c$ :  $ccw'\ t\ p\ q\ ccw'\ t\ r\ p$   
**from** *ccw-two-up-contr*[*OF this(2,1), of s*] *ccws distinct*  
**have** *False* **by** *auto*  
**}** **moreover** {  
**assume**  $c$ :  $ccw'\ t\ q\ r\ ccw'\ t\ r\ p$   
**from** *ccw-two-up-contr*[*OF this, of s*] *ccws distinct*  
**have** *False* **by** *auto*  
**}** **ultimately show** *False*  
**using**  $\langle 0 \leq \det3\ t\ p\ q \rangle$   
 $\langle 0 \leq \det3\ t\ q\ r \rangle \langle 0 \leq \det3\ t\ r\ p \rangle$   
**by** (*auto simp: less-eq-real-def ccw'-def*)  
**qed**

**interpretation** *ccw*: *ccw-system ccw*  
**by** *unfold-locales (metis ccw-transitive-contr nondegenerate)*

**lemma** *ccw-scaleR1*:  
 $\det3\ 0\ xr\ P \neq 0 \implies 0 < e \implies ccw\ 0\ xr\ P \implies ccw\ 0\ (e*_R xr)\ P$   
**by** (*simp add: ccw-def*)

**lemma** *ccw-scaleR2*:  
 $\det3\ 0\ xr\ P \neq 0 \implies 0 < e \implies ccw\ 0\ xr\ P \implies ccw\ 0\ xr\ (e*_R P)$   
**by** (*simp add: ccw-def*)

**lemma** *ccw-translate3-aux*:  
**assumes**  $\neg coll\ 0\ a\ b$   
**assumes**  $x < 1$   
**assumes**  $ccw\ 0\ (a - x*_R a)\ (b - x*_R a)$   
**shows**  $ccw\ 0\ a\ b$   
**proof** –  
**from** *assms* **have**  $\neg coll\ 0\ (a - x*_R a)\ (b - x*_R a)$   
**by** *simp*  
**with** *assms* **have**  $ccw'\ 0\ ((1 - x) *_R a)\ (b - x *_R a)$   
**by** (*simp add: algebra-simps ccw-def*)  
**thus**  $ccw\ 0\ a\ b$   
**using**  $\langle x < 1 \rangle$   
**by** (*simp add: ccw-def*)  
**qed**

**lemma** *ccw-translate3-minus*:  $\det3\ 0\ a\ b \neq 0 \implies x < 1 \implies ccw\ 0\ a\ (b - x *_R a) \implies ccw\ 0\ a\ b$   
**using** *ccw-translate3-ax*[of a b x] *ccw-scaleR1*[of a b - x \*\_R a 1-x ]  
**by** (*auto simp add: algebra-simps*)

**lemma** *ccw-translate3*:  $\det3\ 0\ a\ b \neq 0 \implies x < 1 \implies ccw\ 0\ a\ b \implies ccw\ 0\ a\ (x *_R a + b)$   
**by** (*rule ccw-translate3-minus*) (*auto simp add: algebra-simps*)

**lemma** *ccw-switch23*:  $\det3\ 0\ P\ Q \neq 0 \implies (\neg ccw\ 0\ Q\ P \longleftrightarrow ccw\ 0\ P\ Q)$   
**by** (*auto simp: ccw-def algebra-simps not-ccw'-eq ccw'-not-coll*)

**lemma** *ccw0-upward*:  $fst\ a > 0 \implies snd\ a = 0 \implies snd\ b > snd\ a \implies ccw\ 0\ a\ b$   
**by** (*auto simp: ccw-def det3-def' algebra-simps ccw'-def*)

**lemma** *ccw-uminus3*[*simp*]:  $\det3\ a\ b\ c \neq 0 \implies ccw\ (-a)\ (-b)\ (-c) = ccw\ a\ b\ c$   
**by** (*auto simp: ccw-def ccw'-def algebra-simps det3-def'*)

**lemma** *coll-minus-eq*:  $coll\ (x - a)\ (x - b)\ (x - c) = coll\ a\ b\ c$   
**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *ccw-minus3*:  $\neg coll\ a\ b\ c \implies ccw\ (x - a)\ (x - b)\ (x - c) \longleftrightarrow ccw\ a\ b\ c$   
**by** (*simp add: ccw-def coll-minus-eq*)

**lemma** *ccw0-uminus*[*simp*]:  $\neg coll\ 0\ a\ b \implies ccw\ 0\ (-a)\ (-b) \longleftrightarrow ccw\ 0\ a\ b$   
**using** *ccw-uminus3*[of 0 a b]  
**by** *simp*

**lemma** *lex-convex2*:  
**assumes**  $lex\ p\ q\ lex\ p\ r\ 0 \leq u\ u \leq 1$   
**shows**  $lex\ p\ (u *_R q + (1 - u) *_R r)$   
**proof** *cases*  
**note**  $\langle lex\ p\ q \rangle$   
**also**  
**assume**  $lex\ q\ r$   
**hence**  $lex\ q\ (u *_R q + (1 - u) *_R r)$   
**using**  $\langle 0 \leq u \rangle\ \langle u \leq 1 \rangle$   
**by** (*rule lex-convex-self2*)  
**finally** (*lex-trans*) **show** *?thesis* .  
**next**  
**note**  $\langle lex\ p\ r \rangle$   
**also**  
**assume**  $\neg lex\ q\ r$   
**hence**  $lex\ r\ q$   
**by** *simp*  
**hence**  $lex\ r\ ((1 - u) *_R r + (1 - (1 - u)) *_R q)$   
**using**  $\langle 0 \leq u \rangle\ \langle u \leq 1 \rangle$   
**by** (*intro lex-convex-self2*) *simp-all*  
**finally** (*lex-trans*) **show** *?thesis* **by** (*simp add: ac-simps*)

qed

lemma *lex-convex2'*:

assumes  $lex\ q\ p\ lex\ r\ p\ 0 \leq u\ u \leq 1$

shows  $lex\ (u *_{R}\ q + (1 - u) *_{R}\ r)\ p$

proof –

have  $lex\ (-\ p)\ (u *_{R}\ (-\ q) + (1 - u) *_{R}\ (-\ r))$

using *assms*

by (*intro lex-convex2*) (*auto simp: lex-def*)

thus *?thesis*

by (*auto simp: lex-def algebra-simps*)

qed

lemma *psi-convex1*:

assumes  $psi\ c\ a\ b$

assumes  $psi\ d\ a\ b$

assumes  $0 \leq u\ 0 \leq v\ u + v = 1$

shows  $psi\ (u *_{R}\ c + v *_{R}\ d)\ a\ b$

proof –

from *assms* have  $v: v = (1 - u)$  by *simp*

show *?thesis*

using *assms*

by (*auto simp: psi-def v intro!: lex-convex2' lex-convex2*)

qed

lemma *psi-convex2*:

assumes  $psi\ a\ c\ b$

assumes  $psi\ a\ d\ b$

assumes  $0 \leq u\ 0 \leq v\ u + v = 1$

shows  $psi\ a\ (u *_{R}\ c + v *_{R}\ d)\ b$

proof –

from *assms* have  $v: v = (1 - u)$  by *simp*

show *?thesis*

using *assms*

by (*auto simp: psi-def v intro!: lex-convex2' lex-convex2*)

qed

lemma *psi-convex3*:

assumes  $psi\ a\ b\ c$

assumes  $psi\ a\ b\ d$

assumes  $0 \leq u\ 0 \leq v\ u + v = 1$

shows  $psi\ a\ b\ (u *_{R}\ c + v *_{R}\ d)$

proof –

from *assms* have  $v: v = (1 - u)$  by *simp*

show *?thesis*

using *assms*

by (*auto simp: psi-def v intro!: lex-convex2*)

qed

**lemma** *coll-convex*:  
**assumes**  $coll\ a\ b\ c\ coll\ a\ b\ d$   
**assumes**  $0 \leq u\ 0 \leq v\ u + v = 1$   
**shows**  $coll\ a\ b\ (u *_{R} c + v *_{R} d)$   
**proof** *cases*  
**assume**  $a \neq b$   
**with**  $assms(1, 2)$   
**obtain**  $x\ y$  **where**  $xy: c - a = x *_{R} (b - a)\ d - a = y *_{R} (b - a)$   
**by** (*auto simp: det3-translate-origin dest!: coll-scale*)  
**from**  $assms$  **have**  $(u + v) *_{R} a = a$  **by** *simp*  
**hence**  $u *_{R} c + v *_{R} d - a = u *_{R} (c - a) + v *_{R} (d - a)$   
**by** (*simp add: algebra-simps*)  
**also have**  $\dots = u *_{R} x *_{R} (b - a) + v *_{R} y *_{R} (b - a)$   
**by** (*simp add: xy*)  
**also have**  $\dots = (u * x + v * y) *_{R} (b - a)$  **by** (*simp add: algebra-simps*)  
**also have**  $coll\ 0\ (b - a)\ \dots$   
**by** (*simp add: coll-scaleR-right-eq*)  
**finally show** *?thesis*  
**by** (*auto simp: det3-translate-origin*)  
**qed** *simp*

**lemma** (*in ccw-vector-space*) *convex3*:  
**assumes**  $u \geq 0\ v \geq 0\ u + v = 1\ ccw\ a\ b\ d\ ccw\ a\ b\ c$   
**shows**  $ccw\ a\ b\ (u *_{R} c + v *_{R} d)$   
**proof**  $-$   
**have**  $v = 1 - u$  **using**  $assms$  **by** *simp*  
**hence**  $ccw\ 0\ (b - a)\ (u *_{R} (c - a) + v *_{R} (d - a))$   
**using**  $assms$   
**by** (*cases u = 0 v = 0 rule: bool.exhaust[case-product bool.exhaust]*)  
(*auto simp add: translate-origin intro!: add3*)  
**also**  
**have**  $(u + v) *_{R} a = a$  **by** (*simp add: assms*)  
**hence**  $u *_{R} (c - a) + v *_{R} (d - a) = u *_{R} c + v *_{R} d - a$   
**by** (*auto simp: algebra-simps*)  
**finally show** *?thesis* **by** (*simp add: translate-origin*)  
**qed**

**lemma** *ccw-self[simp]*:  $ccw\ a\ a\ b\ ccw\ b\ a\ a$   
**by** (*auto simp: ccw-def psi-def intro: cyclic*)

**lemma** *ccw-seft'[simp]*:  $ccw\ a\ b\ a$   
**by** (*rule cyclic*) *simp*

**lemma** *ccw-convex'*:  
**assumes**  $uv: u \geq 0\ v \geq 0\ u + v = 1$   
**assumes**  $ccw\ a\ b\ c$  **and**  $1: coll\ a\ b\ c$   
**assumes**  $ccw\ a\ b\ d$  **and**  $2: \neg coll\ a\ b\ d$   
**shows**  $ccw\ a\ b\ (u *_{R} c + v *_{R} d)$   
**proof**  $-$

```

from assms have  $u: 0 \leq u \ u \leq 1$  and  $v: v = 1 - u$ 
  by (auto simp: algebra-simps)
let  $?c = u *_R c + v *_R d$ 
from 1 have abd:  $ccw' a b d$ 
  using assms by (auto simp: ccw-def)
{
  assume 2:  $\neg coll a b c$ 
  from 2 have  $ccw' a b c$ 
    using assms by (auto simp: ccw-def)
  with abd have  $ccw' a b ?c$ 
    using assms by (auto intro!: ccw'.convex3)
  hence ?thesis
    by (simp add: ccw-def)
} moreover {
  assume 2:  $coll a b c$ 
  {
    assume  $a = b$ 
    hence ?thesis by simp
  } moreover {
    assume  $v = 0$ 
    hence ?thesis
      by (auto simp: v assms)
  } moreover {
    assume  $v \neq 0 \ a \neq b$ 
    have  $coll c a b$  using 2 by (auto simp: det3-def' algebra-simps)
    from coll-ex-scaling[OF  $\langle a \neq b \rangle$  this]
    obtain  $r$  where  $c: c = a + r *_R (b - a)$  by auto
    have  $*$ :  $u *_R (a + r *_R (b - a)) + v *_R d - a = (u * r) *_R (b - a) + (1 - u) *_R (d - a)$ 
      by (auto simp: algebra-simps v)
    have  $ccw' a b ?c$ 
      using  $\langle v \neq 0 \rangle$  uv abd
      by (simp add: ccw'.translate-origin c *)
    hence ?thesis by (simp add: ccw-def)
  } ultimately have ?thesis by blast
} ultimately show ?thesis by blast
qed

```

**lemma** *ccw-convex*:

```

assumes uv:  $u \geq 0 \ v \geq 0 \ u + v = 1$ 
assumes  $ccw a b c$ 
assumes  $ccw a b d$ 
assumes lex:  $coll a b c \implies coll a b d \implies lex b a$ 
shows  $ccw a b (u *_R c + v *_R d)$ 

```

**proof** –

```

from assms have  $u: 0 \leq u \ u \leq 1$  and  $v: v = 1 - u$ 
  by (auto simp: algebra-simps)
let  $?c = u *_R c + v *_R d$ 
{

```

```

assume coll: coll a b c  $\wedge$  coll a b d
hence coll a b ?c
  by (auto intro!: coll-convex assms)
moreover
from coll have psi a b c  $\vee$  psi b c a  $\vee$  psi c a b psi a b d  $\vee$  psi b d a  $\vee$  psi d a
using assms by (auto simp add: ccw-def ccw'-not-coll)
hence psi a b ?c  $\vee$  psi b ?c a  $\vee$  psi ?c a b
  using coll uv lex
  by (auto simp: psi-def ccw-def not-lex lexs-def v intro: lex-convex2 lex-convex2')
ultimately have ?thesis
  by (simp add: ccw-def)
} moreover {
  assume 1:  $\neg$  coll a b d and 2:  $\neg$  coll a b c
  from 1 have abd: ccw' a b d
    using assms by (auto simp: ccw-def)
  from 2 have ccw' a b c
    using assms by (auto simp: ccw-def)
  with abd have ccw' a b ?c
    using assms by (auto intro!: ccw'.convex3)
  hence ?thesis
    by (simp add: ccw-def)
} moreover {
  assume  $\neg$  coll a b d coll a b c
  have ?thesis
    by (rule ccw-convex') fact+
} moreover {
  assume 1: coll a b d and 2:  $\neg$  coll a b c
  have  $0 \leq 1 - u$  using assms by (auto)
  from ccw-convex'[OF this  $\langle 0 \leq u \rangle$  -  $\langle$ ccw a b d $\rangle$  1  $\langle$ ccw a b c $\rangle$  2]
  have ?thesis by (simp add: algebra-simps v)
} ultimately show ?thesis by blast
qed

```

**interpretation** *ccw*: *ccw-convex ccw S*  $\lambda a b.$  *lex b a* **for** *S*  
**by** *unfold-locales* (*rule ccw-convex*)

**lemma** *ccw-sorted-scaleR*: *ccw.sortedP 0 xs*  $\implies r > 0 \implies$  *ccw.sortedP 0* (*map*  $((*_R) r)$  *xs*)  
**by** (*induct xs*)  
 (*auto intro!: ccw.sortedP.Cons ccw-scale23 elim!: ccw.sortedP-Cons simp del: scaleR-Pair*)

**lemma** *ccw-sorted-implies-ccw'-sortedP*:  
**assumes** *nonaligned*:  $\bigwedge y z. y \in \text{set } Ps \implies z \in \text{set } Ps \implies y \neq z \implies \neg \text{coll } 0 y z$   
**assumes** *sorted*: *linorder-list0.sortedP* (*ccw 0*) *Ps*  
**assumes** *distinct Ps*  
**shows** *linorder-list0.sortedP* (*ccw' 0*) *Ps*

```

using assms
proof (induction Ps)
  case (Cons P Ps)
  {
    fix p assume p: p ∈ set Ps
    moreover
    from p Cons.prems have ccw 0 P p
      by (auto elim!: linorder-list0.sortedP-Cons intro: Cons)
    ultimately
    have ccw' 0 P p
      using ⟨distinct (P#Ps)⟩
      by (intro ccw-ncoll-imp-ccw Cons) auto
  }
  moreover
  have linorder-list0.sortedP (ccw' 0) Ps
    using Cons.prems
    by (intro Cons) (auto elim!: linorder-list0.sortedP-Cons intro: Cons)
  ultimately
  show ?case
    by (auto intro!: linorder-list0.Cons)
qed (auto intro: linorder-list0.Nil)

end

```

## 10 Intersection

```

theory Intersection
imports
  HOL-Library.Monad-Syntax
  Polygon
  Counterclockwise-2D-Arbitrary
  Affine-Form
begin

```

### 10.1 Polygons and *ccw*, *Counterclockwise-2D-Arbitrary.lex*, *psi*, *coll*

```

lemma polychain-of-ccw-conjunction:
  assumes sorted: ccw'.sortedP 0 Ps
  assumes z: z ∈ set (polychain-of Pc Ps)
  shows list-all (λ(xi, xj). ccw xi xj (fst z) ∧ ccw xi xj (snd z)) (polychain-of Pc Ps)
  using assms
proof (induction Ps arbitrary: Pc z rule: list.induct)
  case (Cons P Ps)
  {
    assume set Ps = {}
    hence ?case using Cons by simp
  }

```



```

} moreover {
  assume set Ps ≠ {}
  hence Ps ≠ [] by simp
  {
    fix a assume a ∈ set Ps
    hence ccw' 0 P a
      using Cons.prem
      by (auto elim!: linorder-list0.sortedP-Cons)
  } note ccw' = this
  have sorted': linorder-list0.sortedP (ccw' 0) Ps
    using Cons.prem
    by (auto elim!: linorder-list0.sortedP-Cons)
  from in-set-polychain-of-imp-sum-list[OF Cons(3)]
  obtain d
  where d: z = (Pc + sum-list (take d (P # Ps)), Pc + sum-list (take (Suc d)
(P # Ps))) .

  from Cons(3)
  have disj: z = (Pc, Pc + P) ∨ z ∈ set (polychain-of (Pc + P) Ps)
    by auto

  let ?th = λ(xi, xj). ccw xi xj Pc ∧ ccw xi xj (Pc + P)
  have la: list-all ?th (polychain-of (Pc + P) Ps)
  proof (rule list-allI)
    fix x assume x: x ∈ set (polychain-of (Pc + P) Ps)
    from in-set-polychain-of-imp-sum-list[OF this]
    obtain e where e: x = (Pc + P + sum-list (take e Ps), Pc + P + sum-list
(take (Suc e) Ps))
      by auto
    {
      assume e ≥ length Ps
      hence ?th x by (auto simp: e)
    } moreover {
      assume e < length Ps
      have 0: ∧e. e < length Ps ⇒ ccw' 0 P (Ps ! e)
        by (rule ccw') simp
      have 2: 0 < e ⇒ ccw' 0 (P + sum-list (take e Ps)) (Ps ! e)
        using ⟨e < length Ps⟩
        by (auto intro!: ccw'.add1 0 ccw'.sum2 sorted' ccw'.sorted-nth-less
simp: sum-list-sum-nth)
      have ccw Pc (Pc + P + sum-list (take e Ps)) (Pc + P + sum-list (take
(Suc e) Ps))
        by (cases e = 0)
        (auto simp add: ccw-translate-origin take-Suc-eq add.assoc[symmetric] 0
2
      intro!: ccw'-imp-ccw intro: cyclic)
      hence ccw (Pc + P + sum-list (take e Ps)) (Pc + P + sum-list (take (Suc
e) Ps)) Pc
        by (rule cyclic)
    }
  }

```

```

moreover
have  $0 < e \implies ccw\ 0\ (Ps\ !\ e)\ (\neg\ sum\ list\ (take\ e\ Ps))$ 
  using  $\langle e < length\ Ps \rangle$ 
  by (auto simp add: take-Suc-eq add.assoc[symmetric]
    sum-list-sum-nth
    intro!: ccw'-imp-ccw ccw'.sum2 sorted' ccw'.sorted-nth-less)
  hence  $ccw\ (Pc + P + sum\ list\ (take\ e\ Ps))\ (Pc + P + sum\ list\ (take\ (Suc\ e)\ Ps))\ (Pc + P)$ 
  by (cases e = 0) (simp-all add: ccw-translate-origin take-Suc-eq)
  ultimately have  $?th\ x$ 
  by (auto simp add: e)
} ultimately show  $?th\ x$  by arith
qed
from disj have  $?case$ 
proof
  assume  $z: z \in set\ (polychain\ of\ (Pc + P)\ Ps)$ 
  have  $ccw\ 0\ P\ (sum\ list\ (take\ d\ (P\ \# \ Ps)))$ 
  proof (cases d)
    case (Suc e) note  $e = this$ 
    show  $?thesis$ 
    proof (cases e)
      case (Suc f)
      have  $ccw\ 0\ P\ (P + sum\ list\ (take\ (Suc\ f)\ Ps))$ 
      using  $z$ 
      by (force simp add: sum-list-sum-nth intro!: ccw'.sum intro: ccw'
        ccw'-imp-ccw)
      thus  $?thesis$ 
      by (simp add: e Suc)
    qed (simp add: e)
  qed simp
  hence  $ccw\ Pc\ (Pc + P)\ (fst\ z)$ 
  by (simp add: d ccw-translate-origin)
  moreover
  from  $z$  have  $ccw\ 0\ P\ (P + sum\ list\ (take\ d\ Ps))$ 
  by (cases d, force)
  (force simp add: sum-list-sum-nth intro!: ccw'-imp-ccw ccw'.sum intro:
    ccw'+)
  hence  $ccw\ Pc\ (Pc + P)\ (snd\ z)$ 
  by (simp add: d ccw-translate-origin)
  moreover
  from  $z$  Cons.prems have  $list\ all\ (\lambda(xi, xj). ccw\ xi\ xj\ (fst\ z) \wedge ccw\ xi\ xj\ (snd\ z))$ 
  (polychain-of (Pc + P) Ps)
  by (intro Cons.IH) (auto elim!: linorder-list0.sortedP-Cons)
  ultimately show  $?thesis$  by simp
  qed (simp add: la)
} ultimately show  $?case$  by blast
qed simp

```

**lemma** *lex-polychain-of-center*:  
 $d \in \text{set } (\text{polychain-of } x0 \text{ } xs) \implies \text{list-all } (\lambda x. \text{lex } x \ 0) \text{ } xs \implies \text{lex } (\text{snd } d) \ x0$   
**proof** (*induction xs arbitrary: x0*)  
  **case** (*Cons x xs*) **thus** ?case  
  **by** (*auto simp add: lex-def lex-translate-origin dest!: Cons.IH*)  
**qed** (*auto simp: lex-translate-origin*)

**lemma** *lex-polychain-of-last*:  
 $(c, d) \in \text{set } (\text{polychain-of } x0 \text{ } xs) \implies \text{list-all } (\lambda x. \text{lex } x \ 0) \text{ } xs \implies$   
 $\text{lex } (\text{snd } (\text{last } (\text{polychain-of } x0 \text{ } xs))) \ d$   
**proof** (*induction xs arbitrary: x0 c d*)  
  **case** (*Cons x xs*)  
  **let** ?c1 =  $c = x0 \wedge d = x0 + x$   
  **let** ?c2 =  $(c, d) \in \text{set } (\text{polychain-of } (x0 + x) \text{ } xs)$   
  **from** *Cons(2)* **have** ?c1  $\vee$  ?c2 **by** *auto*  
  **thus** ?case  
  **proof**  
    **assume** ?c1  
    **with** *Cons.prem*s **show** ?thesis  
    **by** (*auto intro!: lex-polychain-of-center*)  
  **next**  
    **assume** ?c2  
    **from** *Cons.IH[OF this]* *Cons.prem*s  
    **show** ?thesis  
    **by** *auto*  
  **qed**  
**qed** (*auto simp: lex-translate-origin*)

**lemma** *distinct-fst-polychain-of*:  
  **assumes**  $\text{list-all } (\lambda x. x \neq 0) \text{ } xs$   
  **assumes**  $\text{list-all } (\lambda x. \text{lex } x \ 0) \text{ } xs$   
  **shows**  $\text{distinct } (\text{map } \text{fst } (\text{polychain-of } x0 \text{ } xs))$   
  **using** *assms*  
**proof** (*induction xs arbitrary: x0*)  
  **case** *Nil*  
  **thus** ?case **by** *simp*  
**next**  
  **case** (*Cons x xs*)  
  **hence**  $\bigwedge d. \text{list-all } (\lambda x. \text{lex } x \ 0) \text{ } (x \# \text{take } d \text{ } xs)$   
  **by** (*auto simp: list-all-iff dest!: in-set-takeD*)  
  **from** *sum-list-nlex-eq-zero-iff[OF this]* *Cons.prem*s  
  **show** ?case  
  **by** (*cases xs = []*) (*auto intro!: Cons.IH elim!: in-set-polychain-of-imp-sum-list*)  
**qed**

**lemma** *distinct-snd-polychain-of*:  
  **assumes**  $\text{list-all } (\lambda x. x \neq 0) \text{ } xs$   
  **assumes**  $\text{list-all } (\lambda x. \text{lex } x \ 0) \text{ } xs$   
  **shows**  $\text{distinct } (\text{map } \text{snd } (\text{polychain-of } x0 \text{ } xs))$

```

using assms
proof (induction xs arbitrary: x0)
  case Nil
  thus ?case by simp
next
  case (Cons x xs)
  have contra:
     $\bigwedge d. xs \neq [] \implies \text{list-all } (\lambda x. x \neq 0) xs \implies \text{list-all } ((=) 0) (\text{take } (Suc\ d) xs)$ 
 $\implies \text{False}$ 
    by (auto simp: neq-Nil-conv)
  from Cons have  $\bigwedge d. \text{list-all } (\lambda x. \text{lex } x\ 0) (\text{take } (Suc\ d) xs)$ 
    by (auto simp: list-all-iff dest!: in-set-takeD)
  from sum-list-nlex-eq-zero-iff[OF this] Cons.prem1 contra
  show ?case
    by (cases xs = []) (auto intro!: Cons.IH elim!: in-set-polychain-of-imp-sum-list
dest!: contra)
qed

```

## 10.2 Orient all entries

**lift-definition** *nlex-pdevs::point pdevs  $\Rightarrow$  point pdevs*  
**is**  $\lambda x\ n. \text{if } \text{lex } 0\ (x\ n) \text{ then } -\ x\ n \text{ else } x\ n$  **by** *simp*

**lemma** *pdevs-apply-nlex-pdevs[simp]*: *pdevs-apply (nlex-pdevs x) n =*  
*(if lex 0 (pdevs-apply x n) then - pdevs-apply x n else pdevs-apply x n)*  
**by** *transfer simp*

**lemma** *nlex-pdevs-zero-pdevs[simp]*: *nlex-pdevs zero-pdevs = zero-pdevs*  
**by** (*auto intro!: pdevs-eqI*)

**lemma** *pdevs-domain-nlex-pdevs[simp]*: *pdevs-domain (nlex-pdevs x) = pdevs-domain*  
*x*  
**by** (*auto simp: pdevs-domain-def*)

**lemma** *degree-nlex-pdevs[simp]*: *degree (nlex-pdevs x) = degree x*  
**by** (*rule degree-cong*) *auto*

**lemma**  
*pdevs-val-nlex-pdevs:*  
**assumes**  $e \in UNIV \rightarrow I\ \text{uminus } 'I = I$   
**obtains**  $e'$  **where**  $e' \in UNIV \rightarrow I\ \text{pdevs-val } e\ x = \text{pdevs-val } e' (nlex-pdevs\ x)$   
**using** *assms*  
**by** (*atomize-elim, intro exI[where x= $\lambda n. \text{if } \text{lex } 0\ (\text{pdevs-apply } x\ n) \text{ then } -\ e\ n$*   
*else e n]*)  
*(force simp: pdevs-val-pdevs-domain intro!: sum.cong)*

**lemma**  
*pdevs-val-nlex-pdevs2:*  
**assumes**  $e \in UNIV \rightarrow I\ \text{uminus } 'I = I$

**obtains**  $e'$  **where**  $e' \in UNIV \rightarrow I$   $pdevs\text{-}val\ e\ (nlex\text{-}pdevs\ x) = pdevs\text{-}val\ e'\ x$   
**using**  $assms$   
**by** ( $atomize\text{-}elim$ ,  $intro\ exI$ [**where**  $x = \lambda n.$  ( $if\ lex\ 0\ (pdevs\text{-}apply\ x\ n)$  then  $- e\ n$   
else  $e\ n$ )])  
(force  $simp$ :  $pdevs\text{-}val\text{-}pdevs\text{-}domain\ intro!$ :  $sum.cong$ )

**lemma**

$pdevs\text{-}val\text{-}selsort\text{-}ccw$ :  
**assumes**  $distinct\ xs$   
**assumes**  $e \in UNIV \rightarrow I$   
**obtains**  $e'$  **where**  $e' \in UNIV \rightarrow I$   
 $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) = pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ (ccw.selsort\ 0\ xs))$   
**proof** –  
**have**  $set\ xs = set\ (ccw.selsort\ 0\ xs)\ distinct\ xs\ distinct\ (ccw.selsort\ 0\ xs)$   
**by** ( $simp\text{-}all\ add$ :  $assms$ )  
**from**  $this\ assms(2)$  **obtain**  $e'$   
**where**  $e' \in UNIV \rightarrow I$   
 $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) = pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ (ccw.selsort\ 0\ xs))$   
**by** ( $rule\ pdevs\text{-}val\text{-}permute$ )  
**thus**  $thesis\ ..$   
**qed**

**lemma**

$pdevs\text{-}val\text{-}selsort\text{-}ccw2$ :  
**assumes**  $distinct\ xs$   
**assumes**  $e \in UNIV \rightarrow I$   
**obtains**  $e'$  **where**  $e' \in UNIV \rightarrow I$   
 $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (ccw.selsort\ 0\ xs)) = pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ xs)$   
**proof** –  
**have**  $set\ (ccw.selsort\ 0\ xs) = set\ xs\ distinct\ (ccw.selsort\ 0\ xs)\ distinct\ xs$   
**by** ( $simp\text{-}all\ add$ :  $assms$ )  
**from**  $this\ assms(2)$  **obtain**  $e'$   
**where**  $e' \in UNIV \rightarrow I$   
 $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (ccw.selsort\ 0\ xs)) = pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ xs)$   
**by** ( $rule\ pdevs\text{-}val\text{-}permute$ )  
**thus**  $thesis\ ..$   
**qed**

**lemma**  $lex\text{-}nlex\text{-}pdevs$ :  $lex\ (pdevs\text{-}apply\ (nlex\text{-}pdevs\ x)\ i)\ 0$   
**by** ( $auto\ simp$ :  $lex\text{-}def\ algebra\text{-}simps\ prod\text{-}eq\text{-}iff$ )

### 10.3 Lowest Vertex

**definition**  $lowest\text{-}vertex$ :: $'a::ordered\text{-}euclidean\text{-}space\ aform \Rightarrow 'a$  **where**  
 $lowest\text{-}vertex\ X = fst\ X - sum\text{-}list\ (map\ snd\ (list\text{-}of\text{-}pdevs\ (snd\ X)))$

**lemma**  $snd\text{-}abs$ :  $snd\ (abs\ x) = abs\ (snd\ x)$   
**by** ( $metis\ abs\text{-}prod\text{-}def\ snd\text{-}conv$ )

**lemma** *lowest-vertex*:  
**fixes**  $X Y :: (\text{real} * \text{real}) \text{ aform}$   
**assumes**  $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
**assumes**  $\bigwedge i. \text{snd} (\text{pdevs-apply} (\text{snd } X) i) \geq 0$   
**assumes**  $\bigwedge i. \text{abs} (\text{snd} (\text{pdevs-apply} (\text{snd } Y) i)) = \text{abs} (\text{snd} (\text{pdevs-apply} (\text{snd } X) i))$   
**assumes**  $\text{degree-aform } Y = \text{degree-aform } X$   
**assumes**  $\text{fst } Y = \text{fst } X$   
**shows**  $\text{snd} (\text{lowest-vertex } X) \leq \text{snd} (\text{aform-val } e Y)$   
**proof** –  
**from**  $\text{abs-pdevs-val-le-tdev}[\text{OF } \text{assms}(1), \text{of } \text{snd } Y]$   
**have**  $\text{snd} |\text{pdevs-val } e (\text{snd } Y)| \leq (\sum i < \text{degree-aform } Y. |\text{snd} (\text{pdevs-apply} (\text{snd } X) i)|)$   
**unfolding** *lowest-vertex-def*  
**by** (*auto simp: aform-val-def tdev-def less-eq-prod-def snd-sum snd-abs assms*)  
**also have**  $\dots = (\sum i < \text{degree-aform } X. \text{snd} (\text{pdevs-apply} (\text{snd } X) i))$   
**by** (*simp add: assms*)  
**also have**  $\dots \leq \text{snd} (\text{sum-list} (\text{map } \text{snd} (\text{list-of-pdevs} (\text{snd } X))))$   
**by** (*simp add: sum-list-list-of-pdevs dense-list-of-pdevs-def sum-list-distinct-conv-sum-set snd-sum atLeast0LessThan*)  
**finally show** *?thesis*  
**by** (*auto simp: aform-val-def lowest-vertex-def minus-le-iff snd-abs abs-real-def assms split: if-split-asm*)  
**qed**

**lemma** *sum-list-nonposI*:  
**fixes**  $xs :: 'a :: \text{ordered-comm-monoid-add list}$   
**shows**  $\text{list-all } (\lambda x. x \leq 0) xs \implies \text{sum-list } xs \leq 0$   
**by** (*induct xs*) (*auto simp: intro!: add-nonpos-nonpos*)

**lemma** *center-le-lowest*:  
 $\text{fst} (\text{fst } X) \leq \text{fst} (\text{lowest-vertex} (\text{fst } X, \text{nlex-pdevs} (\text{snd } X)))$   
**by** (*auto simp: lowest-vertex-def fst-sum-list intro!: sum-list-nonposI*)  
(*auto simp: lex-def list-all-iff list-of-pdevs-def dest!: in-set-butlastD split: if-split-asm*)

**lemma** *lowest-vertex-eq-center-iff*:  
 $\text{lowest-vertex} (x0, \text{nlex-pdevs} (\text{snd } X)) = x0 \iff \text{snd } X = \text{zero-pdevs}$   
**proof**  
**assume**  $\text{lowest-vertex} (x0, \text{nlex-pdevs} (\text{snd } X)) = x0$   
**then have**  $\text{sum-list} (\text{map } \text{snd} (\text{list-of-pdevs} (\text{nlex-pdevs} (\text{snd } X)))) = 0$   
**by** (*simp add: lowest-vertex-def*)  
**moreover have**  $\text{list-all } (\lambda x. \text{Counterclockwise-2D-Arbitrary.lex } x 0) (\text{map } \text{snd} (\text{list-of-pdevs} (\text{nlex-pdevs} (\text{snd } X))))$   
**by** (*auto simp add: list-all-iff list-of-pdevs-def*)  
**ultimately have**  $\forall x \in \text{set} (\text{list-of-pdevs} (\text{nlex-pdevs} (\text{snd } X))). \text{snd } x = 0$   
**by** (*simp add: sum-list-nlex-eq-zero-iff list-all-iff*)  
**then have**  $\text{pdevs-apply} (\text{snd } X) i = \text{pdevs-apply } \text{zero-pdevs } i$  **for**  $i$   
**by** (*simp add: list-of-pdevs-def split: if-splits*)

**then show**  $\text{snd } X = \text{zero-pdevs}$   
**by** (*rule pdevs-eqI*)  
**qed** (*simp add: lowest-vertex-def*)

## 10.4 Collinear Generators

**lemma** *scaleR-le-self-cancel*:

**fixes**  $c::'a::\text{ordered-real-vector}$

**shows**  $a *_R c \leq c \iff (1 < a \wedge c \leq 0 \vee a < 1 \wedge 0 \leq c \vee a = 1)$

**using** *scaleR-le-0-iff[of a - 1 c]*

**by** (*simp add: algebra-simps*)

**lemma** *pdevs-val-coll*:

**assumes** *coll*:  $\text{list-all } (\text{coll } 0 \ x) \ xs$

**assumes** *nlex*:  $\text{list-all } (\lambda x. \text{lex } x \ 0) \ xs$

**assumes**  $x \neq 0$

**assumes**  $f \in \text{UNIV} \rightarrow \{-1 .. 1\}$

**obtains**  $e$  **where**  $e \in \{-1 .. 1\}$   $\text{pdevs-val } f \ (\text{pdevs-of-list } xs) = e *_R \ (\text{sum-list } xs)$

**proof** *cases*

**assume**  $\text{sum-list } xs = 0$

**have**  $\text{pdevs-of-list } xs = \text{zero-pdevs}$

**by** (*auto intro!: pdevs-eqI sum-list-nlex-eq-zeroI[OF nlex <sum-list xs = 0>]*  
*simp: pdevs-apply-pdevs-of-list list-all-iff dest!: nth-mem*)

**hence**  $0 \in \{-1 .. 1::\text{real}\}$   $\text{pdevs-val } f \ (\text{pdevs-of-list } xs) = 0 *_R \ \text{sum-list } xs$

**by** *simp-all*

**thus** *?thesis ..*

**next**

**assume**  $\text{sum-list } xs \neq 0$

**have**  $\text{sum-list } (\text{map } \text{abs } xs) \geq 0$

**by** (*auto intro!: sum-list-nonneg*)

**hence** [*simp*]:  $\neg \text{sum-list } (\text{map } \text{abs } xs) \leq 0$

**by** (*metis <sum-list xs ≠ 0> abs-le-zero-iff antisym-conv sum-list-abs*)

**have** *collist*:  $\text{list-all } (\text{coll } 0 \ (\text{sum-list } xs)) \ xs$

**proof** (*rule list-allI*)

**fix**  $y$  **assume**  $y \in \text{set } xs$

**hence**  $\text{coll } 0 \ x \ y$

**using** *coll* **by** (*auto simp: list-all-iff*)

**also have**  $\text{coll } 0 \ x \ (\text{sum-list } xs)$

**using** *coll* **by** (*auto simp: list-all-iff intro!: coll-sum-list*)

**finally** (*coll-trans*)

**show**  $\text{coll } 0 \ (\text{sum-list } xs) \ y$

**by** (*simp add: coll-commute <x ≠ 0>*)

**qed**

{

**fix**  $i$  **assume**  $i < \text{length } xs$

**hence**  $\exists r. xs ! i = r *_R \ (\text{sum-list } xs)$

by (*metis* (*mono-tags*) *coll-scale nth-mem*  $\langle \text{sum-list } xs \neq 0 \rangle$  *list-all-iff collist*)  
 } then obtain *r* where  $r: \bigwedge i. i < \text{length } xs \implies (xs ! i) = r i *_R (\text{sum-list } xs)$   
 by *metis*  
 let *?coll* = *pdevs-of-list xs*  
 have *pdevs-val f* (*pdevs-of-list xs*) =  
    $(\sum i < \text{degree } (\text{pdevs-of-list } xs). f i *_R xs ! i)$   
 unfolding *pdevs-val-sum*  
 by (*simp add: pdevs-apply-pdevs-of-list less-degree-pdevs-of-list-imp-less-length*)  
 also have ... =  $(\sum i < \text{degree } ?coll. (f i * r i) *_R (\text{sum-list } xs))$   
 by (*simp add: r less-degree-pdevs-of-list-imp-less-length*)  
 also have ... =  $(\sum i < \text{degree } ?coll. f i * r i) *_R (\text{sum-list } xs)$   
 by (*simp add: algebra-simps scaleR-sum-left*)  
 finally have *eq: pdevs-val f ?coll* =  $(\sum i < \text{degree } ?coll. f i * r i) *_R (\text{sum-list } xs)$   
 (*is - = ?e \*\_R -*)  
 .

have  $\text{abs } (\text{pdevs-val } f \text{ ?coll}) \leq \text{tdev } ?coll$   
 using *assms(4)*  
 by (*intro abs-pdevs-val-le-tdev*) (*auto simp: Pi-iff less-imp-le*)  
 also have ... = *sum-list* (*map abs xs*) using *assms* by *simp*  
 also note *eq*  
 finally have *less: |?e| \*\_R abs* (*sum-list xs*)  $\leq$  *sum-list* (*map abs xs*) by (*simp*  
*add: abs-scaleR*)  
 also have  $|\text{sum-list } xs| = \text{sum-list } (\text{map abs } xs)$   
 using *coll*  $\langle x \neq 0 \rangle$  *nlex*  
 by (*rule abs-sum-list-coll*)  
 finally have *?e*  $\in \{-1 .. 1\}$   
 by (*auto simp add: less-le scaleR-le-self-cancel*)  
 thus *?thesis* using *eq ..*  
 qed

**lemma** *scaleR-eq-self-cancel*:  
 fixes  $x::'a::\text{real-vector}$   
 shows  $a *_R x = x \longleftrightarrow a = 1 \vee x = 0$   
 using *scaleR-cancel-right*[*of a x 1*]  
 by *simp*

**lemma** *abs-pdevs-val-less-tdev*:  
 assumes  $e \in UNIV \rightarrow \{-1 < .. < 1\}$  *degree x*  $> 0$   
 shows  $|\text{pdevs-val } e \ x| < \text{tdev } x$   
**proof** –  
 have *bnds*:  $\bigwedge i. |e \ i| < 1 \wedge i. |e \ i| \leq 1$   
 using *assms*  
 by (*auto simp: Pi-iff abs-less-iff order.strict-implies-order*)  
**moreover**  
 let *?w* = *degree x - 1*  
 have *witness*:  $|e \ ?w| *_R |\text{pdevs-apply } x \ ?w| < |\text{pdevs-apply } x \ ?w|$   
 using *degree-least-nonzero*[*of x*] *assms bnds*  
 by (*intro neq-le-trans*) (*auto simp: scaleR-eq-self-cancel Pi-iff*)



*intro!*: *scaleR-left-le-one-le neq-le-trans*  
*intro!*: *abs-leI less-imp-le dest!:* *order.strict-implies-not-eq*)

**ultimately**  
**show** *?thesis*  
**using** *assms witness bnds*  
**by** (*auto simp: pdevs-val-sum tdev-def abs-scaleR*  
*intro!*: *le-less-trans[OF sum-abs] sum-strict-mono-ex1 scaleR-left-le-one-le*)

**qed**

**lemma** *pdevs-val-coll-strict:*  
**assumes** *coll:* *list-all (coll 0 x) xs*  
**assumes** *nlex:* *list-all (λx. lex x 0) xs*  
**assumes** *x ≠ 0*  
**assumes** *f ∈ UNIV → {-1 <..< 1}*  
**obtains** *e where e ∈ {-1 <..< 1} pdevs-val f (pdevs-of-list xs) = e \*<sub>R</sub> (sum-list xs)*

**proof** *cases*  
**assume** *sum-list xs = 0*  
**have** *pdevs-of-list xs = zero-pdevs*  
**by** (*auto intro!*: *pdevs-eqI sum-list-nlex-eq-zeroI[OF nlex ⟨sum-list xs = 0⟩*  
*simp: pdevs-apply-pdevs-of-list list-all-iff dest!:* *nth-mem*)  
**hence** *0 ∈ {-1 <..< 1::real} pdevs-val f (pdevs-of-list xs) = 0 \*<sub>R</sub> sum-list xs*  
**by** *simp-all*  
**thus** *?thesis ..*

**next**  
**assume** *sum-list xs ≠ 0*  
**have** *sum-list (map abs xs) ≥ 0*  
**by** (*auto intro!*: *sum-list-nonneg*)  
**hence** [*simp*]: *¬sum-list (map abs xs) ≤ 0*  
**by** (*metis ⟨sum-list xs ≠ 0⟩ abs-le-zero-iff antisym-conv sum-list-abs*)

**have** *∃ x ∈ set xs. x ≠ 0*  
**proof** (*rule ccontr*)  
**assume** *¬ (∃ x ∈ set xs. x ≠ 0)*  
**hence** *∧ x. x ∈ set xs ⇒ x = 0* **by** *auto*  
**hence** *sum-list xs = 0*  
**by** (*auto simp: sum-list-eq-0-iff-nonpos list-all-iff less-eq-prod-def prod-eq-iff*  
*fst-sum-list*  
*snd-sum-list*)  
**thus** *False* **using** *⟨sum-list xs ≠ 0⟩* **by** *simp*

**qed**

**then obtain** *i where i: i < length xs xs ! i ≠ 0*  
**by** (*metis in-set-conv-nth*)  
**hence** *i < degree (pdevs-of-list xs)*  
**by** (*auto intro!*: *degree-gt simp: pdevs-apply-pdevs-of-list*)  
**hence** *deg-pos: 0 < degree (pdevs-of-list xs)* **by** *simp*

**have** *collist: list-all (coll 0 (sum-list xs)) xs*  
**proof** (*rule list-allI*)

```

fix y assume y ∈ set xs
hence coll 0 x y
  using coll by (auto simp: list-all-iff)
also have coll 0 x (sum-list xs)
  using coll by (auto simp: list-all-iff intro!: coll-sum-list)
finally (coll-trans)
show coll 0 (sum-list xs) y
  by (simp add: coll-commute ⟨x ≠ 0⟩)
qed

{
  fix i assume i < length xs
  hence ∃ r. xs ! i = r *R (sum-list xs)
    by (metis (mono-tags, lifting) ⟨sum-list xs ≠ 0⟩ coll-scale collist list-all-iff
nth-mem)
} then obtain r where  $\bigwedge i. i < \text{length } xs \implies (xs ! i) = r i *_{R} (\text{sum-list } xs)$ 
  by metis
let ?coll = pdevs-of-list xs
have pdevs-val f (pdevs-of-list xs) =
  (∑ i < degree (pdevs-of-list xs). f i *R xs ! i)
  unfolding pdevs-val-sum
  by (simp add: less-degree-pdevs-of-list-imp-less-length pdevs-apply-pdevs-of-list)
also have ... = (∑ i < degree ?coll. (f i * r i) *R (sum-list xs))
  by (simp add: r less-degree-pdevs-of-list-imp-less-length)
also have ... = (∑ i < degree ?coll. f i * r i) *R (sum-list xs)
  by (simp add: algebra-simps scaleR-sum-left)
finally have eq: pdevs-val f ?coll = (∑ i < degree ?coll. f i * r i) *R (sum-list xs)
  (is - = ?e *R -)
.

have abs (pdevs-val f ?coll) < tdev ?coll
  using assms(4)
  by (intro abs-pdevs-val-less-tdev) (auto simp: Pi-iff less-imp-le deg-pos)
also have ... = sum-list (map abs xs) using assms by simp
also note eq
finally have less: |?e| *R abs (sum-list xs) < sum-list (map abs xs) by (simp
add: abs-scaleR)
also have |sum-list xs| = sum-list (map abs xs)
  using coll ⟨x ≠ 0⟩ nlex
  by (rule abs-sum-list-coll)
finally have ?e ∈ {-1 <..< 1}
  by (auto simp add: less-le scaleR-le-self-cancel)
thus ?thesis using eq ..
qed

```

## 10.5 Independent Generators

```

fun independent-pdevs::point list ⇒ point list
  where

```

```

independent-pdevs [] = []
| independent-pdevs (x#xs) =
  (let
    (cs, is) = List.partition (coll 0 x) xs;
    s = x + sum-list cs
  in (if s = 0 then [] else [s]) @ independent-pdevs is)

```

**lemma** *in-set-independent-pdevsE*:

**assumes**  $y \in \text{set } (\text{independent-pdevs } xs)$

**obtains**  $x$  **where**  $x \in \text{set } xs \text{ coll } 0 \ x \ y$

**proof** *atomize-elim*

**show**  $\exists x. x \in \text{set } xs \wedge \text{coll } 0 \ x \ y$

**using** *assms*

**proof** (*induct xs rule: independent-pdevs.induct*)

**case 1 thus** *?case by simp*

**next**

**case** ( $2 \ z \ zs$ )

**let**  $?c1 = y = z + \text{sum-list } (\text{filter } (\text{coll } 0 \ z) \ zs)$

**let**  $?c2 = y \in \text{set } (\text{independent-pdevs } (\text{filter } (\text{Not } \circ \text{coll } 0 \ z) \ zs))$

**from**  $2$

**have**  $?c1 \vee ?c2$

**by** (*auto simp: Let-def split: if-split-asm*)

**thus** *?case*

**proof**

**assume**  $?c2$

**hence**  $y \in \text{set } (\text{independent-pdevs } (\text{snd } (\text{partition } (\text{coll } 0 \ z) \ zs)))$

**by** *simp*

**from**  $2(1)[\text{OF refl prod.collapse refl this}]$

**show** *?case*

**by** *auto*

**next**

**assume**  $y: ?c1$

**show** *?case*

**unfolding**  $y$

**by** (*rule exI[where x=z]*) (*auto intro!: coll-add coll-sum-list*)

**qed**

**qed**

**qed**

**lemma** *in-set-independent-pdevs-nonzero*:  $x \in \text{set } (\text{independent-pdevs } xs) \implies x \neq 0$

**proof** (*induct xs rule: independent-pdevs.induct*)

**case** ( $2 \ y \ ys$ )

**from**  $2(1)[\text{OF refl prod.collapse refl}] \ 2(2)$

**show** *?case*

**by** (*auto simp: Let-def split: if-split-asm*)

**qed** *simp*

**lemma** *independent-pdevs-pairwise-non-coll*:

```

assumes  $x \in \text{set } (\text{independent-pdevs } xs)$ 
assumes  $y \in \text{set } (\text{independent-pdevs } xs)$ 
assumes  $\bigwedge x. x \in \text{set } xs \implies x \neq 0$ 
assumes  $x \neq y$ 
shows  $\neg \text{coll } 0 \ x \ y$ 
using assms
proof (induct xs rule: independent-pdevs.induct)
  case 1 thus ?case by simp
next
  case ( $2 \ z \ zs$ )
  from  $2$  have  $z \neq 0$  by simp
  from  $2(2)$  have  $x \neq 0$  by (rule in-set-independent-pdevs-nonzero)
  from  $2(3)$  have  $y \neq 0$  by (rule in-set-independent-pdevs-nonzero)
  let  $?c = \text{filter } (\text{coll } 0 \ z) \ zs$ 
  let  $?nc = \text{filter } (\text{Not } \circ \text{coll } 0 \ z) \ zs$ 
  {
    assume  $x \in \text{set } (\text{independent-pdevs } ?nc) \ y \in \text{set } (\text{independent-pdevs } ?nc)$ 
    hence  $\neg \text{coll } 0 \ x \ y$ 
    by (intro 2(1)[OF refl prod.collapse refl - - 2(4) 2(5)] auto)
  } note IH = this
  {
    fix  $x$  assume  $x \neq 0 \ z + \text{sum-list } ?c \neq 0$ 
     $\text{coll } 0 \ x \ (z + \text{sum-list } ?c)$ 
    hence  $x \notin \text{set } (\text{independent-pdevs } ?nc)$ 
    using sum-list-filter-coll-ex-scale[OF ‹z ≠ 0›, of z#zs]
    by (auto elim!: in-set-independent-pdevsE simp: coll-commute)
    (metis (no-types) ‹x ≠ 0› coll-scale coll-scaleR)
  } note nc = this
  from  $2(2,3,4,5)$   $nc[OF ‹x \neq 0›] \ nc[OF ‹y \neq 0›]$ 
  show ?case
  by (auto simp: Let-def IH coll-commute split: if-split-asm)
qed

```

```

lemma distinct-independent-pdevs[simp]:
  shows distinct (independent-pdevs xs)
proof (induct xs rule: independent-pdevs.induct)
  case 1 thus ?case by simp
next
  case ( $2 \ x \ xs$ )
  let  $?is = \text{independent-pdevs } (\text{filter } (\text{Not } \circ \text{coll } 0 \ x) \ xs)$ 
  have distinct ?is
    by (rule 2) (auto intro!: 2)
  thus ?case
proof (clarsimp simp add: Let-def)
  let  $?s = x + \text{sum-list } (\text{filter } (\text{coll } 0 \ x) \ xs)$ 
  assume  $s: ?s \neq 0 \ ?s \in \text{set } ?is$ 
  from in-set-independent-pdevsE[OF s(2)]
  obtain  $y$  where  $y:$ 
     $y \in \text{set } (\text{filter } (\text{Not } \circ \text{coll } 0 \ x) \ xs)$ 

```

```

    coll 0 y (x + sum-list (filter (coll 0 x) xs))
  by auto
  {
    assume y = 0 ∨ x = 0 ∨ sum-list (filter (coll 0 x) xs) = 0
    hence False using s y by (auto simp: coll-commute)
  } moreover {
    assume y ≠ 0 x ≠ 0 sum-list (filter (coll 0 x) xs) ≠ 0
      sum-list (filter (coll 0 x) xs) + x ≠ 0
    have *: coll 0 (sum-list (filter (coll 0 x) xs)) x
      by (auto intro!: coll-sum-list simp: coll-commute)
    have coll 0 y (sum-list (filter (coll 0 x) xs) + x)
      using s y by (simp add: add.commute)
    hence coll 0 y x using *
      by (rule coll-add-trans) fact+
    hence False using s y by (simp add: coll-commute)
  } ultimately show False using s y by (auto simp: add.commute)
qed
qed

```

**lemma** *in-set-independent-pdevs-invariant-nlex*:

$x \in \text{set } (\text{independent-pdevs } xs) \implies (\bigwedge x. x \in \text{set } xs \implies \text{lex } x \ 0) \implies$   
 $(\bigwedge x. x \in \text{set } xs \implies x \neq 0) \implies \text{Counterclockwise-2D-Arbitrary.lex } x \ 0$

**proof** (*induction xs arbitrary: x rule: independent-pdevs.induct*)

case 1 thus ?case by simp

next

case (2 y ys)

from 2 have y ≠ 0 by auto

from 2(2)

have  $x \in \text{set } (\text{independent-pdevs } (\text{filter } (\text{Not } \circ \text{coll } 0 \ y) \ ys)) \vee$

$x = y + \text{sum-list } (\text{filter } (\text{coll } 0 \ y) \ ys)$

by (auto simp: Let-def split: if-split-asm)

thus ?case

**proof**

assume  $x \in \text{set } (\text{independent-pdevs } (\text{filter } (\text{Not } \circ \text{coll } 0 \ y) \ ys))$

from 2(1)[*OF refl prod.collapse refl, simplified, OF this 2(3,4)*]

show ?case by simp

next

assume  $x = y + \text{sum-list } (\text{filter } (\text{coll } 0 \ y) \ ys)$

also have  $\text{lex } \dots \ 0$

by (force intro: nlex-add nlex-sum simp: sum-list-sum-nth

dest: nth-mem intro: 2(3))

finally show ?case .

qed

qed

**lemma**

*pdevs-val-independent-pdevs2*:

assumes  $e \in \text{UNIV} \rightarrow I$

shows  $\exists e'. e' \in \text{UNIV} \rightarrow I \wedge$

```

    pdevs-val e (pdevs-of-list (independent-pdevs xs)) = pdevs-val e' (pdevs-of-list
xs)
  using assms
proof (induct xs arbitrary: e rule: independent-pdevs.induct)
  case 1 thus ?case by force
next
  case (2 x xs)
  let ?coll = (filter (coll 0 x) (x#xs))
  let ?ncoll = (filter (Not o coll 0 x) (x#xs))
  let ?e0 = if sum-list ?coll = 0 then e else e o (+) (Suc 0)
  have pdevs-val e (pdevs-of-list (independent-pdevs (x#xs))) =
    e 0 *R (sum-list ?coll) + pdevs-val ?e0 (pdevs-of-list (independent-pdevs ?ncoll))
    (is - = ?vc + ?vnc)
    by (simp add: Let-def)
  also
  have e-split: (λ-. e 0) ∈ UNIV → I ?e0 ∈ UNIV → I
    using 2(2) by auto
  from 2(1)[OF refl prod.collapse refl e-split(2)]
  obtain e' where e': e' ∈ UNIV → I and ?vnc = pdevs-val e' (pdevs-of-list
?ncoll)
    by (auto simp add: o-def)
  note this(2)
  also
  have ?vc = pdevs-val (λ-. e 0) (pdevs-of-list ?coll)
    by (simp add: pdevs-val-const-pdevs-of-list)
  also
  from pdevs-val-pdevs-of-list-append[OF e-split(1) e'] obtain e'' where
    e'': e'' ∈ UNIV → I
    and pdevs-val (λ-. e 0) (pdevs-of-list ?coll) + pdevs-val e' (pdevs-of-list ?ncoll)
  =
    pdevs-val e'' (pdevs-of-list (?coll @ ?ncoll))
    by metis
  note this(2)
  also
  from pdevs-val-perm[OF partition-permI e'', of coll 0 x x#xs]
  obtain e''' where e''': e''' ∈ UNIV → I
    and ... = pdevs-val e''' (pdevs-of-list (x # xs))
    by metis
  note this(2)
  finally show ?case using e''' by auto
qed

```

**lemma** *list-all-filter*: *list-all p (filter p xs)*  
 by (*induct xs*) *auto*

**lemma** *pdevs-val-independent-pdevs*:  
 assumes *list-all (λx. lex x 0) xs*  
 assumes *list-all (λx. x ≠ 0) xs*  
 assumes  $e \in UNIV \rightarrow \{-1 .. 1\}$

```

shows  $\exists e'. e' \in UNIV \rightarrow \{-1 .. 1\} \wedge pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) =$ 
   $pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ xs))$ 
using assms(1,2,3)
proof (induct xs arbitrary: e rule: independent-pdevs.induct)
  case 1 thus ?case by force
next
  case (2 x xs)

  let ?coll = (filter (coll 0 x) (x#xs))
  let ?ncoll = (filter (Not o coll 0 x) xs)

  from 2 have  $x \neq 0$  by simp

  from pdevs-val-partition[OF 2(4), of x#xs coll 0 x]
  obtain f g where part: pdevs-val e (pdevs-of-list (x # xs)) =
     $pdevs\text{-}val\ f\ (pdevs\text{-}of\text{-}list\ ?coll) +$ 
     $pdevs\text{-}val\ g\ (pdevs\text{-}of\text{-}list\ (filter\ (Not\ o\ coll\ 0\ x)\ (x\#\ xs)))$ 
  and f: f ∈ UNIV → {-1 .. 1} and g: g ∈ UNIV → {-1 .. 1}
  by blast
  note part
  also

  have list-all (λx. lex x 0) (filter (coll 0 x) (x#xs))
  using 2(2) by (auto simp: inner-prod-def list-all-iff)
  from pdevs-val-coll[OF list-all-filter this ⟨x ≠ 0⟩ f]
  obtain f' where  $pdevs\text{-}val\ f\ (pdevs\text{-}of\text{-}list\ ?coll) = f' *_{R}\ sum\text{-}list\ (filter\ (coll\ 0$ 
x) (x#xs))
  and f': f' ∈ {-1 .. 1}
  by blast
  note this(1)
  also

  have  $filter\ (Not\ o\ coll\ 0\ x)\ (x\#\ xs) = ?ncoll$ 
  by simp
  also

  from 2(2) have list-all (λx. lex x 0) ?ncoll list-all (λx. x ≠ 0) ?ncoll
  by (auto simp: list-all-iff)
  from 2(1)[OF refl partition-filter-conv[symmetric] refl this g]
  obtain g'
  where  $pdevs\text{-}val\ g\ (pdevs\text{-}of\text{-}list\ ?ncoll) =$ 
     $pdevs\text{-}val\ g'\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ ?ncoll))$ 
  and g': g' ∈ UNIV → {-1 .. 1}
  by metis
  note this(1)
  also

  define h where  $h = (if\ sum\text{-}list\ ?coll \neq 0\ then\ rec\text{-}nat\ f'\ (\lambda i -. g'\ i)\ else\ g')$ 
  have  $f' *_{R}\ sum\text{-}list\ ?coll + pdevs\text{-}val\ g'\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ ?ncoll))$ 

```

=  $pdevs\text{-}val\ h\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ (x\#\!xs)))$   
 by (*simp add: h-def o-def Let-def*)

**finally**

**have**  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (x\ \# \ xs)) =$   
 $pdevs\text{-}val\ h\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ (x\ \# \ xs)))$  .

**moreover have**  $h \in UNIV \rightarrow \{-1 .. 1\}$

**proof**

**fix**  $i$  **show**  $h\ i \in \{-1 .. 1\}$   
**using**  $f'\ g'$   
**by** (*cases i*) (*auto simp: h-def*)

**qed**

**ultimately show** *?case by blast*

**qed**

**lemma**

*pdevs-val-independent-pdevs-strict:*

**assumes**  $list\text{-}all\ (\lambda x. lex\ x\ 0)\ xs$

**assumes**  $list\text{-}all\ (\lambda x. x \neq 0)\ xs$

**assumes**  $e \in UNIV \rightarrow \{-1 <..< < 1\}$

**shows**  $\exists e'. e' \in UNIV \rightarrow \{-1 <..< < 1\} \wedge pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ xs) =$   
 $pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ xs))$

**using** *assms(1,2,3)*

**proof** (*induct xs arbitrary: e rule: independent-pdevs.induct*)

**case 1 thus** *?case by force*

**next**

**case** ( $2\ x\ xs$ )

**let**  $?coll = (filter\ (coll\ 0\ x)\ (x\#\!xs))$

**let**  $?ncoll = (filter\ (Not\ o\ coll\ 0\ x)\ xs)$

**from**  $2$  **have**  $x \neq 0$  **by** *simp*

**from** *pdevs-val-partition[OF 2(4), of x#xs coll 0 x]*

**obtain**  $f\ g$

**where** *part:*  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (x\ \# \ xs)) =$

$pdevs\text{-}val\ f\ (pdevs\text{-}of\text{-}list\ ?coll) +$

$pdevs\text{-}val\ g\ (pdevs\text{-}of\text{-}list\ (filter\ (Not\ o\ coll\ 0\ x)\ (x\#\!xs)))$

**and**  $f: f \in UNIV \rightarrow \{-1 <..< < 1\}$  **and**  $g: g \in UNIV \rightarrow \{-1 <..< < 1\}$

**by** *blast*

**note** *part*

**also**

**have**  $list\text{-}all\ (\lambda x. lex\ x\ 0)\ (filter\ (coll\ 0\ x)\ (x\#\!xs))$

**using**  $2(2)$  **by** (*auto simp: inner-prod-def list-all-iff*)

**from** *pdevs-val-coll-strict[OF list-all-filter this <x ≠ 0> f]*

**obtain**  $f'$  **where**  $pdevs\text{-}val\ f\ (pdevs\text{-}of\text{-}list\ ?coll) = f' *_{\mathbb{R}} sum\text{-}list\ (filter\ (coll\ 0$



```

x) (x#xs)
  and f': f' ∈ {-1 <..< 1}
  by blast
note this(1)
also

have filter (Not o coll 0 x) (x#xs) = ?ncoll
  by simp
also

from 2(2) have list-all (λx. lex x 0) ?ncoll list-all (λx. x ≠ 0) ?ncoll
  by (auto simp: list-all-iff)
from 2(1)[OF refl partition-filter-conv[symmetric] refl this g]
obtain g'
where pdevs-val g (pdevs-of-list ?ncoll) =
  pdevs-val g' (pdevs-of-list (independent-pdevs ?ncoll))
  and g': g' ∈ UNIV → {-1 <..< 1}
  by metis
note this(1)
also

define h where h = (if sum-list ?coll ≠ 0 then rec-nat f' (λi -. g' i) else g')
have f' *R sum-list ?coll + pdevs-val g' (pdevs-of-list (independent-pdevs ?ncoll))
  = pdevs-val h (pdevs-of-list (independent-pdevs (x#xs)))
  by (simp add: h-def o-def Let-def)

finally
have pdevs-val e (pdevs-of-list (x # xs)) =
  pdevs-val h (pdevs-of-list (independent-pdevs (x # xs))) .

moreover have h ∈ UNIV → {-1 <..< 1}
proof
  fix i show h i ∈ {-1 <..< 1}
  using f' g'
  by (cases i) (auto simp: h-def)
qed

ultimately show ?case by blast
qed

lemma sum-list-independent-pdevs: sum-list (independent-pdevs xs) = sum-list xs
proof (induct xs rule: independent-pdevs.induct)
  case (2 y ys)
  from 2[OF refl prod.collapse refl]
  show ?case
  using sum-list-partition[of coll 0 y ys, symmetric]
  by (auto simp: Let-def)
qed simp

```

**lemma** *independent-pdevs-eq-Nil-iff*:  
 $list\text{-}all (\lambda x. lex\ x\ 0)\ xs \implies list\text{-}all (\lambda x. x \neq 0)\ xs \implies independent\text{-}pdevs\ xs = []$   
 $\longleftrightarrow xs = []$   
**proof** (*induct xs*)  
**case Nil thus ?case by simp**  
**next**  
**case (Cons x xs)**  
**from Cons(2) have**  $list\text{-}all (\lambda x. lex\ x\ 0)\ (x \# filter\ (coll\ 0\ x)\ xs)$   
**by (auto simp: list-all-iff)**  
**from sum-list-nlex-eq-zero-iff[OF this] Cons(3)**  
**show ?case**  
**by (auto simp: list-all-iff)**  
**qed**

## 10.6 Independent Oriented Generators

**definition**  $inl\ p = independent\text{-}pdevs\ (map\ snd\ (list\text{-}of\text{-}pdevs\ (nlex\text{-}pdevs\ p)))$

**lemma** *distinct-inl[simp]*:  $distinct\ (inl\ (snd\ X))$   
**by (auto simp: inl-def)**

**lemma** *in-set-inl-nonzero*:  $x \in set\ (inl\ xs) \implies x \neq 0$   
**by (auto simp: inl-def in-set-independent-pdevs-nonzero)**

**lemma**  
 $inl\text{-}ncoll: y \in set\ (inl\ (snd\ X)) \implies z \in set\ (inl\ (snd\ X)) \implies y \neq z \implies \neg coll\ 0\ y\ z$   
**unfolding inl-def**  
**by (rule independent-pdevs-pairwise-non-coll, assumption+)**  
**(auto simp: inl-def list-of-pdevs-nonzero)**

**lemma** *in-set-inl-lex*:  $x \in set\ (inl\ xs) \implies lex\ x\ 0$   
**by (auto simp: inl-def list-of-pdevs-def dest!: in-set-independent-pdevs-invariant-nlex split: if-split-asm)**

**interpretation** *ccw0*:  $linorder\text{-}list\ ccw\ 0\ set\ (inl\ (snd\ X))$

**proof** *unfold-locales*

**fix**  $a\ b\ c$

**show**  $a \neq b \implies ccw\ 0\ a\ b \vee ccw\ 0\ b\ a$

**by (metis UNIV-I ccw-self(1) nondegenerate)**

**assume**  $a: a \in set\ (inl\ (snd\ X))$

**show**  $ccw\ 0\ a\ a$

**by simp**

**assume**  $b: b \in set\ (inl\ (snd\ X))$

**show**  $ccw\ 0\ a\ b \implies ccw\ 0\ b\ a \implies a = b$

**by (metis ccw-self(1) in-set-inl-nonzero mem-Collect-eq not-ccw-eq a b)**

**assume**  $c: c \in set\ (inl\ (snd\ X))$

**assume** *distinct*:  $a \neq b\ b \neq c\ a \neq c$

**assume** *ab*:  $ccw\ 0\ a\ b$  **and** *bc*:  $ccw\ 0\ b\ c$

```

show  $ccw\ 0\ a\ c$ 
  using  $a\ b\ c\ ab\ bc$ 
proof ( $cases\ a = (0, 1)\ b = (0, 1)\ c = (0, 1)$ )
  rule: case-split[case-product case-split[case-product case-split]]
  assume  $nu: a \neq (0, 1)\ b \neq (0, 1)\ c \neq (0, 1)$ 
  have  $distinct5\ a\ b\ c\ (0, 1)\ 0\ in5\ UNIV\ a\ b\ c\ (0, 1)\ 0$ 
  using  $a\ b\ c\ distinct\ nu$  by (simp-all add: in-set-inl-nonzero)
  moreover have  $ccw\ 0\ (0, 1)\ a\ ccw\ 0\ (0, 1)\ b\ ccw\ 0\ (0, 1)\ c$ 
  by (auto intro!: nlex-ccw-left in-set-inl-lex a b c)
  ultimately show ?thesis using  $ab\ bc$ 
  by (rule ccw.transitive[where S=UNIV and s=(0, 1)])
next
  assume  $a \neq (0, 1)\ b = (0, 1)\ c \neq (0, 1)$ 
  thus ?thesis
  using  $ccw-switch23\ in-set-inl-lex\ inl-ncoll\ nlex-ccw-left\ a\ b\ ab$ 
  by blast
next
  assume  $a \neq (0, 1)\ b \neq (0, 1)\ c = (0, 1)$ 
  thus ?thesis
  using  $ccw-switch23\ in-set-inl-lex\ inl-ncoll\ nlex-ccw-left\ b\ c\ bc$ 
  by blast
qed (auto simp add: nlex-ccw-left in-set-inl-lex)
qed

lemma sorted-inl:  $ccw.sortedP\ 0\ (ccw.selsort\ 0\ (inl\ (snd\ X)))$ 
  by (rule ccw0.sortedP-selsort) auto

lemma sorted-scaled-inl:  $ccw.sortedP\ 0\ (map\ ((*_R)\ 2)\ (ccw.selsort\ 0\ (inl\ (snd\ X))))$ 
  using sorted-inl
  by (rule ccw-sorted-scaleR) simp

lemma distinct-selsort-inl:  $distinct\ (ccw.selsort\ 0\ (inl\ (snd\ X)))$ 
  by simp

lemma distinct-map-scaleRI:
  fixes  $xs::'a::real-vector\ list$ 
  shows  $distinct\ xs \implies c \neq 0 \implies distinct\ (map\ ((*_R)\ c)\ xs)$ 
  by (induct xs) auto

lemma distinct-scaled-inl:  $distinct\ (map\ ((*_R)\ 2)\ (ccw.selsort\ 0\ (inl\ (snd\ X))))$ 
  using distinct-selsort-inl
  by (rule distinct-map-scaleRI) simp

lemma ccw'-sortedP-scaled-inl:
   $ccw'.sortedP\ 0\ (map\ ((*_R)\ 2)\ (ccw.selsort\ 0\ (inl\ (snd\ X))))$ 
  using ccw-sorted-implies-ccw'-sortedP
  by (rule ccw'-sorted-scaleR) (auto simp: sorted-inl inl-ncoll)

```

**lemma** *pdevs-val-pdevs-of-list-inl2E*:  
**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$   
**obtains**  $e'$  **where**  $pdevs\text{-}val\ e\ X = pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ (inl\ X))$   $e' \in UNIV \rightarrow \{-1 .. 1\}$   
**proof** –  
**let**  $?l = map\ snd\ (list\text{-}of\text{-}pdevs\ (nlex\text{-}pdevs\ X))$   
**have**  $l: list\text{-}all\ (\lambda x. Counter-clockwise\text{-}2D\text{-}Arbitrary.lex\ x\ 0)\ ?l$   
 $list\text{-}all\ (\lambda x. x \neq 0)\ (map\ snd\ (list\text{-}of\text{-}pdevs\ (nlex\text{-}pdevs\ X)))$   
**by** (*auto simp: list-all-iff list-of-pdevs-def*)

**from** *pdevs-val-nlex-pdevs[OF assms(1)]*  
**obtain**  $e'$  **where**  $e' \in UNIV \rightarrow \{-1 .. 1\}$   $pdevs\text{-}val\ e\ X = pdevs\text{-}val\ e'\ (nlex\text{-}pdevs\ X)$   
**by** *auto*  
**note** *this(2)*  
**also from** *pdevs-val-of-list-of-pdevs2[OF ‹e' ∈ -›]*  
**obtain**  $e''$  **where**  $e'' \in UNIV \rightarrow \{-1 .. 1\}$   $\dots = pdevs\text{-}val\ e''\ (pdevs\text{-}of\text{-}list\ ?l)$   
**by** *metis*  
**note** *this(2)*  
**also from** *pdevs-val-independent-pdevs[OF l ‹e'' ∈ -›]*  
**obtain**  $e'''$   
**where**  $e''' \in UNIV \rightarrow \{-1 .. 1\}$   
**and**  $\dots = pdevs\text{-}val\ e'''\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ ?l))$   
**by** *metis*  
**note** *this(2)*  
**also have**  $\dots = pdevs\text{-}val\ e'''\ (pdevs\text{-}of\text{-}list\ (inl\ X))$   
**by** (*simp add: inl-def*)  
**finally have**  $pdevs\text{-}val\ e\ X = pdevs\text{-}val\ e'''\ (pdevs\text{-}of\text{-}list\ (inl\ X))$  .  
**thus thesis using**  $\langle e''' \in - \rangle ..$   
**qed**

**lemma** *pdevs-val-pdevs-of-list-inlE*:  
**assumes**  $e \in UNIV \rightarrow I\ uminus\ 'I = I\ 0 \in I$   
**obtains**  $e'$  **where**  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (inl\ X)) = pdevs\text{-}val\ e'\ X\ e' \in UNIV \rightarrow I$   
**proof** –  
**let**  $?l = map\ snd\ (list\text{-}of\text{-}pdevs\ (nlex\text{-}pdevs\ X))$   
**have**  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (inl\ X)) = pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ ?l))$   
**by** (*simp add: inl-def*)  
**also**  
**from** *pdevs-val-independent-pdevs2[OF ‹e ∈ -›]*  
**obtain**  $e'$   
**where**  $pdevs\text{-}val\ e\ (pdevs\text{-}of\text{-}list\ (independent\text{-}pdevs\ ?l)) = pdevs\text{-}val\ e'\ (pdevs\text{-}of\text{-}list\ ?l)$   
**and**  $e' \in UNIV \rightarrow I$   
**by** *auto*  
**note** *this(1)*  
**also**

**from** *pdevs-val-of-list-of-pdevs*[*OF*  $\langle e' \in \rightarrow \langle 0 \in I \rangle$ , of *nlex-pdevs* *X*]  
**obtain**  $e''$  **where** *pdevs-val*  $e'$  (*pdevs-of-list* ?*l*) = *pdevs-val*  $e''$  (*nlex-pdevs* *X*)  
**and**  $e'' \in UNIV \rightarrow I$   
**by** *metis*  
**note** *this*(1)  
**also**  
**from** *pdevs-val-nlex-pdevs2*[*OF*  $\langle e'' \in \rightarrow \langle - = I \rangle$ ]  
**obtain**  $e'''$  **where** *pdevs-val*  $e''$  (*nlex-pdevs* *X*) = *pdevs-val*  $e'''$  *X*  $e''' \in UNIV$   
 $\rightarrow I$   
**by** *metis*  
**note** *this*(1)  
**finally have** *pdevs-val*  $e$  (*pdevs-of-list* (*inl* *X*)) = *pdevs-val*  $e'''$  *X* .  
**thus** ?*thesis* **using**  $\langle e''' \in UNIV \rightarrow I \rangle$  ..  
**qed**

**lemma** *sum-list-nlex-eq-sum-list-inl*:  
*sum-list* (*map* *snd* (*list-of-pdevs* (*nlex-pdevs* *X*))) = *sum-list* (*inl* *X*)  
**by** (*auto simp*: *inl-def sum-list-list-of-pdevs sum-list-independent-pdevs*)

**lemma** *Affine-inl*: *Affine* (*fst* *X*, *pdevs-of-list* (*inl* (*snd* *X*))) = *Affine* *X*  
**by** (*auto simp*: *Affine-def valuate-def aform-val-def*  
*elim*: *pdevs-val-pdevs-of-list-inlE*[*of* - - *snd* *X*] *pdevs-val-pdevs-of-list-inl2E*[*of* -  
*snd* *X*])

## 10.7 Half Segments

**definition** *half-segments-of-aform::point aform*  $\Rightarrow$  (*point\*point*) *list*  
**where** *half-segments-of-aform* *X* =  
(*let*  
 $x0 = \text{lowest-vertex}$  (*fst* *X*, *nlex-pdevs* (*snd* *X*))  
*in*  
*polychain-of*  $x0$  (*map* (( $*$ <sub>R</sub>) 2) (*ccw.selsort* 0 (*inl* (*snd* *X*))))))

**lemma** *subsequent-half-segments*:  
**fixes** *X*  
**assumes** *Suc*  $i < \text{length}$  (*half-segments-of-aform* *X*)  
**shows** *snd* (*half-segments-of-aform* *X* !  $i$ ) = *fst* (*half-segments-of-aform* *X* ! *Suc*  $i$ )  
**using** *assms*  
**by** (*cases*  $i$ ) (*auto simp*: *half-segments-of-aform-def Let-def polychain-of-subsequent-eq*)

**lemma** *polychain-half-segments-of-aform*: *polychain* (*half-segments-of-aform* *X*)  
**by** (*auto simp*: *subsequent-half-segments intro!*: *polychainI*)

**lemma** *fst-half-segments*:  
*half-segments-of-aform* *X*  $\neq [] \implies$   
*fst* (*half-segments-of-aform* *X* ! 0) = *lowest-vertex* (*fst* *X*, *nlex-pdevs* (*snd* *X*))  
**by** (*auto simp*: *half-segments-of-aform-def Let-def o-def split-beta'*)

**lemma** *nlex-half-segments-of-aform*:  $(a, b) \in \text{set } (\text{half-segments-of-aform } X) \implies \text{lex } b \ a$

**by** (*auto simp: half-segments-of-aform-def prod-eq-iff lex-def dest!: in-set-polychain-ofD in-set-inl-lex*)

**lemma** *ccw-half-segments-of-aform-all*:

**assumes** *cd*:  $(c, d) \in \text{set } (\text{half-segments-of-aform } X)$

**shows** *list-all*  $(\lambda(xi, xj). \text{ccw } xi \ xj \ c \ \wedge \ \text{ccw } xi \ xj \ d) (\text{half-segments-of-aform } X)$

**proof** –

**have**

*list-all*  $(\lambda(xi, xj). \text{ccw } xi \ xj \ (\text{fst } (c, d)) \ \wedge \ \text{ccw } xi \ xj \ (\text{snd } (c, d)))$

*(polychain-of (lowest-vertex (fst X, nlex-pdevs (snd X)))*

*((map ((\*<sub>R</sub>) 2) (linorder-list0.selsort (ccw 0) (inl (snd X))))))*

**using** *ccw'-sortedP-scaled-inl cd[unfolded half-segments-of-aform-def Let-def]*

**by** (*rule polychain-of-ccw-conjunction*)

**thus** *?thesis*

**unfolding** *half-segments-of-aform-def[unfolded Let-def, symmetric] fst-conv snd-conv* .

**qed**

**lemma** *ccw-half-segments-of-aform*:

**assumes** *ij*:  $(xi, xj) \in \text{set } (\text{half-segments-of-aform } X)$

**assumes** *c*:  $(c, d) \in \text{set } (\text{half-segments-of-aform } X)$

**shows** *ccw*  $xi \ xj \ c \ \text{ccw } xi \ xj \ d$

**using** *ccw-half-segments-of-aform-all[OF c] ij*

**by** (*auto simp add: list-all-iff*)

**lemma** *half-segments-of-aform1*:

**assumes** *ch*:  $x \in \text{convex hull set } (\text{map fst } (\text{half-segments-of-aform } X))$

**assumes** *ab*:  $(a, b) \in \text{set } (\text{half-segments-of-aform } X)$

**shows** *ccw*  $a \ b \ x$

**using** *finite-set - ch*

**proof** (*rule ccw.convex-hull*)

**fix** *c* **assume** *c*  $\in \text{set } (\text{map fst } (\text{half-segments-of-aform } X))$

**then obtain** *d* **where**  $(c, d) \in \text{set } (\text{half-segments-of-aform } X)$  **by** *auto*

**with** *ab* **show** *ccw*  $a \ b \ c$

**by** (*rule ccw-half-segments-of-aform(1)*)

**qed** (*insert ab, simp add: nlex-half-segments-of-aform*)

**lemma** *half-segments-of-aform2*:

**assumes** *ch*:  $x \in \text{convex hull set } (\text{map snd } (\text{half-segments-of-aform } X))$

**assumes** *ab*:  $(a, b) \in \text{set } (\text{half-segments-of-aform } X)$

**shows** *ccw*  $a \ b \ x$

**using** *finite-set - ch*

**proof** (*rule ccw.convex-hull*)

**fix** *d* **assume** *d*  $\in \text{set } (\text{map snd } (\text{half-segments-of-aform } X))$

**then obtain** *c* **where**  $(c, d) \in \text{set } (\text{half-segments-of-aform } X)$  **by** *auto*

**with** *ab* **show** *ccw*  $a \ b \ d$

**by** (*rule ccw-half-segments-of-aform(2)*)

```

qed (insert ab, simp add: nlex-half-segments-of-aform)

lemma
  in-set-half-segments-of-aform-aform-valE:
  assumes (x2, y2) ∈ set (half-segments-of-aform X)
  obtains e where y2 = aform-val e X e ∈ UNIV → {-1 .. 1}
proof -
  from assms obtain d where
    y2 = lowest-vertex (fst X, nlex-pdevs (snd X)) +
      sum-list (take (Suc d) (map ((*R) 2) (ccw.selsort 0 (inl (snd X)))))
  by (auto simp: half-segments-of-aform-def elim!: in-set-polychain-of-imp-sum-list)
  also have lowest-vertex (fst X, nlex-pdevs (snd X)) =
    fst X - sum-list (map snd (list-of-pdevs (nlex-pdevs (snd X))))
  by (simp add: lowest-vertex-def)
  also have sum-list (map snd (list-of-pdevs (nlex-pdevs (snd X)))) =
    pdevs-val (λ-. 1) (nlex-pdevs (snd X))
  by (auto simp: pdevs-val-sum-list)
  also

  have sum-list (take (Suc d) (map ((*R) 2) (ccw.selsort 0 (inl (snd X))))) =
    pdevs-val (λi. if i ≤ d then 2 else 0) (pdevs-of-list (ccw.selsort 0 (inl (snd
X))))
  (is - = pdevs-val ?e -)
  by (subst sum-list-take-pdevs-val-eq)
  (auto simp: pdevs-val-sum if-distrib pdevs-apply-pdevs-of-list
    degree-pdevs-of-list-scaleR intro!: sum.cong )
  also
  obtain e'' where ... = pdevs-val e'' (pdevs-of-list (inl (snd X))) e'' ∈ UNIV →
{0..2}
  by (auto intro: pdevs-val-selsort-ccw2[of inl (snd X) ?e {0 .. 2}])
  note this(1)
  also note inl-def
  also
  let ?l = map snd (list-of-pdevs (nlex-pdevs (snd X)))
  from pdevs-val-independent-pdevs2[OF ‹e'' ∈ -›]
  obtain e'''
  where pdevs-val e'' (pdevs-of-list (independent-pdevs ?l)) = pdevs-val e''' (pdevs-of-list
?l)
  and e''' ∈ UNIV → {0..2}
  by auto
  note this(1)
  also
  have 0 ∈ {0 .. 2::real} by simp
  from pdevs-val-of-list-of-pdevs[OF ‹e''' ∈ -› this, of nlex-pdevs (snd X)]
  obtain e'''' where pdevs-val e''' (pdevs-of-list ?l) = pdevs-val e'''' (nlex-pdevs
(snd X))
  and e'''' ∈ UNIV → {0 .. 2}
  by metis
  note this(1)

```

**finally have**  
 $y2 = fst X + (pdevs-val e'''' (nlex-pdevs (snd X)) - pdevs-val (\lambda-. 1) (nlex-pdevs (snd X)))$   
**by** *simp*  
**also have**  $pdevs-val e'''' (nlex-pdevs (snd X)) - pdevs-val (\lambda-. 1) (nlex-pdevs (snd X)) =$   
 $pdevs-val (\lambda i. e'''' i - 1) (nlex-pdevs (snd X))$   
**by** (*simp add: pdevs-val-minus*)  
**also**  
**have**  $(\lambda i. e'''' i - 1) \in UNIV \rightarrow \{-1 .. 1\}$  **using**  $\langle e'''' \in - \rangle$  **by** *auto*  
**from** *pdevs-val-nlex-pdevs2*[*OF this*]  
**obtain**  $f$  **where**  $f \in UNIV \rightarrow \{-1 .. 1\}$   
**and**  $pdevs-val (\lambda i. e'''' i - 1) (nlex-pdevs (snd X)) = pdevs-val f (snd X)$   
**by** *auto*  
**note** *this*(2)  
**finally have**  $y2 = aform-val f X$  **by** (*simp add: aform-val-def*)  
**thus** *?thesis* **using**  $\langle f \in - \rangle$  ..  
**qed**

**lemma** *fst-hd-half-segments-of-aform*:  
**assumes** *half-segments-of-aform*  $X \neq []$   
**shows**  $fst (hd (half-segments-of-aform X)) = lowest-vertex (fst X, (nlex-pdevs (snd X)))$   
**using** *assms*  
**by** (*auto simp: half-segments-of-aform-def Let-def fst-hd-polychain-of*)

**lemma**  
 $linorder-list0.sortedP (ccw' (lowest-vertex (fst X, nlex-pdevs (snd X))))$   
 $(map snd (half-segments-of-aform X))$   
**(is**  $linorder-list0.sortedP (ccw' ?x0) ?ms$ **)**  
**unfolding** *half-segments-of-aform-def Let-def*  
**by** (*rule ccw'-sortedP-polychain-of-snd*) (*rule ccw'-sortedP-scaled-inl*)

**lemma** *rev-zip*:  $length\ xs = length\ ys \implies rev (zip\ xs\ ys) = zip (rev\ xs) (rev\ ys)$   
**by** (*induct xs ys rule: list-induct2*) *auto*

**lemma** *zip-upt-self-aux*:  $zip [0..<length\ xs] xs = map (\lambda i. (i, xs ! i)) [0..<length\ xs]$   
**by** (*auto intro!: nth-equalityI*)

**lemma** *half-segments-of-aform-strict*:  
**assumes**  $e \in UNIV \rightarrow \{-1 <..< 1\}$   
**assumes**  $seg \in set (half-segments-of-aform X)$   
**assumes**  $length (half-segments-of-aform X) \neq 1$   
**shows**  $ccw' (fst\ seg) (snd\ seg) (aform-val\ e\ X)$   
**using** *assms* **unfolding** *half-segments-of-aform-def Let-def*

**proof** –  
**have**  $len: length (map ((*_R) 2) (linorder-list0.selsort (ccw 0) (inl (snd X)))) \neq 1$



```

using assms by (auto simp: half-segments-of-aform-def)

have aform-val e X = fst X + pdevs-val e (snd X)
  by (simp add: aform-val-def)
also
obtain e' where e' ∈ UNIV → {-1 <..< 1}
  pdevs-val e (snd X) = pdevs-val e' (nlex-pdevs (snd X))
  using pdevs-val-nlex-pdevs[OF ‹e ∈ -›]
  by auto
note this(2)
also obtain e'' where e'' ∈ UNIV → {-1 <..< 1}
  ... = pdevs-val e'' (pdevs-of-list (map snd (list-of-pdevs (nlex-pdevs (snd X)))))
  by (metis pdevs-val-of-list-of-pdevs2[OF ‹e' ∈ -›])
note this(2)
also
obtain e''' where e''' ∈ UNIV → {-1 <..< 1} ... = pdevs-val e''' (pdevs-of-list
(inl (snd X)))
  unfolding inl-def
  using
    pdevs-val-independent-pdevs-strict[OF list-all-list-of-pdevsI,
      OF lex-nlex-pdevs list-of-pdevs-all-nonzero ‹e'' ∈ -›]
  by metis
note this(2)
also
from pdevs-val-selsort-ccw[OF distinct-inl ‹e''' ∈ -›]
obtain f where f ∈ UNIV → {-1 <..< 1}
  ... = pdevs-val f (pdevs-of-list (linorder-list0.selsort (ccw 0) (inl (snd X))))
  (is - = pdevs-val - (pdevs-of-list ?sl))
  by metis
note this(2)
also have ... = pdevs-val (λi. f i + 1) (pdevs-of-list ?sl) +
  lowest-vertex (fst X, nlex-pdevs (snd X)) - fst X
proof -
  have sum-list (dense-list-of-pdevs (nlex-pdevs (snd X))) =
  sum-list (dense-list-of-pdevs (pdevs-of-list (ccw.selsort 0 (inl (snd X)))))
  by (subst dense-list-of-pdevs-pdevs-of-list
    (auto simp: in-set-independent-pdevs-nonzero dense-list-of-pdevs-pdevs-of-list
inl-def
      sum-list-distinct-selsort sum-list-independent-pdevs sum-list-list-of-pdevs
    ))
  thus ?thesis
  by (auto simp add: pdevs-val-add lowest-vertex-def algebra-simps pdevs-val-sum-list
    sum-list-list-of-pdevs in-set-inl-nonzero dense-list-of-pdevs-pdevs-of-list)
qed
also have pdevs-val (λi. f i + 1) (pdevs-of-list ?sl) =
  pdevs-val (λi. 1/2 * (f i + 1)) (pdevs-of-list (map ((*R) 2) ?sl))
  (is - = pdevs-val ?f' (pdevs-of-list ?ssl))
  by (subst pdevs-val-cmul (simp add: pdevs-of-list-map-scaleR))
also
have distinct ?ssl ?f' ∈ UNIV → {0 <..< 1} using ‹f ∈ -›

```

**by** (*auto simp: distinct-map-scaleRI Pi-iff algebra-simps real-0-less-add-iff*)  
**from** *pdevs-of-list-sum[OF this]*  
**obtain** *g* **where**  $g \in UNIV \rightarrow \{0 < .. < 1\}$   
*pdevs-val ?f' (pdevs-of-list ?ssl) = ( $\sum P \in set ?ssl. g P *_R P$ )*  
**by** *blast*  
**note** *this(2)*  
**finally**  
**have** *aform-val e X = lowest-vertex (fst X, nlex-pdevs (snd X)) + ( $\sum P \in set ?ssl. g P *_R P$ )*  
**by** *simp*  
**also**  
**have** *ccw' (fst seg) (snd seg) ...*  
**using**  $\langle g \in \rightarrow - len \langle seg \in \rightarrow \rangle$  [*unfolded half-segments-of-aform-def Let-def*]  
**by** (*rule in-polychain-of-ccw*) (*simp add: ccw'-sortedP-scaled-inl*)  
**finally show** *?thesis .*  
**qed**

**lemma** *half-segments-of-aform-strict-all:*  
**assumes**  $e \in UNIV \rightarrow \{-1 < .. < 1\}$   
**assumes**  $length (half-segments-of-aform X) \neq 1$   
**shows** *list-all ( $\lambda seg. ccw' (fst seg) (snd seg) (aform-val e X)$ ) (half-segments-of-aform X)*  
**using** *assms*  
**by** (*auto intro!: half-segments-of-aform-strict simp: list-all-iff*)

**lemma** *list-allD: list-all P xs  $\implies x \in set xs \implies P x$*   
**by** (*auto simp: list-all-iff*)

**lemma** *minus-one-less-multI:*  
**fixes**  $e x :: real$   
**shows**  $-1 \leq e \implies 0 < x \implies x < 1 \implies -1 < e * x$   
**by** (*metis abs-add-one-gt-zero abs-real-def le-less-trans less-not-sym mult-less-0-iff mult-less-cancel-left1 real-0-less-add-iff*)

**lemma** *less-one-multI:*  
**fixes**  $e x :: real$   
**shows**  $e \leq 1 \implies 0 < x \implies x < 1 \implies e * x < 1$   
**by** (*metis (erased, opaque-lifting) less-eq-real-def monoid-mult-class.mult.left-neutral mult-strict-mono zero-less-one*)

**lemma** *ccw-half-segments-of-aform-strictI:*  
**assumes**  $e \in UNIV \rightarrow \{-1 < .. < 1\}$   
**assumes**  $(s1, s2) \in set (half-segments-of-aform X)$   
**assumes**  $length (half-segments-of-aform X) \neq 1$   
**assumes**  $x = (aform-val e X)$   
**shows** *ccw' s1 s2 x*  
**using** *half-segments-of-aform-strict[OF assms(1-3)] assms(4)* **by** *simp*

**lemma**

*ccw'-sortedP-subsequent:*  
**assumes**  $Suc\ i < length\ xs\ ccw'.sortedP\ 0\ (map\ dirvec\ xs)\ fst\ (xs\ !\ Suc\ i) = snd\ (xs\ !\ i)$   
**shows**  $ccw'\ (fst\ (xs\ !\ i))\ (snd\ (xs\ !\ i))\ (snd\ (xs\ !\ Suc\ i))$   
**using** *assms*  
**proof** (*induct xs arbitrary: i*)  
**case Nil thus ?case by simp**  
**next**  
**case (Cons x xs)**  
**thus ?case**  
**by** (*auto simp: nth-Cons dirvec-minus split: nat.split elim!: ccw'.sortedP-Cons*)  
*(metis (no-types, lifting) ccw'.renormalize length-greater-0-conv nth-mem prod.case-eq-if)*  
**qed**

**lemma** *ccw'-sortedP-uminus:*  $ccw'.sortedP\ 0\ xs \implies ccw'.sortedP\ 0\ (map\ uminus\ xs)$   
**by** (*induct xs*) (*auto intro!: ccw'.sortedP.Cons elim!: ccw'.sortedP-Cons simp del: uminus-Pair*)

**lemma** *subsequent-half-segments-ccw:*  
**fixes**  $X$   
**assumes**  $Suc\ i < length\ (half-segments-of-aform\ X)$   
**shows**  $ccw'\ (fst\ (half-segments-of-aform\ X\ !\ i))\ (snd\ (half-segments-of-aform\ X\ !\ i))$   
*(snd (half-segments-of-aform X ! Suc i))*  
**using** *assms*  
**by** (*intro ccw'-sortedP-subsequent*)  
*(auto simp: subsequent-half-segments half-segments-of-aform-def sorted-inl polychain-of-subsequent-eq intro!: ccw-sorted-implies-ccw'-sortedP[OF inl-ncoll] ccw'-sorted-scaleR)*

**lemma** *convex-polychain-half-segments-of-aform:*  $convex-polychain\ (half-segments-of-aform\ X)$   
**proof** *cases*  
**assume**  $length\ (half-segments-of-aform\ X) = 1$   
**thus ?thesis**  
**by** (*auto simp: length-Suc-conv convex-polychain-def polychain-def*)  
**next**  
**assume**  $len: length\ (half-segments-of-aform\ X) \neq 1$   
**show** *?thesis*  
**by** (*rule convex-polychainI*)  
*(simp-all add: polychain-half-segments-of-aform subsequent-half-segments-ccw ccw'-def[symmetric])*  
**qed**

**lemma** *hd-distinct-neq-last:*  $distinct\ xs \implies length\ xs > 1 \implies hd\ xs \neq last\ xs$   
**by** (*metis One-nat-def add-Suc-right distinct.simps(2) last.simps last-in-set less-irrefl*)

*list.exhaust list.sel(1) list.size(3) list.size(4) add.right-neutral nat-neq-iff not-less0)*

**lemma** *ccw-hd-last-half-segments-dirvec:*

**assumes** *length (half-segments-of-aform X) > 1*

**shows** *ccw' 0 (dirvec (hd (half-segments-of-aform X))) (dirvec (last (half-segments-of-aform X)))*

**proof** –

**let** *?i = ccw.selsort 0 (inl (snd X))*

**let** *?s = map ((\*<sub>R</sub>) 2) (ccw.selsort 0 (inl (snd X)))*

**from** *assms* **have** *l: 1 < length (inl (snd X)) inl (snd X) ≠ []*

**using** *assms* **by** *(auto simp add: half-segments-of-aform-def)*

**hence** *hd ?i ∈ set ?i last ?i ∈ set ?i*

**by** *(auto intro!: hd-in-set last-in-set simp del: ccw.set-selsort)*

**hence** *¬coll 0 (hd ?i) (last ?i)* **using** *l*

**by** *(intro inl-ncoll[of - X]) (auto simp: hd-distinct-neq-last)*

**hence** *¬coll 0 (hd ?s) (last ?s)* **using** *l*

**by** *(auto simp: hd-map last-map)*

**hence** *ccw' 0 (hd (map ((\*<sub>R</sub>) 2) (linorder-list0.selsort (ccw 0) (inl (snd X))))*

*(last (map ((\*<sub>R</sub>) 2) (linorder-list0.selsort (ccw 0) (inl (snd X))))*

**using** *assms*

**by** *(auto simp add: half-segments-of-aform-def*

*intro!: sorted-inl ccw-sorted-scaleR ccw.hd-last-sorted ccw-ncoll-imp-ccw)*

**with** *assms* **show** *?thesis*

**by** *(auto simp add: half-segments-of-aform-def Let-def*

*dirvec-hd-polychain-of dirvec-last-polychain-of length-greater-0-conv[symmetric]*

*simp del: polychain-of.simps length-greater-0-conv*

*split: if-split-asm)*

**qed**

**lemma** *map-fst-half-segments-aux-eq: [] ≠ half-segments-of-aform X ⇒*

*map fst (half-segments-of-aform X) =*

*fst (hd (half-segments-of-aform X))#butlast (map snd (half-segments-of-aform X))*

**by** *(rule nth-equalityI)*

*(auto simp: nth-Cons hd-conv-nth nth-butlast subsequent-half-segments split: nat.split)*

**lemma** *le-less-Suc-eq: x - Suc 0 ≤ (i::nat) ⇒ i < x ⇒ x - Suc 0 = i*

**by** *simp*

**lemma** *map-snd-half-segments-aux-eq: half-segments-of-aform X ≠ [] ⇒*

*map snd (half-segments-of-aform X) =*

*tl (map fst (half-segments-of-aform X)) @ [snd (last (half-segments-of-aform X))]*

**by** *(rule nth-equalityI)*

*(auto simp: nth-Cons hd-conv-nth nth-append nth-tl subsequent-half-segments*

*not-less last-conv-nth algebra-simps dest!: le-less-Suc-eq*

*split: nat.split)*

**lemma** *ccw'-sortedP-snd-half-segments-of-aform:*  
*ccw'.sortedP (lowest-vertex (fst X, nlex-pdevs (snd X))) (map snd (half-segments-of-aform X))*  
**by** (*auto simp: half-segments-of-aform-def Let-def*  
*intro!: ccw'.sortedP.Cons ccw'-sortedP-polychain-of-snd ccw'-sortedP-scaled-inl*)

**lemma**  
*lex-half-segments-lowest-vertex:*  
**assumes**  $(c, d) \in \text{set } (\text{half-segments-of-aform } X)$   
**shows**  $\text{lex } d \text{ (lowest-vertex (fst } X, \text{nlex-pdevs (snd } X))}$   
**unfolding** *half-segments-of-aform-def Let-def*  
**by** (*rule lex-polychain-of-center[OF assms[unfolded half-segments-of-aform-def*  
*Let-def],*  
*unfolded snd-conv]*)  
*(auto simp: list-all-iff lex-def dest!: in-set-inl-lex)*

**lemma**  
*lex-half-segments-lowest-vertex':*  
**assumes**  $d \in \text{set } (\text{map snd } (\text{half-segments-of-aform } X))$   
**shows**  $\text{lex } d \text{ (lowest-vertex (fst } X, \text{nlex-pdevs (snd } X))}$   
**using** *assms*  
**by** (*auto intro: lex-half-segments-lowest-vertex*)

**lemma**  
*lex-half-segments-last:*  
**assumes**  $(c, d) \in \text{set } (\text{half-segments-of-aform } X)$   
**shows**  $\text{lex } (\text{snd } (\text{last } (\text{half-segments-of-aform } X))) \text{ } d$   
**using** *assms*  
**unfolding** *half-segments-of-aform-def Let-def*  
**by** (*rule lex-polychain-of-last*) (*auto simp: list-all-iff lex-def dest!: in-set-inl-lex*)

**lemma**  
*lex-half-segments-last':*  
**assumes**  $d \in \text{set } (\text{map snd } (\text{half-segments-of-aform } X))$   
**shows**  $\text{lex } (\text{snd } (\text{last } (\text{half-segments-of-aform } X))) \text{ } d$   
**using** *assms*  
**by** (*auto intro: lex-half-segments-last*)

**lemma**  
*ccw'-half-segments-lowest-last:*  
**assumes** *set-butlast:  $(c, d) \in \text{set } (\text{butlast } (\text{half-segments-of-aform } X))$*   
**assumes** *ne:  $\text{inl } (\text{snd } X) \neq []$*   
**shows**  $\text{ccw}' \text{ (lowest-vertex (fst } X, \text{nlex-pdevs (snd } X))} \text{ } d \text{ (snd } (\text{last } (\text{half-segments-of-aform } X)))$   
**using** *set-butlast*  
**unfolding** *half-segments-of-aform-def Let-def*  
**by** (*rule ccw'-polychain-of-sorted-center-last*) (*auto simp: ne ccw'-sortedP-scaled-inl*)

**lemma** *distinct-fst-half-segments:*

*distinct* (map fst (half-segments-of-aform X))  
**by** (auto simp: half-segments-of-aform-def list-all-iff lex-scale1-zero  
 simp del: scaleR-Pair  
 intro!: distinct-fst-polychain-of  
 dest: in-set-inl-nonzero in-set-inl-lex)

**lemma** *distinct-snd-half-segments*:  
*distinct* (map snd (half-segments-of-aform X))  
**by** (auto simp: half-segments-of-aform-def list-all-iff lex-scale1-zero  
 simp del: scaleR-Pair  
 intro!: distinct-snd-polychain-of  
 dest: in-set-inl-nonzero in-set-inl-lex)

## 10.8 Mirror

**definition** *mirror-point*  $x y = 2 *_{\mathbb{R}} x - y$

**lemma** *ccw'-mirror-point3[simp]*:  
 $ccw' (\text{mirror-point } x y) (\text{mirror-point } x z) (\text{mirror-point } x w) \iff ccw' y z w$   
**by** (auto simp: mirror-point-def det3-def' ccw'-def algebra-simps)

**lemma** *mirror-point-self-inverse[simp]*:  
**fixes**  $x::'a::\text{real-vector}$   
**shows**  $\text{mirror-point } p (\text{mirror-point } p x) = x$   
**by** (auto simp: mirror-point-def scaleR-2)

**lemma** *mirror-half-segments-of-aform*:  
**assumes**  $e \in \text{UNIV} \rightarrow \{-1 <..< 1\}$   
**assumes**  $\text{length} (\text{half-segments-of-aform } X) \neq 1$   
**shows**  $\text{list-all } (\lambda \text{seg. } ccw' (\text{fst seg}) (\text{snd seg}) (\text{aform-val } e X))$   
 $(\text{map } (\text{pairself } (\text{mirror-point } (\text{fst } X)))) (\text{half-segments-of-aform } X))$   
**unfolding** *list-all-length*

**proof** *safe*

**let**  $?m = \text{map } (\text{pairself } (\text{mirror-point } (\text{fst } X))) (\text{half-segments-of-aform } X)$   
**fix**  $n$  **assume**  $n < \text{length } ?m$   
**hence**  $ccw' (\text{fst } (\text{half-segments-of-aform } X ! n)) (\text{snd } (\text{half-segments-of-aform } X ! n))$   
 $(\text{aform-val } (- e) X)$   
**using** *assms*  
**by** (auto dest!: nth-mem intro!: half-segments-of-aform-strict)  
**also have**  $\text{aform-val } (-e) X = 2 *_{\mathbb{R}} \text{fst } X - \text{aform-val } e X$   
**by** (auto simp: aform-val-def pdeus-val-sum algebra-simps scaleR-2 sum-negf)  
**finally have** *le*:  
 $ccw' (\text{fst } (\text{half-segments-of-aform } X ! n)) (\text{snd } (\text{half-segments-of-aform } X ! n))$   
 $(2 *_{\mathbb{R}} \text{fst } X - \text{aform-val } e X)$   
 .

**have** *eq*:  $(\text{map } (\text{pairself } (\text{mirror-point } (\text{fst } X))) (\text{half-segments-of-aform } X) ! n)$   
 $=$

```

    (2 *R fst X - fst ((half-segments-of-aform X) ! n),
     2 *R fst X - snd ((half-segments-of-aform X) ! n))
  using <n < length ?m>
  by (cases half-segments-of-aform X ! n) (auto simp add: mirror-point-def)
show ccw' (fst (?m ! n)) (snd (?m ! n)) (aform-val e X)
  using le
  unfolding eq
  by (auto simp: algebra-simps ccw'-def det3-def')
qed

```

**lemma** *last-half-segments*:

```

  assumes half-segments-of-aform X ≠ []
  shows snd (last (half-segments-of-aform X)) =
    mirror-point (fst X) (lowest-vertex (fst X, nlex-pdevs (snd X)))
  using assms
  by (auto simp add: half-segments-of-aform-def Let-def lowest-vertex-def mir-
    ror-point-def scaleR-2
    scaleR-sum-list[symmetric] last-polychain-of-sum-list-distinct-selsort inl-def
    sum-list-independent-pdevs sum-list-list-of-pdevs)

```

**lemma** *convex-polychain-map-mirror*:

```

  assumes convex-polychain hs
  shows convex-polychain (map (pairsel (mirror-point x)) hs)
proof (rule convex-polychainI)
qed (insert assms, auto simp: convex-polychain-def polychain-map-pairsel pair-
  self-apply
  mirror-point-def det3-def' algebra-simps)

```

**lemma** *ccw'-mirror-point0*:

```

ccw' (mirror-point x y) z w ⟷ ccw' y (mirror-point x z) (mirror-point x w)
by (metis ccw'-mirror-point3 mirror-point-self-inverse)

```

**lemma** *ccw'-sortedP-mirror*:

```

ccw'.sortedP x0 (map (mirror-point p0) xs) ⟷ ccw'.sortedP (mirror-point p0
x0) xs
by (induct xs)
  (simp-all add: linorder-list0.sortedP.Nil linorder-list0.sortedP-Cons-iff ccw'-mirror-point0)

```

**lemma** *ccw'-sortedP-mirror2*:

```

ccw'.sortedP (mirror-point p0 x0) (map (mirror-point p0) xs) ⟷ ccw'.sortedP
x0 xs
using ccw'-sortedP-mirror[of mirror-point p0 x0 p0 xs]
by simp

```

**lemma** *map-mirror-o-snd-polychain-of-eq*:  $\text{map} (\text{mirror-point } x0 \circ \text{snd}) (\text{polychain-of } y \text{ } xs) =$

```

  map snd (polychain-of (mirror-point x0 y) (map uminus xs))
by (induct xs arbitrary: x0 y) (auto simp: mirror-point-def o-def algebra-simps)

```

**lemma** *lowest-vertex-eq-mirror-last*:

*half-segments-of-aform*  $X \neq [] \implies$   
    (*lowest-vertex* (*fst*  $X$ , *nlex-pdevs* (*snd*  $X$ ))) =  
    *mirror-point* (*fst*  $X$ ) (*snd* (*last* (*half-segments-of-aform*  $X$ )))  
**using** *last-half-segments[of X]* **by** *simp*

**lemma** *snd-last*:  $xs \neq [] \implies \text{snd } (\text{last } xs) = \text{last } (\text{map } \text{snd } xs)$   
**by** (*induct xs*) *auto*

**lemma** *mirror-point-snd-last-eq-lowest*:

*half-segments-of-aform*  $X \neq [] \implies$   
    *mirror-point* (*fst*  $X$ ) (*last* (*map* *snd* (*half-segments-of-aform*  $X$ ))) =  
    *lowest-vertex* (*fst*  $X$ , *nlex-pdevs* (*snd*  $X$ ))  
**by** (*simp add: lowest-vertex-eq-mirror-last snd-last*)

**lemma** *lex-mirror-point*:  $\text{lex } (\text{mirror-point } x0 \ a) \ (\text{mirror-point } x0 \ b) \implies \text{lex } b \ a$   
**by** (*auto simp: mirror-point-def lex-def*)

**lemma** *ccw'-mirror-point*:

$\text{ccw}' \ (\text{mirror-point } x0 \ a) \ (\text{mirror-point } x0 \ b) \ (\text{mirror-point } x0 \ c) \implies \text{ccw}' \ a \ b \ c$   
**by** (*auto simp: mirror-point-def*)

**lemma** *inj-mirror-point*:  $\text{inj } (\text{mirror-point } (x::'a::\text{real-vector}))$   
**by** (*auto simp: mirror-point-def inj-on-def algebra-simps*)

**lemma**

*distinct-map-mirror-point-eq*:  
 $\text{distinct } (\text{map } (\text{mirror-point } (x::'a::\text{real-vector})) \ xs) = \text{distinct } xs$   
**by** (*auto simp: distinct-map intro!: subset-inj-on[OF inj-mirror-point]*)

**lemma** *eq-self-mirror-iff*: **fixes**  $x::'a::\text{real-vector}$  **shows**  $x = \text{mirror-point } y \ x \longleftrightarrow x = y$

**by** (*auto simp: mirror-point-def algebra-simps scaleR-2[symmetric]*)

## 10.9 Full Segments

**definition** *segments-of-aform::point aform*  $\Rightarrow (\text{point} * \text{point}) \text{ list}$

**where** *segments-of-aform*  $X =$

    (*let*  $hs = \text{half-segments-of-aform } X$  *in*  $hs \ @ \ \text{map } (\text{pairsel } (\text{mirror-point } (\text{fst } X))) \ hs$ )

**lemma** *segments-of-aform-strict*:

**assumes**  $e \in \text{UNIV} \rightarrow \{-1 <..< 1\}$

**assumes**  $\text{length } (\text{half-segments-of-aform } X) \neq 1$

**shows**  $\text{list-all } (\lambda \text{seg. } \text{ccw}' \ (\text{fst } \text{seg}) \ (\text{snd } \text{seg}) \ (\text{aform-val } e \ X)) \ (\text{segments-of-aform } X)$

**using** *assms*

**by** (*auto simp: segments-of-aform-def Let-def mirror-half-segments-of-aform half-segments-of-aform-strict-all*)



**lemma** *mirror-point-aform-val[simp]*:  $\text{mirror-point (fst X) (aform-val e X) = aform-val (- e) X}$   
**by** (*auto simp: mirror-point-def aform-val-def pdevs-val-sum algebra-simps scaleR-2 sum-negf*)

**lemma**  
*in-set-segments-of-aform-aform-valE*:  
**assumes**  $(x2, y2) \in \text{set (segments-of-aform X)}$   
**obtains**  $e$  **where**  $y2 = \text{aform-val e X}$   $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$   
**using** *assms*  
**proof** (*auto simp: segments-of-aform-def Let-def*)  
**assume**  $(x2, y2) \in \text{set (half-segments-of-aform X)}$   
**from** *in-set-half-segments-of-aform-aform-valE[OF this]*  
**obtain**  $e$  **where**  $y2 = \text{aform-val e X}$   $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$  **by** *auto*  
**thus** *?thesis ..*  
**next**  
**fix**  $a b aa ba$   
**assume**  $((a, b), aa, ba) \in \text{set (half-segments-of-aform X)}$   
**from** *in-set-half-segments-of-aform-aform-valE[OF this]*  
**obtain**  $e$  **where**  $e: (aa, ba) = \text{aform-val e X}$   $e \in \text{UNIV} \rightarrow \{-1 .. 1\}$  **by** *auto*  
**assume**  $y2 = \text{mirror-point (fst X) (aa, ba)}$   
**hence**  $y2 = \text{aform-val (-e) X}$   $(-e) \in \text{UNIV} \rightarrow \{-1 .. 1\}$  **using**  $e$  **by** *auto*  
**thus** *?thesis ..*  
**qed**

**lemma**  
*last-half-segments-eq-mirror-hd*:  
**assumes**  $\text{half-segments-of-aform X} \neq []$   
**shows**  $\text{snd (last (half-segments-of-aform X))} = \text{mirror-point (fst X) (fst (hd (half-segments-of-aform X)))}$   
**by** (*simp add: last-half-segments assms fst-hd-half-segments-of-aform*)

**lemma** *polychain-segments-of-aform*:  $\text{polychain (segments-of-aform X)}$   
**by** (*auto simp: segments-of-aform-def Let-def polychain-half-segments-of-aform polychain-map-pairself last-half-segments-eq-mirror-hd hd-map pairself-apply intro!: polychain-appendI*)

**lemma** *segments-of-aform-closed*:  
**assumes**  $\text{segments-of-aform X} \neq []$   
**shows**  $\text{fst (hd (segments-of-aform X))} = \text{snd (last (segments-of-aform X))}$   
**using** *assms*  
**by** (*auto simp: segments-of-aform-def Let-def fst-hd-half-segments-of-aform last-map pairself-apply last-half-segments mirror-point-def*)

**lemma** *convex-polychain-segments-of-aform*:  
**assumes**  $1 < \text{length (half-segments-of-aform X)}$   
**shows**  $\text{convex-polychain (segments-of-aform X)}$

**unfolding** *segments-of-aform-def Let-def*  
**using** *ccw-hd-last-half-segments-dirvec*[*OF assms*]  
**by** (*intro convex-polychain-appendI*)  
 (*auto*  
   *simp: convex-polychain-half-segments-of-aform convex-polychain-map-mirror*  
*dirvec-minus hd-map*  
   *pairself-apply last-half-segments mirror-point-def fst-hd-half-segments-of-aform*  
*det3-def'*  
   *algebra-simps ccw'-def*  
   *intro!: polychain-appendI polychain-half-segments-of-aform polychain-map-pairself*)

**lemma** *convex-polygon-segments-of-aform:*  
**assumes**  $1 < \text{length} (\text{half-segments-of-aform } X)$   
**shows** *convex-polygon* (*segments-of-aform*  $X$ )  
**proof** –  
**from** *assms* **have** *hne: half-segments-of-aform*  $X \neq []$   
**by** *auto*  
**from** *convex-polychain-segments-of-aform*[*OF assms*]  
**have** *convex-polychain* (*segments-of-aform*  $X$ ) .  
**thus** *?thesis*  
**by** (*auto simp: convex-polygon-def segments-of-aform-closed*)  
**qed**

**lemma**  
*previous-segments-of-aformE:*  
**assumes**  $(y, z) \in \text{set} (\text{segments-of-aform } X)$   
**obtains**  $x$  **where**  $(x, y) \in \text{set} (\text{segments-of-aform } X)$   
**proof** –  
**from** *assms* **have** *ne[simp]: segments-of-aform*  $X \neq []$   
**by** *auto*  
**from** *assms*  
**obtain**  $i$  **where**  $i < \text{length} (\text{segments-of-aform } X)$   $(\text{segments-of-aform } X) ! i =$   
 $(y, z)$   
**by** (*auto simp: in-set-conv-nth*)  
**show** *?thesis*  
**proof** (*cases i*)  
**case**  $0$   
**with** *segments-of-aform-closed*[*of X*] *assms*  
**have**  $(\text{fst} (\text{last} (\text{segments-of-aform } X)), y) \in \text{set} (\text{segments-of-aform } X)$   
**by** (*metis fst-conv hd-conv-nth i(2) last-in-set ne snd-conv surj-pair*)  
**thus** *?thesis ..*  
**next**  
**case** (*Suc j*)  
**have**  $(\text{fst} (\text{segments-of-aform } X ! j), \text{snd} (\text{segments-of-aform } X ! j)) \in$   
 $\text{set} (\text{segments-of-aform } X)$   
**using** *Suc i(1)* **by** *auto*  
**also**  
**from**  $i$  **have**  $y = \text{fst} (\text{segments-of-aform } X ! i)$   
**by** *auto*

**hence**  $\text{snd } (\text{segments-of-aform } X ! j) = y$   
**using**  $\text{polychain-segments-of-aform}[of\ X] i(1)\ \text{Suc}$   
**by**  $(\text{auto simp: polychain-def Suc})$   
**finally have**  $(\text{fst } (\text{segments-of-aform } X ! j), y) \in \text{set } (\text{segments-of-aform } X)$  .  
**thus**  $?thesis ..$   
**qed**  
**qed**

**lemma**  $\text{fst-compose-pairself: } \text{fst } o \text{ pairself } f = f o \text{ fst}$   
**and**  $\text{snd-compose-pairself: } \text{snd } o \text{ pairself } f = f o \text{ snd}$   
**by**  $(\text{auto simp: pairself-apply})$

**lemma**  $\text{in-set-butlastI: } xs \neq [] \implies x \in \text{set } xs \implies x \neq \text{last } xs \implies x \in \text{set } (\text{butlast } xs)$   
**by**  $(\text{induct } xs) (\text{auto split: if-splits})$

**lemma**  $\text{distinct-in-set-butlastD:}$   
 $x \in \text{set } (\text{butlast } xs) \implies \text{distinct } xs \implies x \neq \text{last } xs$   
**by**  $(\text{induct } xs) \text{ auto}$

**lemma**  $\text{distinct-in-set-tlD:}$   
 $x \in \text{set } (\text{tl } xs) \implies \text{distinct } xs \implies x \neq \text{hd } xs$   
**by**  $(\text{induct } xs) \text{ auto}$

**lemma**  $\text{ccw'-sortedP-snd-segments-of-aform:}$   
**assumes**  $\text{length } (\text{inl } (\text{snd } X)) > 1$   
**shows**  
 $\text{ccw'.sortedP } (\text{lowest-vertex } (\text{fst } X, \text{nlex-pdevs } (\text{snd } X)))$   
 $(\text{butlast } (\text{map } \text{snd } (\text{segments-of-aform } X)))$   
**proof** *cases*  
**assume**  $[] = \text{half-segments-of-aform } X$   
**from this show**  $?thesis$   
**by**  $(\text{simp add: segments-of-aform-def Let-def ccw'.sortedP.Nil})$   
**next**  
**assume**  $H: [] \neq \text{half-segments-of-aform } X$   
**let**  $?m = \text{mirror-point } (\text{fst } X)$   
**have**  $\text{ne: } \text{inl } (\text{snd } X) \neq []$  **using**  $\text{assms}$  **by**  $\text{auto}$   
**have**  $\text{ccw'.sortedP } (\text{lowest-vertex } (\text{fst } X, \text{nlex-pdevs } (\text{snd } X)))$   
 $(\text{map } \text{snd } (\text{half-segments-of-aform } X) @ \text{butlast } (\text{map } (?m \circ \text{snd})$   
 $(\text{half-segments-of-aform } X)))$   
**proof**  $(\text{rule } \text{ccw'.sortedP-appendI})$   
**show**  $\text{ccw'.sortedP } (\text{lowest-vertex } (\text{fst } X, \text{nlex-pdevs } (\text{snd } X))) (\text{map } \text{snd } (\text{half-segments-of-aform } X))$   
**by**  $(\text{rule } \text{ccw'-sortedP-snd-half-segments-of-aform})$   
**have**  $\text{butlast } (\text{map } (?m \circ \text{snd}) (\text{half-segments-of-aform } X)) =$   
 $\text{butlast}$   
 $(\text{map } (?m \circ \text{snd}) (\text{polychain-of } (\text{lowest-vertex } (\text{fst } X, \text{nlex-pdevs } (\text{snd } X))))$   
 $(\text{map } ((*_R) 2) (\text{ccw.selsort } 0 (\text{inl } (\text{snd } X))))))$   
**by**  $(\text{simp add: half-segments-of-aform-def})$

```

also have ... =
  map snd
    (butlast
      (polychain-of (?m (lowest-vertex (fst X, nlex-pdevs (snd X))))
        (map uminus (map ((*R) 2) (ccw.selsort 0 (inl (snd X)))))))
    (is - = map snd (butlast (polychain-of ?x ?xs)))
  by (simp add: map-mirror-o-snd-polychain-of-eq map-butlast)
also
{
  have ccw'.sortedP 0 ?xs
    by (intro ccw'-sortedP-uminus ccw'-sortedP-scaled-inl)
  moreover
  have ccw'.sortedP ?x (map snd (polychain-of ?x ?xs))
  unfolding ccw'-sortedP-mirror[symmetric] map-map map-mirror-o-snd-polychain-of-eq
  by (auto simp add: o-def intro!: ccw'-sortedP-polychain-of-snd ccw'-sortedP-scaled-inl)
  ultimately
  have ccw'.sortedP (snd (last (polychain-of ?x ?xs)))
    (map snd (butlast (polychain-of ?x ?xs)))
  by (rule ccw'-sortedP-convex-rotate-aux)
}
also have (snd (last (polychain-of ?x ?xs))) =
  ?m (last (map snd (half-segments-of-aform X)))
  by (simp add: half-segments-of-aform-def ne map-mirror-o-snd-polychain-of-eq
    last-map[symmetric, where f=?m]
    last-map[symmetric, where f=snd])
also have ... = lowest-vertex (fst X, nlex-pdevs (snd X))
  using ne H
  by (auto simp: lowest-vertex-eq-mirror-last snd-last)
finally show ccw'.sortedP (lowest-vertex (fst X, nlex-pdevs (snd X)))
  (butlast (map (?m ∘ snd) (half-segments-of-aform X))) .
next
fix x y
assume seg: x ∈ set (map snd (half-segments-of-aform X))
  and mseg: y ∈ set (butlast (map (?m ∘ snd) (half-segments-of-aform X)))
from seg assms have neq-Nil: inl (snd X) ≠ [] half-segments-of-aform X ≠ []
  by auto

from seg obtain a where a: (a, x) ∈ set (half-segments-of-aform X)
  by auto
from mseg obtain b
where mirror-y: (b, ?m y) ∈ set (butlast (half-segments-of-aform X))
  by (auto simp: map-butlast[symmetric])

let ?l = lowest-vertex (fst X, nlex-pdevs (snd X))
let ?ml = ?m ?l

have mirror-eq-last: ?ml = snd (last (half-segments-of-aform X))
  using seg H
  by (intro last-half-segments[symmetric]) simp

```

```

define r
  where r = ?l + (0, abs (snd x - snd ?l) + abs (snd y - snd ?l) + abs (snd
    ?ml - snd ?l) + 1)

  have d1: x ≠ r y ≠ r ?l ≠ r ?ml ≠ r
    by (auto simp: r-def plus-prod-def prod-eq-iff)
  have distinct (map (?m ∘ snd) (half-segments-of-aform X))
    unfolding map-comp-map[symmetric]
    unfolding o-def distinct-map-mirror-point-eq
    by (rule distinct-snd-half-segments)
  from distinct-in-set-butlastD[OF ⟨y ∈ ⟩ this]
  have ?l ≠ y
    by (simp add: neq-Nil lowest-vertex-eq-mirror-last last-map)
  moreover have ?l ≠ ?ml
    using neq-Nil by (auto simp add: eq-self-mirror-iff lowest-vertex-eq-center-iff
inl-def)
  ultimately
  have d2: ?l ≠ y ?l ≠ ?ml
    by auto

  have *: ccw' ?l (?m y) ?ml
    by (metis mirror-eq-last ccw'-half-segments-lowest-last mirror-y neq-Nil(1))
  have ccw' ?ml y ?l
    by (rule ccw'-mirror-point[of fst X]) (simp add: *)
  hence lmy: ccw' ?l ?ml y
    by (simp add: ccw'-def det3-def' algebra-simps)
  let ?ccw = ccw' (lowest-vertex (fst X, nlex-pdevs (snd X))) x y
  {
    assume x ≠ ?ml
    hence x-butlast: (a, x) ∈ set (butlast (half-segments-of-aform X))
      unfolding mirror-eq-last
      using a
      by (auto intro!: in-set-butlastI simp: prod-eq-iff)
    have ccw' ?l x ?ml
      by (metis mirror-eq-last ccw'-half-segments-lowest-last x-butlast neq-Nil(1))
  } note lxml = this
  {
    assume x = ?ml
    hence ?ccw
      by (simp add: lmy)
  } moreover {
    assume x ≠ ?ml y = ?ml
    hence ?ccw
      by (simp add: lxml)
  } moreover {
    assume d3: x ≠ ?ml y ≠ ?ml

  from ⟨x ∈ set ⟩

```

```

have  $x \in \text{set} (\text{map snd} (\text{half-segments-of-aform } X))$  by force
hence  $x \in \text{set} (\text{tl} (\text{map fst} (\text{half-segments-of-aform } X)))$ 
  using  $d3$ 
  unfolding  $\text{map-snd-half-segments-aux-eq}[OF \text{ neq-Nil}(2)]$ 
  by  $(\text{auto simp: mirror-eq-last})$ 
from  $\text{distinct-in-set-tlD}[OF \text{ this distinct-fst-half-segments}]$ 
have  $?l \neq x$ 
  by  $(\text{simp add: fst-hd-half-segments-of-aform neq-Nil hd-map})$ 

from  $\text{lxml}[OF \langle x \neq ?ml \rangle \langle \text{ccw}' ?l ?ml y \rangle]$ 
have  $d4: x \neq y$ 
  by  $(\text{rule neq-left-right-of lxml})$ 

have  $\text{distinct5 } x ?ml y r ?l$ 
  using  $d1 d2 \langle ?l \neq x \rangle d3 d4$ 
  by  $\text{simp-all}$ 
moreover
note -
moreover
have  $\text{lex } x ?l$ 
  by  $(\text{rule lex-half-segments-lowest-vertex}) \text{ fact}$ 
hence  $\text{ccw } ?l r x$ 
  unfolding  $r\text{-def}$  by  $(\text{rule lex-ccw-left}) \text{ simp}$ 
moreover
have  $\text{lex } ?ml ?l$ 
  using  $\text{last-in-set}[OF H[\text{symmetric}]]$ 
  by  $(\text{auto simp: mirror-eq-last intro: lex-half-segments-lowest-vertex})$ 
hence  $\text{ccw } ?l r ?ml$ 
  unfolding  $r\text{-def}$  by  $(\text{rule lex-ccw-left}) \text{ simp}$ 
moreover
have  $\text{lex} ( ?m (\text{lowest-vertex} (\text{fst } X, \text{nlex-pdevs} (\text{snd } X)))) ( ?m y)$ 
  using  $\text{mirror-y}$ 
  by  $(\text{force dest!: in-set-butlastD intro: lex-half-segments-last' simp: mirror-eq-last})$ 
hence  $\text{lex } y ?l$ 
  by  $(\text{rule lex-mirror-point})$ 
hence  $\text{ccw } ?l r y$ 
  unfolding  $r\text{-def}$  by  $(\text{rule lex-ccw-left}) \text{ simp}$ 
moreover
from  $\langle x \neq ?ml \rangle$  have  $\text{ccw } ?l x ?ml$ 
  by  $(\text{simp add: ccw-def lxml})$ 
moreover
from  $\text{lmy}$  have  $\text{ccw } ?l ?ml y$ 
  by  $(\text{simp add: ccw-def})$ 
ultimately
have  $\text{ccw } ?l x y$ 
  by  $(\text{rule ccw.transitive[where } S=\text{UNIV}]) \text{ simp}$ 

moreover

```

```

{
  have  $x \neq ?l$  using  $\langle ?l \neq x \rangle$  by simp
  moreover
  have  $\text{lex } (?m \ y) \ (?m \ ?ml)$ 
    using mirror-y
    by (force intro: lex-half-segments-lowest-vertex in-set-butlastD)
  hence  $\text{lex } ?ml \ y$ 
    by (rule lex-mirror-point)
  moreover
  from a have  $\text{lex } ?ml \ x$ 
    unfolding mirror-eq-last
    by (rule lex-half-segments-last)
  moreover note  $\langle \text{lex } y \ ?l \rangle \langle \text{lex } x \ ?l \rangle \langle \text{ccw}' \ ?l \ x \ ?ml \rangle \langle \text{ccw}' \ ?l \ ?ml \ y \rangle$ 
  ultimately
  have  $\text{ncoll}: \neg \text{coll } ?l \ x \ y$ 
    by (rule not-coll-ordered-lexI)
}
ultimately have  $?ccw$ 
  by (simp add: ccw-def)
} ultimately show  $?ccw$ 
  by blast
qed
thus  $?thesis$  using H
  by (simp add: segments-of-aform-def Let-def butlast-append snd-compose-pairself)
qed

```

```

lemma polychain-of-segments-of-aform1:
  assumes  $\text{length } (\text{segments-of-aform } X) = 1$ 
  shows False
  using assms
  by (auto simp: segments-of-aform-def Let-def half-segments-of-aform-def add-is-1 split: if-split-asm)

```

```

lemma polychain-of-segments-of-aform2:
  assumes  $\text{segments-of-aform } X = [x, y]$ 
  assumes  $\text{between } (\text{fst } x, \text{snd } x) \ p$ 
  shows  $p \in \text{convex hull set } (\text{map fst } (\text{segments-of-aform } X))$ 
proof -
  from polychain-segments-of-aform[of X] segments-of-aform-closed[of X] assms
  have  $\text{fst } y = \text{snd } x \ \text{snd } y = \text{fst } x$  by (simp-all add: polychain-def)
  thus  $?thesis$ 
    using assms
    by (cases x) (auto simp: between-mem-convex-hull)
qed

```

```

lemma append-eq-2:
  assumes  $\text{length } xs = \text{length } ys$ 
  shows  $xs @ ys = [x, y] \longleftrightarrow xs = [x] \wedge ys = [y]$ 
  using assms

```

**proof** (*cases xs*)  
**case** (*Cons z zs*)  
**thus** *?thesis using assms by (cases zs) auto*  
**qed simp**

**lemma** *segments-of-aform-line-segment*:  
**assumes** *segments-of-aform X = [x, y]*  
**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$   
**shows** *aform-val e X ∈ closed-segment (fst x) (snd x)*  
**proof** –  
**from** *pdevs-val-pdevs-of-list-inl2E[OF ‹e ∈ -›, of snd X]*  
**obtain** *e' where e': pdevs-val e (snd X) = pdevs-val e' (pdevs-of-list (inl (snd X)))*  
 $e' \in UNIV \rightarrow \{-1..1\}$ .  
**from** *e' have  $0 \leq 1 + e' 0$  by (auto simp: Pi-iff dest!: spec[where x=0])*  
**with** *assms e' show ?thesis*  
**by** (*auto simp: segments-of-aform-def Let-def append-eq-2 half-segments-of-aform-def polychain-of-singleton-iff mirror-point-def ccw.selsort-singleton-iff lowest-vertex-def aform-val-def sum-list-nlex-eq-sum-list-inl closed-segment-def Pi-iff intro!: exI[where x=(1 + e' 0) / 2] (auto simp: algebra-simps)*)  
**qed**

## 10.10 Continuous Generalization

**lemma** *LIMSEQ-minus-fract-mult*:  
 $(\lambda n. r * (1 - 1 / \text{real} (\text{Suc} (\text{Suc} n)))) \longrightarrow r$   
**by** (*rule tendsto-eq-rhs[OF tendsto-mult[where a=r and b = 1]] (auto simp: inverse-eq-divide[symmetric] simp del: of-nat-Suc intro: filterlim-compose[OF LIMSEQ-inverse-real-of-nat filterlim-Suc] tendsto-eq-intros)*)

**lemma** *det3-nonneg-segments-of-aform*:  
**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$   
**assumes**  $\text{length} (\text{half-segments-of-aform } X) \neq 1$   
**shows**  $\text{list-all} (\lambda \text{seg}. \text{det3} (\text{fst seg}) (\text{snd seg}) (\text{aform-val } e X) \geq 0) (\text{segments-of-aform } X)$   
**unfolding** *list-all-iff*  
**proof** *safe*  
**fix** *a b c d*  
**assume** *seg: ((a, b), c, d) ∈ set (segments-of-aform X) (is ?seg ∈ -)*  
**define** *normal-of-segment*  
**where**  $\text{normal-of-segment} = (\lambda((a0, a1), b0, b1). (b1 - a1, a0 - b0)::\text{real}*\text{real})$   
**define** *support-of-segment*  
**where**  $\text{support-of-segment} = (\lambda(a, b). \text{normal-of-segment } (a, b) \cdot a)$   
**have**  $\text{closed} ((\lambda x. x \cdot \text{normal-of-segment } ?\text{seg}) - \{..\text{support-of-segment } ?\text{seg}\})$   
*(is closed ?cl)*  
**by** (*auto intro!: continuous-intros closed-vimage*)  
**moreover**



```

define f where f n i = e i * ( 1 - 1 / (Suc (Suc n))) for n i
have  $\forall n. \text{aform-val } (f\ n) X \in ?cl$ 
proof
  fix n
  have  $f\ n \in UNIV \rightarrow \{-1 <..< 1\}$ 
  using assms
  by (auto simp: f-def Pi-iff intro!: less-one-multI minus-one-less-multI)
  from list-allD[OF segments-of-aform-strict[OF this assms(2)] seg]
  show  $\text{aform-val } (f\ n) X \in (\lambda x. x \cdot \text{normal-of-segment } ((a, b), c, d)) - \{ \dots \text{support-of-segment } ((a, b), c, d) \}$ 
  by (auto simp: list-all-iff normal-of-segment-def support-of-segment-def det3-def' field-simps inner-prod-def ccw'-def)
qed
moreover
have  $\bigwedge i. (\lambda n. f\ n\ i) \longrightarrow e\ i$ 
  unfolding f-def
  by (rule LIMSEQ-minus-fract-mult)
hence  $(\lambda n. \text{aform-val } (f\ n) X) \longrightarrow \text{aform-val } e\ X$ 
  by (auto simp: aform-val-def pdevs-val-sum intro!: tendsto-intros)
ultimately have  $\text{aform-val } e\ X \in ?cl$ 
  by (meson closed-sequentially)
thus  $\text{det3 } (fst\ ?seg) (snd\ ?seg) (\text{aform-val } e\ X) \geq 0$ 
  by (auto simp: list-all-iff normal-of-segment-def support-of-segment-def det3-def' field-simps inner-prod-def)
qed

```

```

lemma det3-nonneg-segments-of-aformI:
  assumes  $e \in UNIV \rightarrow \{-1 .. 1\}$ 
  assumes  $\text{length } (\text{half-segments-of-aform } X) \neq 1$ 
  assumes  $\text{seg} \in \text{set } (\text{segments-of-aform } X)$ 
  shows  $\text{det3 } (fst\ \text{seg}) (snd\ \text{seg}) (\text{aform-val } e\ X) \geq 0$ 
  using assms det3-nonneg-segments-of-aform by (auto simp: list-all-iff)

```

## 10.11 Intersection of Vertical Line with Segment

```

fun intersect-segment-xline'::nat  $\Rightarrow$  point * point  $\Rightarrow$  real  $\Rightarrow$  point option
  where intersect-segment-xline' p ((x0, y0), (x1, y1)) xl =
    (if  $x0 \leq xl \wedge xl \leq x1$  then
      if  $x0 = x1$  then Some ((min y0 y1), (max y0 y1))
      else
        let
          yl = truncate-down p (truncate-down p (real-divl p (y1 - y0) (x1 - x0)
            * (xl - x0)) + y0);
          yr = truncate-up p (truncate-up p (real-divr p (y1 - y0) (x1 - x0) * (xl
            - x0)) + y0)
          in Some (yl, yr)
        else None)

```

```

lemma norm-pair-fst0[simp]: norm (0, x) = norm x
  by (auto simp: norm-prod-def)

lemmas add-right-mono-le = order-trans[OF add-right-mono]
lemmas mult-right-mono-le = order-trans[OF mult-right-mono]

lemmas add-right-mono-ge = order-trans[OF - add-right-mono]
lemmas mult-right-mono-ge = order-trans[OF - mult-right-mono]

lemma dest-segment:
  fixes x b::real
  assumes (x, b) ∈ closed-segment (x0, y0) (x1, y1)
  assumes x0 ≠ x1
  shows b = (y1 - y0) * (x - x0) / (x1 - x0) + y0
proof -
  from assms obtain u where u: x = x0 *R (1 - u) + u * x1 b = y0 *R (1 -
u) + u * y1 0 ≤ u u ≤ 1
  by (auto simp: closed-segment-def algebra-simps)
  show b = (y1 - y0) * (x - x0) / (x1 - x0) + y0
  using assms by (auto simp: closed-segment-def field-simps u)
qed

lemma intersect-segment-xline':
  assumes intersect-segment-xline' prec (p0, p1) x = Some (m, M)
  shows closed-segment p0 p1 ∩ {p. fst p = x} ⊆ {(x, m) .. (x, M)}
  using assms
proof (cases p0)
  case (Pair x0 y0) note p0 = this
  show ?thesis
  proof (cases p1)
  case (Pair x1 y1) note p1 = this
  {
    assume x0 = x1 x = x1 m = min y0 y1 M = max y0 y1
    hence ?thesis
    by (force simp: abs-le-iff p0 p1 min-def max-def algebra-simps dest: seg-
ment-bound
      split: if-split-asm)
  } thus ?thesis
  using assms
  by (auto simp: abs-le-iff p0 p1 split: if-split-asm
    intro!: truncate-up-le truncate-down-le
    add-right-mono-le[OF truncate-down]
    add-right-mono-le[OF real-divl]
    add-right-mono-le[OF mult-right-mono-le[OF real-divl]]
    add-right-mono-ge[OF - truncate-up]
    add-right-mono-ge[OF - mult-right-mono-ge[OF - real-divr]]
    dest!: dest-segment)
qed
qed

```

**lemma**

*in-segment-fst-le:*

**fixes**  $x0\ x1\ b::real$

**assumes**  $x0 \leq x1$   $(x, b) \in closed\_segment\ (x0, y0)\ (x1, y1)$

**shows**  $x \leq x1$

**using** *assms* **using** *mult-left-mono*[*OF*  $\langle x0 \leq x1 \rangle$ , **where**  $c=1 - u$  **for**  $u$ ]

**by** (*force simp add: min-def max-def split: if-split-asm*

*simp add: algebra-simps not-le closed-segment-def*)

**lemma**

*in-segment-fst-ge:*

**fixes**  $x0\ x1\ b::real$

**assumes**  $x0 \leq x1$   $(x, b) \in closed\_segment\ (x0, y0)\ (x1, y1)$

**shows**  $x0 \leq x$

**using** *assms* **using** *mult-left-mono*[*OF*  $\langle x0 \leq x1 \rangle$ ]

**by** (*force simp add: min-def max-def split: if-split-asm*

*simp add: algebra-simps not-le closed-segment-def*)

**lemma** *intersect-segment-xline'-eq-None:*

**assumes** *intersect-segment-xline' prec*  $(p0, p1)\ x = None$

**assumes**  $fst\ p0 \leq fst\ p1$

**shows**  $closed\_segment\ p0\ p1 \cap \{p.\ fst\ p = x\} = \{\}$

**using** *assms*

**by** (*cases p0, cases p1*)

(*auto simp: abs-le-iff split: if-split-asm dest: in-segment-fst-le in-segment-fst-ge*)

**fun** *intersect-segment-xline*

**where** *intersect-segment-xline prec*  $((a, b), (c, d))\ x =$

(*if*  $a \leq c$  *then* *intersect-segment-xline' prec*  $((a, b), (c, d))\ x$

*else* *intersect-segment-xline' prec*  $((c, d), (a, b))\ x$ )

**lemma** *closed-segment-commute:*  $closed\_segment\ a\ b = closed\_segment\ b\ a$

**by** (*meson convex-contains-segment convex-closed-segment dual-order.antisym ends-in-segment*)

**lemma** *intersect-segment-xline:*

**assumes** *intersect-segment-xline prec*  $(p0, p1)\ x = Some\ (m, M)$

**shows**  $closed\_segment\ p0\ p1 \cap \{p.\ fst\ p = x\} \subseteq \{(x, m) .. (x, M)\}$

**using** *assms*

**by** (*cases p0, cases p1*)

(*auto simp: closed-segment-commute split: if-split-asm simp del: intersect-segment-xline'.simps dest!: intersect-segment-xline'*)

**lemma** *intersect-segment-xline-fst-snd:*

**assumes** *intersect-segment-xline prec seg*  $x = Some\ (m, M)$

**shows**  $closed\_segment\ (fst\ seg)\ (snd\ seg) \cap \{p.\ fst\ p = x\} \subseteq \{(x, m) .. (x, M)\}$

**using** *intersect-segment-xline*[*of prec fst seg snd seg x m M*] *assms*

**by** *simp*

**lemma** *intersect-segment-xline-eq-None*:  
**assumes** *intersect-segment-xline prec (p0, p1) x = None*  
**shows** *closed-segment p0 p1 ∩ {p. fst p = x} = {}*  
**using** *assms*  
**by** (*cases p0, cases p1*)  
*(auto simp: closed-segment-commute split: if-split-asm simp del: intersect-segment-xline'.simps*  
*dest!: intersect-segment-xline'-eq-None)*

**lemma** *inter-image-empty-iff*:  $(X \cap \{p. f p = x\} = \{\}) \longleftrightarrow (x \notin f \text{ ` } X)$   
**by** *auto*

**lemma** *fst-closed-segment[simp]*: *fst ` closed-segment a b = closed-segment (fst a)*  
*(fst b)*  
**by** (*force simp: closed-segment-def*)

**lemma** *intersect-segment-xline-eq-empty*:  
**fixes** *p0 p1::real \* real*  
**assumes** *closed-segment p0 p1 ∩ {p. fst p = x} = {}*  
**shows** *intersect-segment-xline prec (p0, p1) x = None*  
**using** *assms*  
**by** (*cases p0, cases p1*)  
*(auto simp: inter-image-empty-iff closed-segment-eq-real-ivl split: if-split-asm)*

**lemma** *intersect-segment-xline-le*:  
**assumes** *intersect-segment-xline prec y xl = Some (m0, M0)*  
**shows**  $m0 \leq M0$   
**using** *assms*  
**by** (*cases y*) (*auto simp: min-def split: if-split-asm intro!: truncate-up-le truncate-down-le*  
*order-trans[OF real-divl] order-trans[OF - real-divr] mult-right-mono*)

**lemma** *intersect-segment-xline-None-iff*:  
**fixes** *p0 p1::real \* real*  
**shows** *intersect-segment-xline prec (p0, p1) x = None*  $\longleftrightarrow$  *closed-segment p0 p1*  
 $\cap \{p. fst p = x\} = \{\}$   
**by** (*auto intro!: intersect-segment-xline-eq-empty dest!: intersect-segment-xline-eq-None*)

## 10.12 Bounds on Vertical Intersection with Oriented List of Segments

**primrec** *bound-intersect-2d* **where**  
*bound-intersect-2d prec [] x = None*  
| *bound-intersect-2d prec (X # Xs) xl =*  
*(case bound-intersect-2d prec Xs xl of*  
*None  $\Rightarrow$  intersect-segment-xline prec X xl*  
| *Some (m, M)  $\Rightarrow$*   
*(case intersect-segment-xline prec X xl of*  
*None  $\Rightarrow$  Some (m, M)*

|  $\text{Some } (m', M') \Rightarrow \text{Some } (\min m' m, \max M' M))$ )

**lemma**

*bound-intersect-2d-eq-None:*

**assumes** *bound-intersect-2d prec Xs x = None*

**assumes**  $X \in \text{set } Xs$

**shows** *intersect-segment-xline prec X x = None*

**using** *assms by (induct Xs) (auto split: option.split-asm)*

**lemma** *bound-intersect-2d-upper:*

**assumes** *bound-intersect-2d prec Xs x = Some (m, M)*

**obtains**  $X m'$  **where**  $X \in \text{set } Xs$  *intersect-segment-xline prec X x = Some (m', M)*

$\bigwedge X m' M'. X \in \text{set } Xs \implies \text{intersect-segment-xline prec X x = Some } (m', M') \implies M' \leq M$

**proof** *atomize-elim*

**show**  $\exists X m'. X \in \text{set } Xs \wedge \text{intersect-segment-xline prec X x = Some } (m', M) \wedge (\forall X m' M'. X \in \text{set } Xs \longrightarrow \text{intersect-segment-xline prec X x = Some } (m', M') \longrightarrow M' \leq M)$

**using** *assms*

**proof** (*induct Xs arbitrary: m M*)

**case Nil thus ?case by (simp add: bound-intersect-2d-def)**

**next**

**case (Cons X Xs)**

**show** *?case*

**proof** (*cases bound-intersect-2d prec Xs x*)

**case None**

**thus ?thesis using Cons**

**by** (*intro exI[where x=X] exI[where x=m]*)

(*simp del: intersect-segment-xline.simps add: bound-intersect-2d-eq-None*)

**next**

**case (Some mM)**

**note** *Some1=this*

**then obtain**  $m' M'$  **where**  $mM: mM = (m', M')$  **by** (*cases mM*)

**from** *Cons(1)[OF Some[unfolded mM]]*

**obtain**  $X' m''$  **where**  $X': X' \in \text{set } Xs$

**and**  $m'': \text{intersect-segment-xline prec X' x = Some } (m'', M')$

**and**  $\text{max}: \bigwedge X m' M'a. X \in \text{set } Xs \implies \text{intersect-segment-xline prec X x = Some } (m', M'a) \implies$

$M'a \leq M'$

**by** *auto*

**show** *?thesis*

**proof** (*cases intersect-segment-xline prec X x*)

**case None thus ?thesis using Some1 mM Cons(2) X' m'' max**

**by** (*intro exI[where x=X'] exI[where x=m'']*)

(*auto simp del: intersect-segment-xline.simps split: option.split-asm*)

**next**

**case (Some mM''')**

**thus ?thesis using Some1 mM Cons(2) X' m''**

```

    by (cases mM''')
      (force simp: max-def min-def simp del: intersect-segment-xline.simps
        split: option.split-asm if-split-asm dest!: max
        intro!: exI[where x = if M' ≥ snd mM''' then X' else X]
        exI[where x = if M' ≥ snd mM''' then m'' else fst mM'''])
  qed
qed
qed
qed

lemma bound-intersect-2d-lower:
  assumes bound-intersect-2d prec Xs x = Some (m, M)
  obtains X M' where X ∈ set Xs intersect-segment-xline prec X x = Some (m,
M')
  ∧ X m' M'. X ∈ set Xs ⇒ intersect-segment-xline prec X x = Some (m', M')
  ⇒ m ≤ m'
proof atomize-elim
  show ∃ X M'. X ∈ set Xs ∧ intersect-segment-xline prec X x = Some (m, M') ∧
    (∀ X m' M'. X ∈ set Xs → intersect-segment-xline prec X x = Some (m', M')
    → m ≤ m')
  using assms
  proof (induct Xs arbitrary: m M)
  case Nil thus ?case by (simp add: bound-intersect-2d-def)
  next
  case (Cons X Xs)
  show ?case
  proof (cases bound-intersect-2d prec Xs x)
  case None
  thus ?thesis using Cons
  by (intro exI[where x = X])
  (simp del: intersect-segment-xline.simps add: bound-intersect-2d-eq-None)
  next
  case (Some mM)
  note Some1=this
  then obtain m' M' where mM: mM = (m', M') by (cases mM)
  from Cons(1)[OF Some[unfolded mM]]
  obtain X' M'' where X': X' ∈ set Xs
  and M'': intersect-segment-xline prec X' x = Some (m', M'')
  and min: ∧ X m'a M'. X ∈ set Xs ⇒ intersect-segment-xline prec X x =
Some (m'a, M') ⇒
  m' ≤ m'a
  by auto
  show ?thesis
  proof (cases intersect-segment-xline prec X x)
  case None thus ?thesis using Some1 mM Cons(2) X' M'' min
  by (intro exI[where x = X] exI[where x = M''])
  (auto simp del: intersect-segment-xline.simps split: option.split-asm)
  next
  case (Some mM''')

```

```

thus ?thesis using Some1 mM Cons(2) X' M'' min
  by (cases mM''')
    (force simp: max-def min-def
      simp del: intersect-segment-xline.simps
      split: option.split-asm if-split-asm
      dest!: min
      intro!: exI[where x = if m' ≤ fst mM''' then X' else X]
      exI[where x = if m' ≤ fst mM''' then M'' else snd mM'''])
  qed
qed
qed
qed

```

```

lemma bound-intersect-2d:
  assumes bound-intersect-2d prec Xs x = Some (m, M)
  shows (⋃ (p1, p2) ∈ set Xs. closed-segment p1 p2) ∩ {p. fst p = x} ⊆ {(x, m)
  .. (x, M)}
  proof (clarsimp, safe)
    fix b x0 y0 x1 y1
    assume H: ((x0, y0), x1, y1) ∈ set Xs (x, b) ∈ closed-segment (x0, y0) (x1,
  y1)
    hence intersect-segment-xline prec ((x0, y0), x1, y1) x ≠ None
      by (intro notI)
      (auto dest!: intersect-segment-xline-eq-None simp del: intersect-segment-xline.simps)
    then obtain e f where ef: intersect-segment-xline prec ((x0, y0), x1, y1) x =
  Some (e, f)
      by auto
    with H have m ≤ e
      by (auto intro: bound-intersect-2d-lower[OF assms])
    also have ... ≤ b
      using intersect-segment-xline[OF ef] H
      by force
    finally show m ≤ b .
    have b ≤ f
      using intersect-segment-xline[OF ef] H
      by force
    also have ... ≤ M
      using H ef by (auto intro: bound-intersect-2d-upper[OF assms])
    finally show b ≤ M .
  qed

```

```

lemma bound-intersect-2d-eq-None-iff:
  bound-intersect-2d prec Xs x = None ⟷ (∀ X ∈ set Xs. intersect-segment-xline
  prec X x = None)
  by (induct Xs) (auto simp: split: option.split-asm)

```

```

lemma bound-intersect-2d-nonempty:
  assumes bound-intersect-2d prec Xs x = Some (m, M)
  shows (⋃ (p1, p2) ∈ set Xs. closed-segment p1 p2) ∩ {p. fst p = x} ≠ {}

```

**proof** –  
**from** *assms* **have** *bound-intersect-2d prec Xs x ≠ None* **by** *simp*  
**then obtain** *p1 p2* **where**  $(p1, p2) \in \text{set } Xs \text{ intersect-segment-xline prec } (p1, p2)$  *x ≠ None*  
**unfolding** *bound-intersect-2d-eq-None-iff* **by** *auto*  
**hence** *closed-segment p1 p2*  $\cap \{p. \text{fst } p = x\} \neq \{\}$   
**by** (*simp add: intersect-segment-xline-None-iff*)  
**thus** *?thesis* **using**  $\langle(p1, p2) \in \text{set } Xs\rangle$  **by** *auto*  
**qed**

**lemma** *bound-intersect-2d-le*:  
**assumes** *bound-intersect-2d prec Xs x = Some (m, M)* **shows**  $m \leq M$   
**proof** –  
**from** *bound-intersect-2d-nonempty[OF assms]* *bound-intersect-2d[OF assms]*  
**show**  $m \leq M$  **by** *auto*  
**qed**

### 10.13 Bounds on Vertical Intersection with General List of Segments

**definition** *bound-intersect-2d-ud prec X xl =*  
*(case segments-of-aform X of*  
 $\square \Rightarrow \text{if } \text{fst } (\text{fst } X) = xl \text{ then let } m = \text{snd } (\text{fst } X) \text{ in } \text{Some } (m, m) \text{ else } \text{None}$   
 $| [x, y] \Rightarrow \text{intersect-segment-xline prec } x \ xl$   
 $| xs \Rightarrow$   
*(case bound-intersect-2d prec (filter ( $\lambda((x1, y1), x2, y2). x1 < x2$ ) xs) xl of*  
 $\text{Some } (m, M') \Rightarrow$   
*(case bound-intersect-2d prec (filter ( $\lambda((x1, y1), x2, y2). x1 > x2$ ) xs) xl of*  
 $\text{Some } (m', M) \Rightarrow \text{Some } (\min m \ m', \max M \ M')$   
 $| \text{None} \Rightarrow \text{None}$   
 $| \text{None} \Rightarrow \text{None}))$

**lemma** *list-cases4*:  
 $\bigwedge x P. (x = \square \Rightarrow P) \Rightarrow (\bigwedge y. x = [y] \Rightarrow P) \Rightarrow$   
 $(\bigwedge y z. x = [y, z] \Rightarrow P) \Rightarrow$   
 $(\bigwedge w y z zs. x = w \# y \# z \# zs \Rightarrow P) \Rightarrow P$   
**by** (*metis list.exhaust*)

**lemma** *bound-intersect-2d-ud-segments-of-aform-le*:  
*bound-intersect-2d-ud prec X xl = Some (m0, M0)  $\Rightarrow m0 \leq M0$*   
**by** (*cases segments-of-aform X rule: list-cases4*)  
*(auto simp: Let-def bound-intersect-2d-ud-def min-def max-def intersect-segment-xline-le*  
*if-split-eq1 split: option.split-asm prod.split-asm list.split-asm*  
*dest!: bound-intersect-2d-le)*

**lemma** *pdevs-domain-eq-empty-iff[simp]*: *pdevs-domain (snd X) = {}  $\iff \text{snd } X = \text{zero-pdevs}$*   
**by** (*auto simp: intro!: pdevs-eqI*)



**lemma** *ccw-contr-on-line-left*:

**assumes**  $\det3 (a, b) (x, c) (x, d) \geq 0$   $a > x$

**shows**  $d \leq c$

**proof** –

**from** *assms* **have**  $d * (a - x) \leq c * (a - x)$

**by** (*auto simp: det3-def' algebra-simps*)

**with** *assms* **show**  $c \geq d$  **by** *auto*

**qed**

**lemma** *ccw-contr-on-line-right*:

**assumes**  $\det3 (a, b) (x, c) (x, d) \geq 0$   $a < x$

**shows**  $d \geq c$

**proof** –

**from** *assms* **have**  $c * (x - a) \leq d * (x - a)$

**by** (*auto simp: det3-def' algebra-simps*)

**with** *assms* **show**  $d \geq c$  **by** *auto*

**qed**

**lemma** *singleton-contrE*:

**assumes**  $\bigwedge x y. x \neq y \implies x \in X \implies y \in X \implies \text{False}$

**assumes**  $X \neq \{\}$

**obtains**  $x$  **where**  $X = \{x\}$

**using** *assms* **by** *blast*

**lemma** *segment-intersection-singleton*:

**fixes**  $xl$  **and**  $a b :: \text{real} * \text{real}$

**defines**  $i \equiv \text{closed-segment } a b \cap \{p. \text{fst } p = xl\}$

**assumes**  $ne1: \text{fst } a \neq \text{fst } b$

**assumes**  $upper: i \neq \{\}$

**obtains**  $p1$  **where**  $i = \{p1\}$

**proof** (*rule singleton-contrE[OF - upper]*)

**fix**  $x y$  **assume**  $H: x \neq y$   $x \in i$   $y \in i$

**then obtain**  $u v$  **where**  $uv: x = u *_R b + (1 - u) *_R a$   $y = v *_R b + (1 - v)$

$*_R a$

$0 \leq u$   $u \leq 1$   $0 \leq v$   $v \leq 1$

**by** (*auto simp add: closed-segment-def i-def field-simps*)

**then have**  $x + u *_R a = a + u *_R b$   $y + v *_R a = a + v *_R b$

**by** *simp-all*

**then have**  $\text{fst } (x + u *_R a) = \text{fst } (a + u *_R b)$   $\text{fst } (y + v *_R a) = \text{fst } (a + v$

$*_R b)$

**by** *simp-all*

**then have**  $u = v * (\text{fst } a - \text{fst } b) / (\text{fst } a - \text{fst } b)$

**using**  $ne1$   $H(2,3)$   $\langle 0 \leq u \rangle \langle u \leq 1 \rangle \langle 0 \leq v \rangle \langle v \leq 1 \rangle$

**by** (*simp add: closed-segment-def i-def field-simps*)

**then have**  $u = v$

**by** (*simp add: ne1*)

**then show**  $\text{False}$  **using**  $H uv$

**by** *simp*

**qed**

**lemma** *bound-intersect-2d-ud-segments-of-aform*:

**assumes** *bound-intersect-2d-ud prec X xl = Some (m0, M0)*

**assumes**  $e \in UNIV \rightarrow \{-1 .. 1\}$

**shows**  $\{aform\text{-val } e X\} \cap \{x. fst x = xl\} \subseteq \{(xl, m0) .. (xl, M0)\}$

**proof** *safe*

**fix**  $a b$

**assume** *safeassms: (a, b) = aform-val e X xl = fst (a, b)*

**hence** *fst-aform-val: fst (aform-val e X) = xl*

**by** *simp*

{

**assume** *len: length (segments-of-aform X) > 2*

**with** *assms obtain m M m' M'*

**where** *lb: bound-intersect-2d prec*

$[(x1, y1), x2, y2] \leftarrow \text{segments-of-aform } X . x1 < x2] xl = \text{Some } (m, M')$

**and** *ub: bound-intersect-2d prec*

$[(x1, y1), x2, y2] \leftarrow \text{segments-of-aform } X . x2 < x1] xl = \text{Some } (m', M)$

**and** *minmax: m0 = min m m' M0 = max M M'*

**by** (*auto simp: bound-intersect-2d-ud-def split: option.split-asm list.split-asm*)

)

**from** *bound-intersect-2d-upper[OF ub] obtain X1 m1*

**where** *upper:*

$X1 \in \text{set } [((x1, y1), x2, y2) \leftarrow \text{segments-of-aform } X . x2 < x1]$

*intersect-segment-xline prec X1 xl = Some (m1, M)*

**by** *metis*

**from** *bound-intersect-2d-lower[OF lb] obtain X2 M2*

**where** *lower:*

$X2 \in \text{set } [((x1, y1), x2, y2) \leftarrow \text{segments-of-aform } X . x1 < x2]$

*intersect-segment-xline prec X2 xl = Some (m, M2)*

**by** *metis*

**from** *upper(1) lower(1)*

**have**  $X1: X1 \in \text{set } (\text{segments-of-aform } X) \text{ fst } (fst X1) > \text{fst } (snd X1)$

**and**  $X2: X2 \in \text{set } (\text{segments-of-aform } X) \text{ fst } (fst X2) < \text{fst } (snd X2)$

**by** *auto*

**note** *upper-seg = intersect-segment-xline-fst-snd[OF upper(2)]*

**note** *lower-seg = intersect-segment-xline-fst-snd[OF lower(2)]*

**from** *len have lh: length (half-segments-of-aform X)  $\neq$  1*

**by** (*auto simp: segments-of-aform-def Let-def*)

**from**  $X1$  **have**  $ne1: \text{fst } (fst X1) \neq \text{fst } (snd X1)$

**by** *simp*

**moreover** **have**  $\text{closed-segment } (fst X1) (snd X1) \cap \{p. \text{fst } p = xl\} \neq \{\}$

**using** *upper(2)*

**by** (*simp add: intersect-segment-xline-None-iff[of prec, symmetric]*)

**ultimately obtain**  $p1$  **where**  $p1: \text{closed-segment } (fst X1) (snd X1) \cap \{p. \text{fst } p = xl\} = \{p1\}$

**by** (*rule segment-intersection-singleton*)

**then obtain**  $u$  **where**  $u: p1 = (1 - u) *_R \text{fst } X1 + u *_R (snd X1) \ 0 \leq u \ u \leq 1$

**by** (*auto simp: closed-segment-def algebra-simps*)

**have**  $coll1$ :  $det3$  (fst  $X1$ )  $p1$  (aform-val  $e$   $X$ )  $\geq 0$   
**and**  $coll1'$ :  $det3$   $p1$  (snd  $X1$ ) (aform-val  $e$   $X$ )  $\geq 0$   
**unfolding** *atomize-conj*  
**using**  $u$   
**by** (*cases*  $u = 0 \vee u = 1$ )  
*(auto simp: u(1) intro: det3-nonneg-scaleR-segment1 det3-nonneg-scaleR-segment2*  
*det3-nonneg-segments-of-aformI[OF  $\langle e \in \rightarrow$  lh  $X1(1)$ ])*

**from**  $X2$  **have**  $ne2$ :  $fst$  (fst  $X2$ )  $\neq$   $fst$  (snd  $X2$ ) **by** *simp*  
**moreover**  
**have** *closed-segment* (fst  $X2$ ) (snd  $X2$ )  $\cap$   $\{p. fst\ p = xl\} \neq \{\}$   
**using** *lower(2)*  
**by** (*simp add: intersect-segment-xline-None-iff[of prec, symmetric]*)  
**ultimately**  
**obtain**  $p2$  **where**  $p2$ : *closed-segment* (fst  $X2$ ) (snd  $X2$ )  $\cap$   $\{p. fst\ p = xl\} =$   
 $\{p2\}$   
**by** (*rule segment-intersection-singleton*)  
**then obtain**  $v$  **where**  $v$ :  $p2 = (1 - v) *_R$  *fst*  $X2 + v *_R$  (snd  $X2$ )  $0 \leq v$   $v$   
 $\leq 1$   
**by** (*auto simp: closed-segment-def algebra-simps*)  
**have**  $coll2$ :  $det3$  (fst  $X2$ )  $p2$  (aform-val  $e$   $X$ )  $\geq 0$   
**and**  $coll2'$ :  $det3$   $p2$  (snd  $X2$ ) (aform-val  $e$   $X$ )  $\geq 0$   
**unfolding** *atomize-conj*  
**using**  $v$   
**by** (*cases*  $v = 0 \vee v = 1$ )  
*(auto simp: v(1) intro: det3-nonneg-scaleR-segment1 det3-nonneg-scaleR-segment2*  
*det3-nonneg-segments-of-aformI[OF  $\langle e \in \rightarrow$  lh  $X2(1)$ ])*

**from** *in-set-segments-of-aform-aform-valE*  
 $[of\ fst\ X1\ snd\ X1\ X,\ unfolded\ prod.collapse,\ OF\ X1(1)]$   
**obtain**  $e1s$  **where**  $e1s$ :  $snd\ X1 = aform-val\ e1s\ X\ e1s \in UNIV \rightarrow \{-1..1\}$  .  
**from** *previous-segments-of-aformE*  
 $[of\ fst\ X1\ snd\ X1\ X,\ unfolded\ prod.collapse,\ OF\ X1(1)]$   
**obtain**  $fX0$  **where**  $(fX0,\ fst\ X1) \in set$  (*segments-of-aform*  $X$ ) .  
**from** *in-set-segments-of-aform-aform-valE*[*OF this*]  
**obtain**  $e1f$  **where**  $e1f$ :  $fst\ X1 = aform-val\ e1f\ X\ e1f \in UNIV \rightarrow \{-1..1\}$  .  
**have**  $p1 \in closed-segment$  (aform-val  $e1f\ X$ ) (aform-val  $e1s\ X$ )  
**using**  $p1$  **by** (*auto simp: e1s e1f*)  
**with** *segment-in-aform-val*[*OF e1s(2) e1f(2), of X*]  
**obtain**  $ep1$  **where**  $ep1$ :  $ep1 \in UNIV \rightarrow \{-1..1\}$   $p1 = aform-val\ ep1\ X$   
**by** (*auto simp: Affine-def valuate-def closed-segment-commute*)

**from** *in-set-segments-of-aform-aform-valE*  
 $[of\ fst\ X2\ snd\ X2\ X,\ unfolded\ prod.collapse,\ OF\ X2(1)]$   
**obtain**  $e2s$  **where**  $e2s$ :  $snd\ X2 = aform-val\ e2s\ X\ e2s \in UNIV \rightarrow \{-1..1\}$  .  
**from** *previous-segments-of-aformE*  
 $[of\ fst\ X2\ snd\ X2\ X,\ unfolded\ prod.collapse,\ OF\ X2(1)]$   
**obtain**  $fX02$  **where**  $(fX02,\ fst\ X2) \in set$  (*segments-of-aform*  $X$ ) .  
**from** *in-set-segments-of-aform-aform-valE*[*OF this*]

```

obtain e2f where e2f: fst X2 = aform-val e2f X e2f ∈ UNIV → {− 1..1} .
have p2 ∈ closed-segment (aform-val e2f X) (aform-val e2s X)
  using p2 by (auto simp: e2s e2f)
with segment-in-aform-val[OF e2f(2) e2s(2), of X]
obtain ep2 where ep2: ep2 ∈ UNIV → {−1 .. 1} p2 = aform-val ep2 X
  by (auto simp: Affine-def valuate-def)

from det3-nonneg-segments-of-aformI[OF ep2(1), of X (fst X1, snd X1), un-
folded prod.collapse,
  OF lh X1(1), unfolded ep2(2)[symmetric]]
have c2: det3 (fst X1) (snd X1) p2 ≥ 0 .
hence c12: det3 (fst X1) p1 p2 ≥ 0
  using u by (cases u = 0) (auto simp: u(1) det3-nonneg-scaleR-segment2)
from det3-nonneg-segments-of-aformI[OF ep1(1), of X (fst X2, snd X2), un-
folded prod.collapse,
  OF lh X2(1), unfolded ep1(2)[symmetric]]
have c1: det3 (fst X2) (snd X2) p1 ≥ 0 .
hence c21: det3 (fst X2) p2 p1 ≥ 0
  using v by (cases v = 0) (auto simp: v(1) det3-nonneg-scaleR-segment2)
from p1 p2 have p1p2xl: fst p1 = xl fst p2 = xl
  by (auto simp: det3-def')
from upper-seg p1 have snd p1 ≤ M by (auto simp: less-eq-prod-def)
from lower-seg p2 have m ≤ snd p2 by (auto simp: less-eq-prod-def)

{
  have *: (fst p1, snd (aform-val e X)) = aform-val e X
    by (simp add: prod-eq-iff p1p2xl fst-aform-val)
  hence coll:
    det3 (fst (fst X1), snd (fst X1)) (fst p1, snd p1) (fst p1, snd (aform-val e
X)) ≥ 0
    and coll':
    det3 (fst (snd X1), snd (snd X1)) (fst p1, snd (aform-val e X)) (fst p1, snd
p1) ≥ 0
    using coll1 coll1'
    by (auto simp: det3-rotate)
  have snd (aform-val e X) ≤ M
  proof (cases fst (fst X1) = xl)
  case False
    have fst (fst X1) ≥ fst p1
      unfolding u using X1
      by (auto simp: algebra-simps intro!: mult-left-mono u)
    moreover
    have fst (fst X1) ≠ fst p1
      using False
      by (simp add: p1p2xl)
    ultimately
    have fst (fst X1) > fst p1 by simp
    from ccw-contr-on-line-left[OF coll this]
    show ?thesis using ⟨snd p1 ≤ M⟩ by simp
}

```

```

next
  case True
  have  $\text{fst} (\text{snd } X1) * (1 - u) \leq \text{fst} (\text{fst } X1) * (1 - u)$ 
    using  $X1$   $u$ 
    by (auto simp: intro!: mult-right-mono)
  hence  $\text{fst} (\text{snd } X1) \leq \text{fst } p1$ 
    unfolding  $u$  by (auto simp: algebra-simps)
  moreover
  have  $\text{fst} (\text{snd } X1) \neq \text{fst } p1$ 
    using True  $ne1$ 
    by (simp add:  $p1p2xl$ )
  ultimately
  have  $\text{fst} (\text{snd } X1) < \text{fst } p1$  by simp
  from ccw-contr-on-line-right[OF  $coll'$  this]
  show ?thesis using  $\langle \text{snd } p1 \leq M \rangle$  by simp
qed
} moreover {
  have  $(\text{fst } p2, \text{snd} (\text{aform-val } e X)) = \text{aform-val } e X$ 
    by (simp add: prod-eq-iff  $p1p2xl$  fst-aform-val)
  hence coll:
     $\text{det3} (\text{fst} (\text{fst } X2), \text{snd} (\text{fst } X2)) (\text{fst } p2, \text{snd } p2) (\text{fst } p2, \text{snd} (\text{aform-val } e X)) \geq 0$ 
    and coll':
     $\text{det3} (\text{fst} (\text{snd } X2), \text{snd} (\text{snd } X2)) (\text{fst } p2, \text{snd} (\text{aform-val } e X)) (\text{fst } p2, \text{snd } p2) \geq 0$ 
    using coll2 coll2'
    by (auto simp: det3-rotate)
  have  $m \leq \text{snd} (\text{aform-val } e X)$ 
  proof (cases  $\text{fst} (\text{fst } X2) = xl$ )
  case False
  have  $\text{fst} (\text{fst } X2) \leq \text{fst } p2$ 
    unfolding  $v$  using  $X2$ 
    by (auto simp: algebra-simps intro!: mult-left-mono  $v$ )
  moreover
  have  $\text{fst} (\text{fst } X2) \neq \text{fst } p2$ 
    using False
    by (simp add:  $p1p2xl$ )
  ultimately
  have  $\text{fst} (\text{fst } X2) < \text{fst } p2$  by simp
  from ccw-contr-on-line-right[OF coll this]
  show ?thesis using  $\langle m \leq \text{snd } p2 \rangle$  by simp
}
next
  case True
  have  $(1 - v) * \text{fst} (\text{snd } X2) \geq (1 - v) * \text{fst} (\text{fst } X2)$ 
    using  $X2$   $v$ 
    by (auto simp: intro!: mult-left-mono)
  hence  $\text{fst} (\text{snd } X2) \geq \text{fst } p2$ 
    unfolding  $v$  by (auto simp: algebra-simps)
  moreover

```

```

    have fst (snd X2) ≠ fst p2
      using True ne2
      by (simp add: p1p2xl)
    ultimately
    have fst (snd X2) > fst p2 by simp
    from ccw-contr-on-line-left[OF coll' this]
    show ?thesis using ⟨m ≤ snd p2⟩ by simp
  qed
} ultimately have aform-val e X ∈ {(xl, m) .. (xl, M)}
  by (auto simp: less-eq-prod-def fst-aform-val)
hence aform-val e X ∈ {(xl, m0) .. (xl, M0)}
  by (auto simp: minmax less-eq-prod-def)
} moreover {
  assume length (segments-of-aform X) = 2
  then obtain a b where s: segments-of-aform X = [a, b]
    by (auto simp: numeral-2-eq-2 length-Suc-conv)
  from segments-of-aform-line-segment[OF this assms(2)]
  have aform-val e X ∈ closed-segment (fst a) (snd a) .
  moreover
  from assms
  have intersect-segment-xline prec a xl = Some (m0, M0)
    by (auto simp: bound-intersect-2d-ud-def s)
  note intersect-segment-xline-fst-snd[OF this]
  ultimately
  have aform-val e X ∈ {(xl, m0) .. (xl, M0)}
    by (auto simp: less-eq-prod-def fst-aform-val)
} moreover {
  assume length (segments-of-aform X) = 1
  from polychain-of-segments-of-aform1[OF this]
  have aform-val e X ∈ {(xl, m0) .. (xl, M0)} by auto
} moreover {
  assume len: length (segments-of-aform X) = 0
  hence independent-pdevs (map snd (list-of-pdevs (nlex-pdevs (snd X)))) = []
    by (simp add: segments-of-aform-def Let-def half-segments-of-aform-def inl-def)
  hence snd X = zero-pdevs
    by (subst (asm) independent-pdevs-eq-Nil-iff) (auto simp: list-all-iff list-of-pdevs-def)
  hence aform-val e X = fst X
    by (simp add: aform-val-def)
  with len assms have aform-val e X ∈ {(xl, m0) .. (xl, M0)}
    by (auto simp: bound-intersect-2d-ud-def Let-def split: if-split-asm)
} ultimately have aform-val e X ∈ {(xl, m0)..(xl, M0)}
  by arith
thus (a, b) ∈ {(fst (a, b), m0)..(fst (a, b), M0)}
  using safeassms
  by simp
qed

```

## 10.14 Approximation from Orthogonal Directions

**definition** *inter-aform-plane-ortho*:

$\text{nat} \Rightarrow 'a::\text{executable-euclidean-space aform} \Rightarrow 'a \Rightarrow \text{real} \Rightarrow 'a \text{ aform option}$   
**where**  
*inter-aform-plane-ortho*  $p \ Z \ n \ g = \text{do} \{$   
 $mMs \leftarrow \text{those} (\text{map} (\lambda b. \text{bound-intersect-2d-ud } p \ (\text{inner2-aform } Z \ n \ b) \ g)$   
*Basis-list*);  
 $\text{let } l = (\sum (b, m) \leftarrow \text{zip } \text{Basis-list} \ (\text{map } \text{fst } mMs). \ m \ *_R \ b);$   
 $\text{let } u = (\sum (b, M) \leftarrow \text{zip } \text{Basis-list} \ (\text{map } \text{snd } mMs). \ M \ *_R \ b);$   
 $\text{Some } (\text{aform-of-ivl } l \ u)$   
 $\}$

**lemma**

*those-eq-SomeD*:  
**assumes**  $\text{those} (\text{map } f \ xs) = \text{Some } ys$   
**shows**  $ys = \text{map} (\text{the } o \ f) \ xs \wedge (\forall i. \exists y. i < \text{length } xs \longrightarrow f \ (xs \ ! \ i) = \text{Some } y)$   
**using** *assms*  
**by** (*induct xs arbitrary: ys*) (*auto split: option.split-asm simp: o-def nth-Cons split: nat.split*)

**lemma**

*sum-list-zip-map*:  
**assumes** *distinct xs*  
**shows**  $(\sum (x, y) \leftarrow \text{zip } xs \ (\text{map } g \ xs). \ f \ x \ y) = (\sum x \in \text{set } xs. \ f \ x \ (g \ x))$   
**by** (*force simp add: sum-list-distinct-conv-sum-set assms distinct-zipI1 split-beta' in-set-zip in-set-conv-nth inj-on-convol-ident intro!: sum.reindex-cong[where l= $\lambda x. (x, g \ x)$ ]*)

**lemma**

*inter-aform-plane-ortho-overappr*:  
**assumes** *inter-aform-plane-ortho*  $p \ Z \ n \ g = \text{Some } X$   
**shows**  $\{x. \forall i \in \text{Basis}. x \cdot i \in \{y. (g, y) \in (\lambda x. (x \cdot n, x \cdot i)) \text{ 'Affine } Z\}\} \subseteq \text{Affine } X$

**proof** –

**let**  $?inter = (\lambda b. \text{bound-intersect-2d-ud } p \ (\text{inner2-aform } Z \ n \ b) \ g)$   
**obtain**  $xs$   
**where**  $xs: \text{those} (\text{map } ?inter \ \text{Basis-list}) = \text{Some } xs$   
**using** *assms* **by** (*cases those (map ?inter Basis-list)*) (*auto simp: inter-aform-plane-ortho-def*)

**from** *those-eq-SomeD*[*OF this*]

**obtain**  $y'$  **where**  $xs\text{-eq}: xs = \text{map} (\text{the } o \ ?inter) \ \text{Basis-list}$   
**and**  $y': \bigwedge i. i < \text{length} (\text{Basis-list}::'a \ \text{list}) \Longrightarrow ?inter \ (\text{Basis-list} \ ! \ i) = \text{Some} \ (y' \ i)$

**by** *metis*

**have**  $\forall (i::'a) \in \text{Basis}. \exists j < \text{length} (\text{Basis-list}::'a \ \text{list}). i = \text{Basis-list} \ ! \ j$

**by** (*metis Basis-list in-set-conv-nth*)

**then obtain**  $j$  **where**  $j$ :

$\bigwedge i::'a. i \in \text{Basis} \Longrightarrow j \ i < \text{length} (\text{Basis-list}::'a \ \text{list})$

$\bigwedge i::'a. i \in \text{Basis} \Longrightarrow i = \text{Basis-list} \ ! \ j \ i$

by *metis*  
**define**  $y$  **where**  $y = y' \circ j$   
**with**  $y' j$  **have**  $y: \bigwedge i. i \in \text{Basis} \implies ?\text{inter } i = \text{Some } (y \ i)$   
 by (*metis comp-def*)  
**hence**  $y\text{-le}: \bigwedge i. i \in \text{Basis} \implies \text{fst } (y \ i) \leq \text{snd } (y \ i)$   
 by (*auto intro!: bound-intersect-2d-ud-segments-of-aform-le*)  
**hence**  $(\sum b \in \text{Basis}. \text{fst } (y \ b) *_{\mathbb{R}} b) \leq (\sum b \in \text{Basis}. \text{snd } (y \ b) *_{\mathbb{R}} b)$   
 by (*auto simp: eucl-le[where 'a='a]*)  
**with** *assms* **have**  $X: \text{Affine } X = \{\sum b \in \text{Basis}. \text{fst } (y \ b) *_{\mathbb{R}} b .. \sum b \in \text{Basis}. \text{snd } (y \ b) *_{\mathbb{R}} b\}$   
 by (*auto simp: inter-aform-plane-ortho-def sum-list-zip-map xs xs-eq y Affine-aform-of-ivl*)

**show** *?thesis*

**proof** *safe*

**fix**  $x$  **assume**  $x: \forall i \in \text{Basis}. x \cdot i \in \{y. (g, y) \in (\lambda x. (x \cdot n, x \cdot i)) \text{ ' Affine } Z\}$   
 {  
**fix**  $i::'a$  **assume**  $i: i \in \text{Basis}$   
**from**  $x \ i$  **have**  $x\text{-in2}: (g, x \cdot i) \in (\lambda x. (x \cdot n, x \cdot i)) \text{ ' Affine } Z$   
 by *auto*  
**from**  $x\text{-in2}$  **obtain**  $e$   
**where**  $e: e \in \text{UNIV} \rightarrow \{-1..1\}$   
**and**  $g: g = \text{aform-val } e \ Z \cdot n$   
**and**  $x: x \cdot i = \text{aform-val } e \ Z \cdot i$   
 by (*auto simp: Affine-def evaluate-def*)  
**have**  $\{\text{aform-val } e \ (\text{inner2-aform } Z \ n \ i)\} = \{\text{aform-val } e \ (\text{inner2-aform } Z \ n \ i)\} \cap \{x. \text{fst } x = g\}$   
 by (*auto simp: g inner2-def*)  
**also**  
**from**  $y[\text{OF } \langle i \in \text{Basis} \rangle]$   
**have**  $?\text{inter } i = \text{Some } (\text{fst } (y \ i), \text{snd } (y \ i))$  **by** *simp*  
**note** *bound-intersect-2d-ud-segments-of-aform*[*OF this e*]  
**finally** **have**  $x \cdot i \in \{\text{fst } (y \ i) .. \text{snd } (y \ i)\}$   
 by (*auto simp: x inner2-def*)  
**} thus**  $x \in \text{Affine } X$   
**unfolding**  $X$   
 by (*auto simp: eucl-le[where 'a='a]*)

**qed**

**qed**

**lemma** *inter-proj-eq*:

**fixes**  $n \ g \ l$

**defines**  $G \equiv \{x. x \cdot n = g\}$

**shows**  $(\lambda x. x \cdot l) \text{ ' } (Z \cap G) =$

$\{y. (g, y) \in (\lambda x. (x \cdot n, x \cdot l)) \text{ ' } Z\}$

by (*auto simp: G-def*)

**lemma**

*inter-overappr*:

**fixes**  $n \ \gamma \ l$



**shows**  $Z \cap \{x. x \cdot n = g\} \subseteq \{x. \forall i \in \text{Basis}. x \cdot i \in \{y. (g, y) \in (\lambda x. (x \cdot n, x \cdot i)) ' Z\}\}$   
**by** *auto*

**lemma** *inter-inter-aform-plane-ortho*:  
**assumes** *inter-aform-plane-ortho*  $p$   $Z$   $n$   $g = \text{Some } X$   
**shows** *Affine*  $Z \cap \{x. x \cdot n = g\} \subseteq \text{Affine } X$   
**proof** –  
**note** *inter-overappr*[*of Affine*  $Z$   $n$   $g$ ]  
**also note** *inter-aform-plane-ortho-overappr*[*OF assms*]  
**finally show** *?thesis* .  
**qed**

## 10.15 “Completeness” of Intersection

**abbreviation** *encompasses*  $x$  *seg*  $\equiv \text{det3 } (fst \text{ seg}) (snd \text{ seg}) x \geq 0$

**lemma** *encompasses-cases*:  
*encompasses*  $x$  *seg*  $\vee$  *encompasses*  $x$  (*snd seg*, *fst seg*)  
**by** (*auto simp: det3-def' algebra-simps*)

**lemma** *list-all-encompasses-cases*:  
**assumes** *list-all* (*encompasses*  $p$ ) ( $x \# y \# zs$ )  
**obtains** *list-all* (*encompasses*  $p$ ) [ $x$ ,  $y$ , (*snd y*, *fst x*)]  
| *list-all* (*encompasses*  $p$ ) ((*fst x*, *snd y*)# $zs$ )  
**using** *encompasses-cases*  
**proof**  
**assume** *encompasses*  $p$  (*snd y*, *fst x*)  
**hence** *list-all* (*encompasses*  $p$ ) [ $x$ ,  $y$ , (*snd y*, *fst x*)]  
**using** *assms* **by** (*auto simp: list-all-iff*)  
**thus** *?thesis* ..

**next**  
**assume** *encompasses*  $p$  (*snd* (*snd y*, *fst x*), *fst* (*snd y*, *fst x*))  
**hence** *list-all* (*encompasses*  $p$ ) ((*fst x*, *snd y*)# $zs$ )  
**using** *assms* **by** (*auto simp: list-all-iff*)  
**thus** *?thesis* ..  
**qed**

**lemma** *triangle-encompassing-polychain-of*:  
**assumes**  $\text{det3 } p \ a \ b \geq 0$   $\text{det3 } p \ b \ c \geq 0$   $\text{det3 } p \ c \ a \geq 0$   
**assumes** *ccw' a b c*  
**shows**  $p \in \text{convex hull } \{a, b, c\}$   
**proof** –  
**from** *assms* **have** *nn*:  $\text{det3 } b \ c \ p \geq 0$   $\text{det3 } c \ a \ p \geq 0$   $\text{det3 } a \ b \ p \geq 0$   $\text{det3 } a \ b \ c$   
 $\geq 0$   
**by** (*auto simp: det3-def' algebra-simps*)  
**have**  $\text{det3 } a \ b \ c \ *_R \ p = \text{det3 } b \ c \ p \ *_R \ a + \text{det3 } c \ a \ p \ *_R \ b + \text{det3 } a \ b \ p \ *_R \ c$   
**by** (*auto simp: det3-def' algebra-simps prod-eq-iff*)  
**hence**  $\text{inverse } (\text{det3 } a \ b \ c) \ *_R (\text{det3 } a \ b \ c \ *_R \ p) =$

```

      inverse (det3 a b c) *R (det3 b c p *R a + det3 c a p *R b + det3 a b p *R c)
    by simp
  with assms have p-eq: p =
    (det3 b c p / det3 a b c) *R a + (det3 c a p / det3 a b c) *R b + (det3 a b p /
det3 a b c) *R c
    (is - = scaleR ?u - + scaleR ?v - + scaleR ?w -)
    by (simp add: inverse-eq-divide algebra-simps ccw'-def)
  have det-eq: det3 b c p / det3 a b c + det3 c a p / det3 a b c + det3 a b p / det3
a b c = 1
    using assms(4)
    by (simp add: add-divide-distrib[symmetric] det3-def' algebra-simps ccw'-def)
  show ?thesis
    unfolding convex-hull-3
    using assms(4)
    by (blast intro: exI[where x=?u] exI[where x=?v] exI[where x=?w]
      intro!: p-eq divide-nonneg-nonneg nn det-eq)
qed

```

**lemma** *encompasses-convex-polygon3*:

```

  assumes list-all (encompasses p) (x#y#z#zs)
  assumes convex-polygon (x#y#z#zs)
  assumes ccw'.sortedP (fst x) (map snd (butlast (x#y#z#zs)))
  shows p ∈ convex hull (set (map fst (x#y#z#zs)))
  using assms
proof (induct zs arbitrary: x y z p)
  case Nil
  thus ?case
    by (auto simp: det3-def' algebra-simps
      elim!: ccw'.sortedP-Cons ccw'.sortedP-Nil
      intro!: triangle-encompassing-polychain-of)

```

**next**

```

  case (Cons w ws)
  from Cons.prem(2) have snd y = fst z by auto
  from Cons.prem(1)
  show ?case
proof (rule list-all-encompasses-cases)
  assume list-all (encompasses p) [x, y, (snd y, fst x)]
  hence p ∈ convex hull {fst x, fst y, snd y}
    using Cons.prem
  by (auto simp: det3-def' algebra-simps
    elim!: ccw'.sortedP-Cons ccw'.sortedP-Nil
    intro!: triangle-encompassing-polychain-of)
  thus ?case
    by (rule rev-subsetD[OF - hull-mono]) (auto simp: ⟨snd y = fst z⟩)

```

**next**

```

  assume *: list-all (encompasses p) ((fst x, snd y) # z # w # ws)
  from Cons.prem
  have enc: ws ≠ [] ⇒ encompasses p (last ws)
    by (auto simp: list-all-iff)

```

```

have set (map fst ((fst x, snd y)#z#w#ws)) ⊆ set (map fst (x # y # z # w
# ws))
  by auto
moreover
{
  note *
  moreover
  have convex-polygon ((fst x, snd y) # z # w # ws)
    by (metis convex-polygon-skip Cons.prem1(2,3))
  moreover
  have ccw'.sortedP (fst (fst x, snd y)) (map snd (butlast ((fst x, snd y) # z #
w # ws)))
    using Cons.prem1(3)
    by (auto elim!: ccw'.sortedP-Cons intro!: ccw'.sortedP.Cons ccw'.sortedP.Nil
split: if-split-asm)
  ultimately have p ∈ convex hull set (map fst ((fst x, snd y)#z#w#ws))
    by (rule Cons.hyps)
}
ultimately
show p ∈ convex hull set (map fst (x # y # z # w # ws))
  by (rule subsetD[OF hull-mono])
qed
qed

```

**lemma** *segments-of-aform-empty-Affine-eq*:

```

assumes segments-of-aform X = []
shows Affine X = {fst X}
proof -
  have independent-pdevs (map snd (list-of-pdevs (nlex-pdevs (snd X)))) = [] ↔
(list-of-pdevs (nlex-pdevs (snd X))) = []
  by (subst independent-pdevs-eq-Nil-iff) (auto simp: list-all-iff list-of-pdevs-def )
  with assms show ?thesis
  by (force simp: aform-val-def list-of-pdevs-def Affine-def valuate-def segments-of-aform-def
Let-def half-segments-of-aform-def inl-def)
qed

```

**lemma** *not-segments-of-aform-singleton*: segments-of-aform X ≠ [x]

```

by (auto simp: segments-of-aform-def Let-def add-is-1 dest!: arg-cong[where
f=length])

```

**lemma** *encompasses-segments-of-aform-in-AffineI*:

```

assumes length (segments-of-aform X) > 2
assumes list-all (encompasses p) (segments-of-aform X)
shows p ∈ Affine X
proof -
  from assms(1) obtain x y z zs where eq: segments-of-aform X = x#y#z#zs
  by (cases segments-of-aform X rule: list-cases4) auto
  hence fst x = fst (hd (half-segments-of-aform X))
  by (metis segments-of-aform-def hd-append list.map-disc-iff list.sel(1))

```

```

also have ... = lowest-vertex (fst X, nlex-pdevs (snd X))
  using assms
  by (intro fst-hd-half-segments-of-aform) (auto simp: segments-of-aform-def)
finally have fstx: fst x = lowest-vertex (fst X, nlex-pdevs (snd X)) .
have p ∈ convex hull (set (map fst (segments-of-aform X)))
  using assms(2)
  unfolding eq
proof (rule encompasses-convex-polygon3)
  show convex-polygon (x # y # z # zs)
    using assms(1) unfolding eq[symmetric]
    by (intro convex-polygon-segments-of-aform) (simp add: segments-of-aform-def
Let-def)
  show ccw'.sortedP (fst x) (map snd (butlast (x # y # z # zs)))
    using assms(1)
    unfolding fstx map-butlast eq[symmetric]
    by (intro ccw'-sortedP-snd-segments-of-aform)
      (simp add: segments-of-aform-def Let-def half-segments-of-aform-def)
qed
also have ... ⊆ convex hull (Affine X)
proof (rule hull-mono, safe)
  fix a b assume (a, b) ∈ set (map fst (segments-of-aform X))
  then obtain c d where ((a, b), c, d) ∈ set (segments-of-aform X) by auto
  from previous-segments-of-aformE[OF this]
  obtain x where (x, a, b) ∈ set (segments-of-aform X) by auto
  from in-set-segments-of-aform-aform-valE[OF this]
  obtain e where (a, b) = aform-val e X e ∈ UNIV → {− 1..1} by auto
  thus (a, b) ∈ Affine X
    by (auto simp: Affine-def valuate-def image-iff)
qed
also have ... = Affine X
  by (simp add: convex-Affine convex-hull-eq)
finally show ?thesis .
qed
end

```

## 11 Implementation

```

theory Affine-Code
  imports
    Affine-Approximation
    Intersection
begin

```

Implementing partial deviations as sorted lists of coefficients.

### 11.1 Reverse Sorted, Distinct Association Lists

```

typedef (overloaded) ('a, 'b) slist =

```

$\{xs::('a::\text{linorder} \times 'b) \text{ list. distinct } (\text{map fst } xs) \wedge \text{sorted } (\text{rev } (\text{map fst } xs))\}$   
**by** (auto intro!: exI[**where**  $x=[]$ ])

**setup-lifting** *type-definition-slist*

**lift-definition** *map-of-slist*::(nat, 'a::zero) *slist*  $\Rightarrow$  nat  $\Rightarrow$  'a option **is** *map-of* .

**lemma** *finite-dom-map-of-slist*[intro, simp]: *finite* (dom (map-of-slist *xs*))  
**by** *transfer* (auto simp: *finite-dom-map-of*)

**abbreviation** *the-default* a *x*  $\equiv$  (case *x* of None  $\Rightarrow$  a | Some *b*  $\Rightarrow$  b)

**definition** *Pdevs-raw* *xs* *i* = *the-default* 0 (map-of *xs* *i*)

**lemma** *nonzeros-Pdevs-raw-subset*:  $\{i. \text{Pdevs-raw } xs \ i \neq 0\} \subseteq \text{dom } (\text{map-of } xs)$   
**unfolding** *Pdevs-raw-def*[*abs-def*]  
**by** *transfer* (auto simp: *Pdevs-raw-def split: option.split-asm*)

**lift-definition** *Pdevs*::(nat, 'a::zero) *slist*  $\Rightarrow$  'a *pdevs*  
**is** *Pdevs-raw*  
**by** (rule *finite-subset*[OF *nonzeros-Pdevs-raw-subset*]) (simp add: *finite-dom-map-of*)

**code-datatype** *Pdevs*

## 11.2 Degree

**primrec** *degree-list*::(nat  $\times$  'a::zero) *list*  $\Rightarrow$  nat **where**  
*degree-list* [] = 0  
| *degree-list* (*x*#:*xs*) = (if *snd* *x* = 0 then *degree-list* *xs* else *Suc* (*fst* *x*))

**lift-definition** *degree-slist*::(nat, 'a::zero) *slist*  $\Rightarrow$  nat **is** *degree-list* .

**lemma** *degree-list-eq-zeroD*:  
**assumes** *degree-list* *xs* = 0  
**shows** *the-default* 0 (map-of *xs* *i*) = 0  
**using** *assms*  
**by** (*induct* *xs*) (auto simp: *Pdevs-raw-def sorted-append split: if-split-asm*)

**lemma** *degree-slist-eq-zeroD*: *degree-slist* *xs* = 0  $\Longrightarrow$  *degree* (*Pdevs* *xs*) = 0  
**unfolding** *degree-eq-Suc-max*  
**by** *transfer* (auto dest: *degree-list-eq-zeroD simp: Pdevs-raw-def*)

**lemma** *degree-slist-eq-SucD*: *degree-slist* *xs* = *Suc* *n*  $\Longrightarrow$  *pdevs-apply* (*Pdevs* *xs*) *n*  
 $\neq$  0

**proof** (*transfer*, *goal-cases*)  
**case** (1 *xs* *n*)  
**thus** ?*case*  
**by** (*induct* *xs*)  
(auto simp: *Pdevs-raw-def sorted-append map-of-eq-None-iff*[*symmetric*])

*split: if-split-asm option.split-asm)*

**qed**

**lemma** *degree-slist-zero*:  
 $degree-slist\ xs = n \implies n \leq j \implies pdevs-apply\ (Pdevs\ xs)\ j = 0$   
**proof** (*transfer, goal-cases*)  
**case** ( $1\ xs\ n\ j$ )  
**thus** *?case*  
**by** (*induct xs*)  
*(auto simp: Pdevs-raw-def sorted-append split: if-split-asm option.split)*  
**qed**

**lemma** *compute-degree[code]*:  $degree\ (Pdevs\ xs) = degree-slist\ xs$   
**by** (*cases degree-slist xs*)  
*(auto dest: degree-slist-eq-zeroD degree-slist-eq-SucD intro!: degree-eqI degree-slist-zero)*

### 11.3 Auxiliary Definitions

**fun** *binop* **where**  
 $binop\ f\ z1\ z2\ []\ [] = []$   
 $| binop\ f\ z1\ z2\ ((i, x)\#xs)\ [] = (i, f\ x\ z2)\ \# binop\ f\ z1\ z2\ xs\ []$   
 $| binop\ f\ z1\ z2\ []\ ((i, y)\#ys) = (i, f\ z1\ y)\ \# binop\ f\ z1\ z2\ []\ ys$   
 $| binop\ f\ z1\ z2\ ((i, x)\#xs)\ ((j, y)\#ys) =$   
 $\quad (if\ (i = j)\ \ then\ (i, f\ x\ y)\ \# binop\ f\ z1\ z2\ xs\ ys$   
 $\quad \quad else\ if\ (i > j)\ then\ (i, f\ x\ z2)\ \# binop\ f\ z1\ z2\ xs\ ((j, y)\#ys)$   
 $\quad \quad else\ \quad \quad (j, f\ z1\ y)\ \# binop\ f\ z1\ z2\ ((i, x)\#xs)\ ys)$

**lemma** *set-binop-elemD1*:  
 $(a, b) \in set\ (binop\ f\ z1\ z2\ xs\ ys) \implies (a \in set\ (map\ fst\ xs) \vee a \in set\ (map\ fst\ ys))$   
**by** (*induct f z1 z2 xs ys rule: binop.induct*) *(auto split: if-split-asm)*

**lemma** *set-binop-elemD2*:  
 $(a, b) \in set\ (binop\ f\ z1\ z2\ xs\ ys) \implies$   
 $\quad (\exists x \in set\ (map\ snd\ xs). b = f\ x\ z2) \vee$   
 $\quad (\exists y \in set\ (map\ snd\ ys). b = f\ z1\ y) \vee$   
 $\quad (\exists x \in set\ (map\ snd\ xs). \exists y \in set\ (map\ snd\ ys). b = f\ x\ y)$   
**by** (*induct f z1 z2 xs ys rule: binop.induct*) *(auto split: if-split-asm)*

**abbreviation**  $rsorted \equiv \lambda x. sorted\ (rev\ x)$

**lemma** *rsorted-binop*:  
**fixes**  $xs::('a::linorder * 'b)\ list$  **and**  $ys::('a::linorder * 'c)\ list$   
**assumes** *rsorted ((map fst xs))*  
**assumes** *rsorted ((map fst ys))*  
**shows** *rsorted ((map fst (binop f z1 z2 xs ys)))*  
**using** *assms*  
**by** (*induct f z1 z2 xs ys rule: binop.induct*) *(force simp: sorted-append dest!: set-binop-elemD1)+*

**lemma** *distinct-binop*:  
**fixes**  $xs::('a::linorder * 'b) \text{ list}$  **and**  $ys::('a::linorder * 'c) \text{ list}$   
**assumes**  $\text{distinct } (\text{map } \text{fst } xs)$   
**assumes**  $\text{distinct } (\text{map } \text{fst } ys)$   
**assumes**  $\text{rsorted } ((\text{map } \text{fst } xs))$   
**assumes**  $\text{rsorted } ((\text{map } \text{fst } ys))$   
**shows**  $\text{distinct } (\text{map } \text{fst } (\text{binop } f \ z1 \ z2 \ xs \ ys))$   
**using**  $\text{assms}$   
**by**  $(\text{induct } f \ z1 \ z2 \ xs \ ys \ \text{rule: } \text{binop.induct})$   
 $(\text{force } \text{dest!}: \text{set-binop-elemD1 } \text{simp: } \text{sorted-append})+$

**lemma** *binop-plus*:  
**fixes**  $b::(\text{nat} * 'a::\text{euclidean-space}) \text{ list}$   
**shows**  
 $(\sum (i, y) \leftarrow \text{binop } (+) \ 0 \ 0 \ b \ \text{ba. } e \ i \ *_R \ y) = (\sum (i, y) \leftarrow b. e \ i \ *_R \ y) + (\sum (i, y) \leftarrow \text{ba. } e \ i \ *_R \ y)$   
**by**  $(\text{induct } (+) :: 'a \Rightarrow - \ 0 :: 'a \ 0 :: 'a \ b \ \text{ba} \ \text{rule: } \text{binop.induct})$   
 $(\text{auto } \text{simp: } \text{algebra-simps})$

**lemma** *binop-compose*:  
 $\text{binop } (\lambda x \ y. f \ (g \ x \ y)) \ z1 \ z2 \ xs \ ys = \text{map } (\text{apsnd } f) \ (\text{binop } g \ z1 \ z2 \ xs \ ys)$   
**by**  $(\text{induct } \lambda x \ y. f \ (g \ x \ y) \ z1 \ z2 \ xs \ ys \ \text{rule: } \text{binop.induct}) \ \text{auto}$

**lemma** *linear-cmul-left*[*intro, simp*]:  $\text{linear } ((* \ x::\text{real} \Rightarrow -)$   
**by**  $(\text{auto } \text{intro!}: \text{linearI } \text{simp: } \text{algebra-simps})$

**lemma** *length-merge-sorted-eq*:  
 $\text{length } (\text{binop } f \ z1 \ z2 \ xs \ ys) = \text{length } (\text{binop } g \ y1 \ y2 \ xs \ ys)$   
**by**  $(\text{induction } f \ z1 \ z2 \ xs \ ys \ \text{rule: } \text{binop.induct}) \ \text{auto}$

## 11.4 Pointwise Addition

**lift-definition**  $\text{add-slist}::(\text{nat}, 'a::\{\text{plus}, \text{zero}\}) \ \text{slist} \Rightarrow (\text{nat}, 'a) \ \text{slist} \Rightarrow (\text{nat}, 'a) \ \text{slist}$  **is**

$\lambda xs \ ys. \ \text{binop } (+) \ 0 \ 0 \ xs \ ys$   
**by**  $(\text{auto } \text{simp: } \text{intro!}: \text{distinct-binop } \text{rsorted-binop})$

**lemma** *map-of-binop*[*simp*]:  $\text{rsorted } (\text{map } \text{fst } xs) \Longrightarrow \text{rsorted } (\text{map } \text{fst } ys) \Longrightarrow$   
 $\text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{distinct } (\text{map } \text{fst } ys) \Longrightarrow$   
 $\text{map-of } (\text{binop } f \ z1 \ z2 \ xs \ ys) \ i =$   
 $(\text{case } \text{map-of } xs \ i \ \text{of}$   
 $\ \ \ \ \ \text{Some } x \Rightarrow \text{Some } (f \ x \ (\text{case } \text{map-of } ys \ i \ \text{of } \text{Some } x \Rightarrow x \ | \ \text{None} \Rightarrow z2))$   
 $\ | \ \text{None} \Rightarrow (\text{case } \text{map-of } ys \ i \ \text{of } \text{Some } y \Rightarrow \text{Some } (f \ z1 \ y) \ | \ \text{None} \Rightarrow \text{None}))$   
**by**  $(\text{induct } f \ z1 \ z2 \ xs \ ys \ \text{rule: } \text{binop.induct})$   
 $(\text{auto } \text{split: } \text{option.split } \text{option.split-asm } \text{simp: } \text{sorted-append})$

**lemma** *pdevs-apply-Pdevs-add-slist*[*simp*]:  
**fixes**  $xs \ ys::(\text{nat}, 'a::\text{monoid-add}) \ \text{slist}$

**shows**  $pdevs\text{-}apply (Pdevs (add\text{-}slist\ xs\ ys))\ i =$   
 $pdevs\text{-}apply (Pdevs\ xs)\ i + pdevs\text{-}apply (Pdevs\ ys)\ i$   
**by** (*transfer*) (*auto simp: Pdevs-raw-def split: option.split*)

**lemma** *compute-add-pdevs*[code]:  $add\text{-}pdevs (Pdevs\ xs)\ (Pdevs\ ys) = Pdevs (add\text{-}slist\ xs\ ys)$   
**by** (*rule pdevs-eqI*) *simp*

## 11.5 prod of pdevs

**lift-definition** *prod-slist*:: $(nat, 'a::zero)\ slist \Rightarrow (nat, 'b::zero)\ slist \Rightarrow (nat, ('a \times 'b))\ slist$  **is**  
 $\lambda xs\ ys. binop\ Pair\ 0\ 0\ xs\ ys$   
**by** (*auto simp: intro!: distinct-binop rsorted-binop*)

**lemma** *pdevs-apply-Pdevs-prod-slist*[*simp*]:  
 $pdevs\text{-}apply (Pdevs (prod\text{-}slist\ xs\ ys))\ i = (pdevs\text{-}apply (Pdevs\ xs)\ i, pdevs\text{-}apply (Pdevs\ ys)\ i)$   
**by** *transfer* (*auto simp: Pdevs-raw-def zero-prod-def split: option.splits*)

**lemma** *compute-prod-of-pdevs*[code]:  $prod\text{-}of\text{-}pdevs (Pdevs\ xs)\ (Pdevs\ ys) = Pdevs (prod\text{-}slist\ xs\ ys)$   
**by** (*rule pdevs-eqI*) *simp*

## 11.6 Set of Coefficients

**lift-definition** *set-slist*:: $(nat, 'a::real\text{-}vector)\ slist \Rightarrow (nat * 'a)\ set$  **is** *set* .

**lemma** *finite-set-slist*[*intro, simp*]:  $finite (set\text{-}slist\ xs)$   
**by** *transfer simp*

## 11.7 Domain

**lift-definition** *list-of-slist*:: $('a::linorder, 'b::zero)\ slist \Rightarrow ('a * 'b)\ list$   
**is**  $\lambda xs. filter (\lambda x. snd\ x \neq 0)\ xs$  .

**lemma** *compute-pdevs-domain*[code]:  $pdevs\text{-}domain (Pdevs\ xs) = set (map\ fst (list\text{-}of\text{-}slist\ xs))$   
**unfolding** *pdevs-domain-def*  
**by** *transfer* (*force simp: Pdevs-raw-def split: option.split-asm*)

**lemma** *sort-rev-eq-sort*:  $distinct\ xs \Longrightarrow sort (rev\ xs) = sort\ xs$   
**by** (*rule sorted-distinct-set-unique*) *auto*

**lemma** *compute-list-of-pdevs*[code]:  $list\text{-}of\text{-}pdevs (Pdevs\ xs) = list\text{-}of\text{-}slist\ xs$

**proof** –

**have**  $list\text{-}of\text{-}pdevs (Pdevs\ xs) =$   
 $map (\lambda i. (i, pdevs\text{-}apply (Pdevs\ xs)\ i)) (rev (sorted\text{-}list\text{-}of\text{-}set (pdevs\text{-}domain (Pdevs\ xs))))$   
**by** (*simp add: list-of-pdevs-def*)



**also have** (*sorted-list-of-set* (*pdevs-domain* (*Pdevs xs*))) = *rev* (*map fst* (*list-of-slist xs*))  
**unfolding** *compute-pdevs-domain sorted-list-of-set-sort-remdups*  
**proof** (*transfer, goal-cases*)  
**case** *prems: (1 xs)*  
**hence** *distinct: distinct* (*map fst [x←xs . snd x ≠ 0]*)  
**by** (*auto simp: filter-map distinct-map intro: subset-inj-on*)  
**with prems show** *?case*  
**using** *sort-rev-eq-sort[symmetric, OF distinct]*  
**by** (*auto simp: rev-map rev-filter distinct-map distinct-remdups-id intro!: sorted-sort-id sorted-filter*)  
**qed**  
**also**  
**have** *map* ( $\lambda i. (i, pdevs\ apply\ (Pdevs\ xs)\ i)$ ) (*rev ...*) = *list-of-slist xs*  
**proof** (*transfer, goal-cases*)  
**case** (*1 xs*)  
**thus** *?case*  
**unfolding** *Pdevs-raw-def o-def rev-rev-ident map-map*  
**by** (*subst map-cong[where g= $\lambda x. x$ ] (auto simp: map-filter-map-filter)*)  
**qed**  
**finally show** *?thesis .*  
**qed**

**lift-definition** *slist-of-pdevs::'a pdevs  $\Rightarrow$  (nat, 'a::real-vector) slist is list-of-pdevs*  
**by** (*auto simp: list-of-pdevs-def rev-map rev-filter filter-map o-def distinct-map image-def intro!: distinct-filter sorted-filter[of  $\lambda x. x$ , simplified]*)

## 11.8 Application

**lift-definition** *slist-apply::('a::linorder, 'b::zero) slist  $\Rightarrow$  'a  $\Rightarrow$  'b is*  
 $\lambda xs\ i. the\_default\ 0\ (map\ of\ xs\ i)$  .

**lemma** *compute-pdevs-apply[code]: pdevs-apply (Pdevs x) i = slist-apply x i*  
**by** *transfer (auto simp: Pdevs-raw-def)*

## 11.9 Total Deviation

**lift-definition** *tdev-slist::(nat, 'a::ordered-euclidean-space) slist  $\Rightarrow$  'a is*  
 $sum\_list\ o\ map\ (abs\ o\ snd)$  .

**lemma** *tdev-slist-sum: tdev-slist xs = sum (abs  $\circ$  snd) (set-slist xs)*  
**by** *transfer (auto simp: distinct-map sum-list-distinct-conv-sum-set[symmetric] o-def)*

**lemma** *pdevs-apply-set-slist: x  $\in$  set-slist xs  $\implies$  snd x = pdevs-apply (Pdevs xs) (fst x)*  
**by** *transfer (auto simp: Pdevs-raw-def)*

**lemma**

*tdev-list-eq-zeroI*:  
**shows**  $(\bigwedge i. \text{pdevs-apply } (P\text{devs } xs) i = 0) \implies \text{tdev-slist } xs = 0$   
**unfolding** *tdev-slist-sum*  
**by** (*auto simp: pdevs-apply-set-slist*)

**lemma** *inj-on-fst-set-slist*: *inj-on fst (set-slist xs)*  
**by** *transfer (simp add: distinct-map)*

**lemma** *pdevs-apply-Pdevs-eq-0*:  
*pdevs-apply (Pdevs xs) i = 0  $\longleftrightarrow$  (( $\forall x. (i, x) \in \text{set-slist } xs \longrightarrow x = 0$ ))*  
**by** *transfer (safe, auto simp: Pdevs-raw-def split: option.split)*

**lemma** *compute-tdev[code]*: *tdev (Pdevs xs) = tdev-slist xs*

**proof** –

**have** *tdev (Pdevs xs) = ( $\sum i < \text{degree } (P\text{devs } xs). |pdevs-apply (P\text{devs } xs) i|$ )*  
**by** (*simp add: tdev-def*)

**also have**  $\dots =$

$(\sum i < \text{degree } (P\text{devs } xs).$

*if pdevs-apply (Pdevs xs) i = 0 then 0 else |pdevs-apply (Pdevs xs) i|*)

**by** (*auto intro!: sum.cong*)

**also have**  $\dots =$

$(\sum i \in \{0..<\text{degree } (P\text{devs } xs)\} \cap \{x. \text{pdevs-apply } (P\text{devs } xs) x \neq 0\}.$

*|pdevs-apply (Pdevs xs) i|*)

**by** (*auto simp: sum.If-cases Collect-neg-eq atLeast0LessThan*)

**also have**  $\dots = (\sum x \in \text{fst ' set-slist } xs. |pdevs-apply (P\text{devs } xs) x|)$

**by** (*rule sum.mono-neutral-cong-left*)

(*force simp: pdevs-apply-Pdevs-eq-0 intro!: imageI degree-gt*)+

**also have**  $\dots = (\sum x \in \text{set-slist } xs. |pdevs-apply (P\text{devs } xs) (\text{fst } x)|)$

**by** (*rule sum.reindex-cong[of fst] (auto simp: inj-on-fst-set-slist)*)

**also have**  $\dots = \text{tdev-slist } xs$

**by** (*simp add: tdev-slist-sum pdevs-apply-set-slist*)

**finally show** *?thesis* .

**qed**

## 11.10 Minkowski Sum

**lemma** *dropWhile-rsorted-eq-filter*:

*rsorted (map fst xs)  $\implies$  dropWhile ( $\lambda(i, x). i \geq (m::\text{nat})$ ) xs = filter ( $\lambda(i, x). i < m$ ) xs*

(**is**  $- \implies ?lhs \text{ } xs = ?rhs \text{ } xs$ )

**proof** (*induct xs*)

**case** (*Cons x xs*)

**hence** *?rhs (x#xs) = ?lhs (x#xs)*

**by** (*auto simp: sorted-append filter-id-conv intro: sym*)

**thus** *?case ..*

**qed** *simp*

**lift-definition** *msum-slist::nat  $\Rightarrow$  (nat, 'a) slist  $\Rightarrow$  (nat, 'a) slist  $\Rightarrow$  (nat, 'a) slist*  
**is**  $\lambda m \text{ } xs \text{ } ys. \text{map } (\text{apfst } (\lambda n. n + m)) \text{ } ys @ \text{dropWhile } (\lambda(i, x). i \geq m) \text{ } xs$

**proof** (*safe, goal-cases*)  
**case** (*1 n l1 l2*)  
**then have**  $\text{set } (\text{dropWhile } (\lambda(i, x). n \leq i) l1) \subseteq \text{set } l1$   
**by** (*simp add: set-dropWhileD subrelI*)  
**with 1 show** *?case*  
**by** (*auto simp add: distinct-map add.commute [of - n] intro!: comp-inj-on intro: subset-inj-on*)  
*(simp add: dropWhile-rsorted-eq-filter)*  
**next**  
**case** *prems: (2 n l1 l2)*  
**hence**  $\text{sorted } (\text{map } ((\lambda na. na + n) \circ \text{fst}) (\text{rev } l2))$   
**by** (*simp add: sorted-iff-nth-mono rev-map*)  
**with prems show** *?case*  
**by** (*auto simp: sorted-append dropWhile-rsorted-eq-filter rev-map rev-filter sorted-filter*)  
**qed**

**lemma** *slist-apply-msum-slist:*

$\text{slist-apply } (\text{msum-slist } m \text{ } xs \text{ } ys) \ i =$   
*(if  $i < m$  then  $\text{slist-apply } xs \ i$  else  $\text{slist-apply } ys \ (i - m)$ )*

**proof** (*transfer, goal-cases*)

**case** *prems: (1 m xs ys i)*

**thus** *?case*

**proof** (*cases  $i \in \text{dom } (\text{map-of } (\text{map } (\lambda(x, y). (x + m, y)) \text{ } ys))$* )

**case** *False*

**have**  $\bigwedge a. i < m \implies i \notin \text{fst } \{x \in \text{set } xs. \text{case } x \text{ of } (i, x) \Rightarrow i < m\} \implies (i, a) \notin \text{set } xs$

$\bigwedge a. i \notin \text{fst } \{x \in \text{set } xs \implies (i, a) \notin \text{set } xs$

$\bigwedge a. m \leq i \implies i \notin \text{fst } \{(\lambda(x, y). (x + m, y)) \text{ } \text{set } ys \implies (i - m, a) \notin \text{set } ys$

**by** *force+*

**thus** *?thesis*

**using** *prems False*

**by** (*auto simp add: dropWhile-rsorted-eq-filter map-of-eq-None-iff distinct-map-fst-snd-eqD split: option.split dest!: map-of-SomeD*)

**qed** (*force simp: map-of-eq-None-iff distinct-map-fst-snd-eqD*)

*split: option.split*

*dest!: map-of-SomeD*)

**qed**

**lemma** *pdevs-apply-msum-slist:*

$\text{pdevs-apply } (Pdevs (\text{msum-slist } m \text{ } xs \text{ } ys)) \ i =$

*(if  $i < m$  then  $\text{pdevs-apply } (Pdevs \ xs) \ i$  else  $\text{pdevs-apply } (Pdevs \ ys) \ (i - m)$ )*

**by** (*auto simp: compute-pdevs-apply slist-apply-msum-slist*)

**lemma** *compute-msum-pdevs[`code`]:*  $\text{msum-pdevs } m \ (Pdevs \ xs) \ (Pdevs \ ys) = Pdevs (\text{msum-slist } m \ xs \ ys)$

**by** (*rule pdevs-eqI*) (*auto simp: pdevs-apply-msum-slist pdevs-apply-msum-pdevs*)

### 11.11 Unary Operations

**lift-definition** *map-slist*::('a ⇒ 'b) ⇒ (nat, 'a) slist ⇒ (nat, 'b) slist **is** λf. map (apsnd f)  
**by** *simp*

**lemma** *pdevs-apply-map-slist*:

*f 0 = 0 ⇒ pdevs-apply (Pdevs (map-slist f xs)) i = f (pdevs-apply (Pdevs xs) i)*

**by** *transfer*

(*force simp: Pdevs-raw-def map-of-eq-None-iff distinct-map-fst-snd-eqD image-def*

*split: option.split dest: distinct-map-fst-snd-eqD*)

**lemma** *compute-scaleR-pdves[code]*: *scaleR-pdevs r (Pdevs xs) = Pdevs (map-slist (λx. r \*<sub>R</sub> x) xs)*

**and** *compute-pdevs-scaleR[code]*: *pdevs-scaleR (Pdevs rs) x = Pdevs (map-slist (λr. r \*<sub>R</sub> x) rs)*

**and** *compute-uminus-pdevs[code]*: *uminus-pdevs (Pdevs xs) = Pdevs (map-slist (λx. - x) xs)*

**and** *compute-abs-pdevs[code]*: *abs-pdevs (Pdevs xs) = Pdevs (map-slist abs xs)*

**and** *compute-pdevs-inner[code]*: *pdevs-inner (Pdevs xs) b = Pdevs (map-slist (λx. x · b) xs)*

**and** *compute-pdevs-inner2[code]*:

*pdevs-inner2 (Pdevs xs) b c = Pdevs (map-slist (λx. (x · b, x · c)) xs)*

**and** *compute-inner-scaleR-pdevs[code]*:

*inner-scaleR-pdevs x (Pdevs ys) = Pdevs (map-slist (λy. (x · y) \*<sub>R</sub> y) ys)*

**and** *compute-trunc-pdevs[code]*:

*trunc-pdevs p (Pdevs xs) = Pdevs (map-slist (λx. eucl-truncate-down p x) xs)*

**and** *compute-trunc-err-pdevs[code]*:

*trunc-err-pdevs p (Pdevs xs) = Pdevs (map-slist (λx. eucl-truncate-down p x - x) xs)*

**by** (*auto intro!: pdevs-eqI simp: pdevs-apply-map-slist zero-prod-def abs-pdevs-def*)

### 11.12 Filter

**lift-definition** *filter-slist*::(nat ⇒ 'a ⇒ bool) ⇒ (nat, 'a) slist ⇒ (nat, 'a) slist

**is** λP xs. *filter* (λ(i, x). (P i x)) xs

**by** (*auto simp: o-def filter-map distinct-map rev-map rev-filter sorted-filter*

*intro: subset-inj-on*)

**lemma** *slist-apply-filter-slist*: *slist-apply (filter-slist P xs) i =*

(*if P i (slist-apply xs i) then slist-apply xs i else 0*)

**by** *transfer* (*force simp: Pdevs-raw-def o-def map-of-eq-None-iff distinct-map-fst-snd-eqD*

*dest: map-of-SomeD distinct-map-fst-snd-eqD split: option.split*)

**lemma** *pdevs-apply-filter-slist*: *pdevs-apply (Pdevs (filter-slist P xs)) i =*

(*if P i (pdevs-apply (Pdevs xs) i) then pdevs-apply (Pdevs xs) i else 0*)

**by** (*simp add: compute-pdevs-apply slist-apply-filter-slist*)

**lemma** *compute-filter-pdevs*[code]: *filter-pdevs P (Pdevs xs) = Pdevs (filter-slist P xs)*

**by** (*auto simp: pdevs-apply-filter-slist intro!: pdevs-eqI*)

### 11.13 Constant

**lift-definition** *zero-slist*::(*nat*, '*a*) *slist* is [] **by** *simp*

**lemma** *compute-zero-pdevs*[code]: *zero-pdevs = Pdevs (zero-slist)*

**by** *transfer (auto simp: Pdevs-raw-def)*

**lift-definition** *One-slist*::(*nat*, '*a*::*executable-euclidean-space*) *slist*

**is** *rev (zip [0..*length (Basis-list)::'a list*] (*Basis-list)::'a list*)*

**by** (*simp add: zip-rev[symmetric]*)

**lemma**

*map-of-rev-zip-upto-length-eq-nth*:

**assumes** *i < length B d = length B*

**shows** (*map-of (rev (zip [0..*d*] B)) i = Some (B ! i)*)

**proof** –

**have** *length (rev [0..*length B*]) = length (rev B)*

**by** *simp*

**from** *map-of-zip-is-Some[OF this, of i] assms*

**obtain** *y where y: map-of (zip (rev [0..*length B*]) (rev B)) i = Some y*

**by** (*auto simp: zip-rev*)

**hence** *y = B ! i*

**by** (*auto simp: in-set-zip rev-nth*)

**with** *y show ?thesis*

**by** (*simp add: zip-rev assms*)

**qed**

**lemma**

*map-of-rev-zip-upto-length-eq-None*:

**assumes**  $\neg i < \text{length } B$

**assumes** *d = length B*

**shows** (*map-of (rev (zip [0..*d*] B)) i = None*)

**using** *assms*

**by** (*auto simp: map-of-eq-None-iff in-set-zip*)

**lemma** *pdevs-apply-One-slist*:

*pdevs-apply (Pdevs One-slist) i =*

*(if i < length (Basis-list)::'a::executable-euclidean-space list)*

*then (Basis-list)::'a list ! i*

*else 0)*

**by** *transfer (auto simp: Pdevs-raw-def map-of-rev-zip-upto-length-eq-nth map-of-rev-zip-upto-length-eq-None in-set-zip split: option.split)*

**lemma** *compute-One-pdevs*[code]: *One-pdevs = Pdevs One-slist*

**by** (*rule pdevs-eqI (simp add: pdevs-apply-One-slist)*)

**lift-definition** *coord-slist*::*nat*  $\Rightarrow$  (*nat*, *real*) *slist* **is**  $\lambda i. [(i, 1)]$  **by** *simp*

**lemma** *compute-coord-pdevs*[*code*]: *coord-pdevs* *i* = *Pdevs* (*coord-slist* *i*)  
**by** *transfer* (*auto simp: Pdevs-raw-def*)

## 11.14 Update

**primrec** *update-list*::*nat*  $\Rightarrow$  '*a*  $\Rightarrow$  (*nat* \* '*a*) *list*  $\Rightarrow$  (*nat* \* '*a*) *list*

**where**

*update-list* *n* *x* [] = [(*n*, *x*)]  
| *update-list* *n* *x* (*y*#*ys*) =  
  (*if* *n* > *fst* *y* *then* (*n*, *x*)#*y*#*ys*  
  *else if* *n* = *fst* *y* *then* (*n*, *x*)#*ys*  
  *else* *y*#(*update-list* *n* *x* *ys*))

**lemma** *map-of-update-list*[*simp*]: *map-of* (*update-list* *n* *x* *ys*) = (*map-of* *ys*)(*n*:=*Some* *x*)  
**by** (*induct* *ys*) *auto*

**lemma** *in-set-update-listD*:

**assumes** *y*  $\in$  *set* (*update-list* *n* *x* *ys*)  
**shows** *y* = (*n*, *x*)  $\vee$  (*y*  $\in$  *set* *ys*)  
**using** *assms*  
**by** (*induct* *ys*) (*auto split: if-split-asm*)

**lemma** *in-set-update-listI*:

*y* = (*n*, *x*)  $\vee$  (*fst* *y*  $\neq$  *n*  $\wedge$  *y*  $\in$  *set* *ys*)  $\implies$  *y*  $\in$  *set* (*update-list* *n* *x* *ys*)  
**by** (*induct* *ys*) (*auto split: if-split-asm*)

**lemma** *in-set-update-list*: (*n*, *x*)  $\in$  *set* (*update-list* *n* *x* *xs*)

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

**lemma** *overwrite-update-list*: (*a*, *b*)  $\in$  *set* *xs*  $\implies$  (*a*, *b*)  $\notin$  *set* (*update-list* *n* *x* *xs*)  
 $\implies$  *a* = *n*

**by** (*induct* *xs*) (*auto split: if-split-asm*)

**lemma** *insert-update-list*:

*distinct* (*map* *fst* *xs*)  $\implies$  *rsorted* (*map* *fst* *xs*)  $\implies$  (*a*, *b*)  $\in$  *set* (*update-list* *a* *x* *xs*)  $\implies$  *b* = *x*  
**by** (*induct* *xs*) (*force split: if-split-asm simp: sorted-append*)+

**lemma** *set-update-list-eq*: *distinct* (*map* *fst* *xs*)  $\implies$  *rsorted* (*map* *fst* *xs*)  $\implies$

*set* (*update-list* *n* *x* *xs*) = *insert* (*n*, *x*) (*set* *xs* - {*x*. *fst* *x* = *n*})

**by** (*auto intro!: in-set-update-listI dest: in-set-update-listD simp: insert-update-list*)

**lift-definition** *update-slist*::*nat*  $\Rightarrow$  '*a*  $\Rightarrow$  (*nat*, '*a*) *slist*  $\Rightarrow$  (*nat*, '*a*) *slist* **is** *update-list*

**proof** *goal-cases*

**case** (1 n a l)  
**thus** ?case  
**by** (induct l) (force simp: sorted-append distinct-map not-less dest!: in-set-update-listD)+  
**qed**

**lemma** pdevs-apply-update-slist: pdevs-apply (Pdevs (update-slist n x xs)) i =  
 (if i = n then x else pdevs-apply (Pdevs xs) i)  
**by** transfer (auto simp: Pdevs-raw-def)

**lemma** compute-pdev-upd[code]: pdev-upd (Pdevs xs) n x = Pdevs (update-slist n x xs)  
**by** (rule pdevs-eqI) (auto simp: pdevs-apply-update-slist)

### 11.15 Approximate Total Deviation

**lift-definition** fold-slist::('a ⇒ 'b ⇒ 'b) ⇒ (nat, 'a::zero) slist ⇒ 'b ⇒ 'b  
 is λf xs z. fold (f o snd) (filter (λx. snd x ≠ 0) xs) z .

**lemma** Pdevs-raw-Cons:  
 Pdevs-raw ((a, b) # xs) = (λi. if i = a then b else Pdevs-raw xs i)  
**by** (auto simp: Pdevs-raw-def map-of-eq-None-iff  
 dest!: map-of-SomeD  
 split: option.split)

**lemma** zeros-aux: – (λi. if i = a then b else Pdevs-raw xs i) – ‘{0} ⊆  
 – Pdevs-raw xs – ‘{0} ∪ {a}  
**by** auto

**lemma** compute-tdev'[code]:  
 tdev' p (Pdevs xs) = fold-slist (λa b. eucl-truncate-up p (|a| + b)) xs 0  
**unfolding** tdev'-def sum-list'-def compute-list-of-pdevs  
**by** transfer (auto simp: o-def fold-map)

### 11.16 Equality

**lemma** slist-apply-list-of-slist-eq: slist-apply a i = the-default 0 (map-of (list-of-slist a) i)  
**by** (transfer)  
 (force split: option.split simp: map-of-eq-None-iff distinct-map-fst-snd-eqD  
 dest!: map-of-SomeD)

**lemma** compute-equal-pdevs[code]:  
 equal-class.equal (Pdevs a) (Pdevs b) ⟷ (list-of-slist a) = (list-of-slist b)  
**by** (auto intro!: pdevs-eqI simp: equal-pdevs-def compute-pdevs-apply slist-apply-list-of-slist-eq  
 compute-list-of-pdevs[symmetric])

### 11.17 From List of Generators

**lift-definition** slist-of-list::'a::zero list ⇒ (nat, 'a) slist  
 is λxs. rev (zip [0.. $\text{length } xs$ ] xs)

```

by (auto simp: rev-map[symmetric] )

lemma slist-apply-slist-of-list:
  slist-apply (slist-of-list xs) i = (if i < length xs then xs ! i else 0)
by transfer (auto simp: in-set-zip map-of-rev-zip-upto-length-eq-nth map-of-rev-zip-upto-length-eq-None)

lemma compute-pdevs-of-list[code]: pdevs-of-list xs = Pdevs (slist-of-list xs)
by (rule pdevs-eqI)
  (auto simp: compute-pdevs-apply slist-apply-slist-of-list pdevs-apply-pdevs-of-list)

abstraction function which can be used in code equations

lift-definition abs-slist-if::('a::linorder×'b) list ⇒ ('a, 'b) slist
  is λlist. if distinct (map fst list) ∧ rsorted (map fst list) then list else []
by auto

definition slist = Abs-slist

lemma [code-post]: Abs-slist = slist
by (simp add: slist-def)

lemma [code]: slist = (λxs.
  (if distinct (map fst xs) ∧ rsorted (map fst xs) then abs-slist-if xs else Code.abort
  (STR "'") (λ-. slist xs)))
by (auto simp add: slist-def abs-slist-if.abs-eq)

abbreviation pdevs ≡ (λx. Pdevs (slist x))

lift-definition nlex-slist::(nat, point) slist ⇒ (nat, point) slist is
  map (λ(i, x). (i, if lex 0 x then - x else x))
by (auto simp: o-def split-beta')

lemma Pdevs-raw-map: f 0 = 0 ⇒ Pdevs-raw (map (λ(i, x). (i, f x)) xs) i = f
(Pdevs-raw xs i)
by (auto simp: Pdevs-raw-def map-of-map split: option.split)

lemma compute-nlex-pdevs[code]: nlex-pdevs (Pdevs x) = Pdevs (nlex-slist x)
by transfer (auto simp: Pdevs-raw-map)

end

```

## 12 Optimizations for Code Integer

```

theory Optimize-Integer
imports
  Complex-Main
  HOL-Library.Code-Target-Numeral
begin

shallowly embed log and power

```



```

definition log2::int ⇒ int
  where log2 a = floor (log 2 (of-int a))

context includes integer.lifting begin

lift-definition log2-integer :: integer ⇒ integer
  is log2 :: int ⇒ int
  .

end

lemma [code]: log2 (int-of-integer a) = int-of-integer (log2-integer a)
  by (simp add: log2-integer.rep-eq)

code-printing
  constant log2-integer :: integer ⇒ - →
    (SML) IntInf.log2

definition power-int::int ⇒ int ⇒ int
  where power-int a b = a ^ (nat b)

context includes integer.lifting begin

lift-definition power-integer :: integer ⇒ integer ⇒ integer
  is power-int :: int ⇒ int ⇒ int
  .

end

code-printing
  constant power-integer :: integer ⇒ - ⇒ - →
    (SML) IntInf.pow ((-), (-))

lemma [code]: power-int (int-of-integer a) (int-of-integer b) = int-of-integer (power-integer
a b)
  by (simp add: power-integer.rep-eq)

end

```

## 13 Optimizations for Code Float

```

theory Optimize-Float
imports
  HOL-Library.Float
  Optimize-Integer
begin

```

```

lemma compute-bitlen[code]: bitlen a = (if a > 0 then log2 a + 1 else 0)
  by (simp add: bitlen-alt-def log2-def)

```

**lemma** *compute-float-plus*[code]:  $\text{Float } m1 \ e1 + \text{Float } m2 \ e2 =$   
*(if*  $m1 = 0$  *then*  $\text{Float } m2 \ e2$  *else if*  $m2 = 0$  *then*  $\text{Float } m1 \ e1$  *else*  
*if*  $e1 \leq e2$  *then*  $\text{Float } (m1 + m2 * \text{power-int } 2 \ (e2 - e1)) \ e1$   
*else*  $\text{Float } (m2 + m1 * \text{power-int } 2 \ (e1 - e2)) \ e2$ )  
**by** (*simp add: Float.compute-float-plus power-int-def*)

**lemma** *compute-real-of-float*[code]:  
 $\text{real-of-float } (\text{Float } m \ e) = (\text{if } e \geq 0 \text{ then } m * 2^{\text{nat } e} \text{ else } m / \text{power-int } 2$   
 $(-e))$   
**unfolding** *power-int-def*[*symmetric, of 2 e*]  
**using** *compute-real-of-float power-int-def* **by** *auto*

**lemma** *compute-float-down*[code]:  
 $\text{float-down } p \ (\text{Float } m \ e) =$   
*(if*  $p + e < 0$  *then*  $\text{Float } (m \text{ div } \text{power-int } 2 \ (-(p + e))) \ (-p)$  *else*  $\text{Float } m \ e$ )  
**by** (*simp add: Float.compute-float-down power-int-def*)

**lemma** *compute-lapprox-posrat*[code]:  
**fixes**  $\text{prec}::\text{nat}$  **and**  $x \ y::\text{nat}$   
**shows** *lapprox-posrat prec x y =*  
*(let*  
 $l = \text{rat-precision } \text{prec } x \ y;$   
 $d = \text{if } 0 \leq l \text{ then } \text{int } x * \text{power-int } 2 \ l \ \text{div } y \text{ else } \text{int } x \ \text{div } \text{power-int } 2 \ (-l)$   
*div } y*  
*in*  $\text{normfloat } (\text{Float } d \ (-l))$ )  
**by** (*auto simp add: Float.compute-lapprox-posrat power-int-def Let-def zdiv-int*  
*of-nat-power of-nat-mult*)

**lemma** *compute-rapprox-posrat*[code]:  
**fixes**  $\text{prec } x \ y$   
**defines**  $l \equiv \text{rat-precision } \text{prec } x \ y$   
**shows** *rapprox-posrat prec x y = (let*  
 $l = l;$   
 $(r, s) = \text{if } 0 \leq l \text{ then } (\text{int } x * \text{power-int } 2 \ l, \text{int } y) \text{ else } (\text{int } x, \text{int } y * \text{power-int}$   
 $2 \ (-l));$   
 $d = r \ \text{div } s;$   
 $m = r \ \text{mod } s$   
*in*  $\text{normfloat } (\text{Float } (d + (\text{if } m = 0 \vee y = 0 \text{ then } 0 \text{ else } 1)) \ (-l))$ )  
**by** (*auto simp add: l-def Float.compute-rapprox-posrat power-int-def Let-def zdiv-int*  
*of-nat-power of-nat-mult*)

**lemma** *compute-float-truncate-down*[code]:  
 $\text{float-round-down } \text{prec } (\text{Float } m \ e) = (\text{let } d = \text{bitlen } (\text{abs } m) - \text{int } \text{prec} - 1 \ \text{in}$   
*if*  $0 < d$  *then*  $\text{let } P = \text{power-int } 2 \ d; n = m \ \text{div } P \ \text{in } \text{Float } n \ (e + d)$   
*else*  $\text{Float } m \ e$ )  
**by** (*simp add: Float.compute-float-round-down power-int-def cong: if-cong*)

**lemma** *compute-int-floor-fl*[code]:

*int-floor-fl* (Float m e) = (if 0 ≤ e then m \* power-int 2 e else m div (power-int 2 (-e)))

**by** (simp add: Float.compute-int-floor-fl power-int-def)

**lemma** compute-floor-fl[code]:

*floor-fl* (Float m e) = (if 0 ≤ e then Float m e else Float (m div (power-int 2 ((-e)))) 0)

**by** (simp add: Float.compute-floor-fl power-int-def)

**end**

## 14 Target Language debug messages

**theory** Print

**imports**

*HOL-Decision-Procs.Approximation*

*Affine-Code*

*Show.Show-Instances*

*HOL-Library.Monad-Syntax*

*Optimize-Float*

**begin**

**hide-const** (open) floatarith.Max

### 14.1 Printing

Just for debugging purposes

**definition** *print*::String.literal ⇒ unit **where** *print* x = ()

**context** includes *integer.lifting* **begin**

**end**

**code-printing constant** *print* ↪ (SML) *TextIO.print*

### 14.2 Write to File

**definition** *file-output*::String.literal ⇒ ((String.literal ⇒ unit) ⇒ 'a) ⇒ 'a **where**

*file-output* - f = f (λ-. ())

**code-printing constant** *file-output* ↪ (SML) (fn s => fn f => File'-Stream.open'-output (fn os => f (File'-Stream.output os)) (Path.explode s))

### 14.3 Show for Floats

**definition** *showsp-float* :: float showsp

**where**

*showsp-float* p x = (

let m = mantissa x; e = exponent x in

if  $e = 0$  then `showsp-int p m` else `showsp-int p m o shows-string "*"2^" o showsp-int p e)`

**lemma** `show-law-float` [`show-law-intros`]:  
`show-law showsp-float r`  
**by** (`auto simp: showsp-float-def Let-def show-law-simps intro!: show-lawI`)

**lemma** `showsp-float-append` [`show-law-simps`]:  
`showsp-float p r (x @ y) = showsp-float p r x @ y`  
**by** (`intro show-lawD show-law-intros`)

**local-setup**  $\langle$ `Show-Generator.register-foreign-showsp @{typ float} @{term showsp-float}`  
`@{thm show-law-float}` $\rangle$

**derive** `show float`

## 14.4 Convert Float to Decimal number

type for decimal floating point numbers (currently just for printing, TODO?  
generalize theory `Float` for arbitrary base)

**datatype** `float10` = `Float10 (mantissa10: int) (exponent10: int)`  
**notation** `Float10` (**infix**  $\langle e \rangle$  999)

**partial-function** (`tailrec`) `normalize-float10`  
**where** [`code`]: `normalize-float10 f =`  
(`if mantissa10 f mod 10  $\neq$  0  $\vee$  mantissa10 f = 0 then f`  
`else normalize-float10 (Float10 (mantissa10 f div 20) (exponent10 f + 1))`)

### 14.4.1 Version that should be easy to prove correct, but slow!

**context includes** `floatarith-syntax` **begin**

**definition** `float-to-float10-approximation f = the`  
(`do {`  
`let (x, y) = (mantissa f * 1024, exponent f - 10);`  
`let p = nat (bitlen (abs x) + bitlen (abs y) + 80);` — FIXME: are there  
guarantees?  
`y-log  $\leftarrow$  approx p (Mult (Num (of-int y))`  
`((Mult (Ln (Num 2))`  
`(Inverse (Ln (Num 10)))))) [];`  
`let e-fl = floor-fl (lower y-log);`  
`let e = int-floor-fl e-fl;`  
`m  $\leftarrow$  approx p (Mult (Num (of-int x)) (Powr (Num 10) (Add (Var 0) (Minus`  
`(Num e-fl)))) [Some y-log];`  
`let ml = lower m;`  
`let mu = upper m;`  
`Some (normalize-float10 (Float10 (int-floor-fl ml) e), normalize-float10 (Float10`  
`(- int-floor-fl (- mu)) e)`  
`})`

**end**

**lemma** *compute-float-of*[code]: *float-of (real-of-float f) = f* **by** *simp*

## 14.5 Trusted, but faster version

TODO: this is the HOL version of the SML-code in Approximation.thy

**lemma** *prod-case-call-mono*[*partial-function-mono*]:  
*mono-tailrec* ( $\lambda f. (\text{let } (d, e) = a \text{ in } (\lambda y. f (c d e y))) b$ )  
**by** (*simp add: split-beta' call-mono*)

**definition** *divmod-int::int*  $\Rightarrow$  *int*  $\Rightarrow$  *int* \* *int*  
**where** *divmod-int a b = (a div b, a mod b)*

**partial-function** (*tailrec*) *f2f10-frac* **where**  
*f2f10-frac c p r digits cnt e =*  
    (*if* *r = 0* *then* (*digits, cnt, 0*)  
    *else if* *p = 0* *then* (*digits, cnt, r*)  
    *else* (*let*  
        (*d, r*) = *divmod-int (r \* 10) (power-int 2 (-e))*  
        *in f2f10-frac (c  $\vee$  d  $\neq$  0) (if d  $\neq$  0  $\vee$  c *then* *p - 1* *else* *p) r*  
        (*digits \* 10 + d) (cnt + 1) e*)  
**declare** *f2f10-frac.simps*[code]*

**definition** *float2-float10::int*  $\Rightarrow$  *bool*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\Rightarrow$  (*int* \* *int*) **where**  
*float2-float10 prec rd m e =* (  
    *let*  
        (*m, e*) = (*if* *e < 0* *then* (*m, e*) *else* (*m \* power-int 2 e, 0*));  
        *sgn = sgn m;*  
        *m = abs m;*  
  
        *round-down = (sgn = 1  $\wedge$  rd)  $\vee$  (sgn = -1  $\wedge$   $\neg$  rd);*  
  
        (*x, r*) = *divmod-int m ((power-int 2 (-e)))*;  
  
        *p = ((if* *x = 0* *then* *prec* *else* *prec - (log2 x + 1) \* 3) div 10 + 1*;  
  
        (*digits, e10, r*) = *if* *p > 0* *then* *f2f10-frac (x  $\neq$  0) p r 0 0 e* *else* (*0, 0, 0*);  
  
        *digits = if* *round-down  $\vee$  r = 0* *then* *digits* *else* *digits + 1*  
  
    *in (sgn \* (digits + x \* (power-int 10 e10)), -e10)*)

**definition** *lfloat10 r = (let f = float-of r in case-prod Float10 (float2-float10 20 True (mantissa f) (exponent f)))*

**definition** *ufloat10 r = (let f = float-of r in case-prod Float10 (float2-float10 20 False (mantissa f) (exponent f)))*

**partial-function** (*tailrec*) *digits*  
**where** [*code*]: *digits m ds = (if m = 0 then ds else digits (m div 10) (m mod 10 # ds))*

**primrec** *showsp-float10 :: float10 showsp*  
**where**

```

showsp-float10 p (Float10 m e) = (
  let
    ds = digits (nat (abs m)) [];
    d = int (length ds);
    e = e + d - 1;
    mp = take 1 ds;
    ms = drop 1 ds;
    ms = rev (dropWhile ((=) 0) (rev ms));
    show-digits = shows-list-gen (showsp-nat p) "0" "" "" "" ""
  in (if m < 0 then shows-string "-" else (λx. x)) o
    show-digits mp o
    (if ms = [] then (λx. x) else shows-string "." o show-digits ms) o
    (if e = 0 then (λx. x) else shows-string "e" o showsp-int p e))

```

**lemma** *show-law-float10-aux*:  
**fixes** *m e*  
**shows** *show-law showsp-float10 (Float10 m e)*  
**apply** (*rule show-lawI*)  
**unfolding** *showsp-float10.simps Let-def*  
**apply** (*simp add: show-law-simps*)  
**done**

**lemma** *show-law-float10 [show-law-intros]: show-law showsp-float10 r*  
**by** (*cases r*) (*auto simp: show-law-float10-aux*)

**lemma** *showsp-float10-append [show-law-simps]*:  
*showsp-float10 p r (x @ y) = showsp-float10 p r x @ y*  
**by** (*intro show-lawD show-law-intros*)

**local-setup**  $\langle$ *Show-Generator.register-foreign-showsp @{typ float10} @{term showsp-float10}*  
 $\rangle$   $\langle$ *thm show-law-float10* $\rangle$

**derive** *show float10*

**definition** *showsp-real p x = showsp-float10 p (lfloat10 x)*

**lemma** *show-law-real[show-law-intros]: show-law showsp-real x*  
**using** *show-law-float10[of lfloat10 x]*  
**by** (*auto simp: showsp-real-def[abs-def] Let-def show-law-def*  
*simp del: showsp-float10.simps intro!: show-law-intros*)

**local-setup**  $\langle$ *Show-Generator.register-foreign-showsp @{typ real} @{term showsp-real}*  
 $\rangle$   $\langle$ *thm show-law-real* $\rangle$

**derive** *show real*

## 14.6 gnuplot output

### 14.6.1 vector output of 2D zonotope

**fun** *polychain-of-segments::((real × real) × (real × real)) list ⇒ (real × real) list*  
**where**

*polychain-of-segments* [] = []  
| *polychain-of-segments* (((x0, y0), z)#segs) = (x0, y0)#z#map snd segs

**definition** *shows-segments-of-aform*

**where** *shows-segments-of-aform* a b xs color =  
*shows-list-gen* id "" "" " $\leftrightarrow$ " " $\leftrightarrow$ " (map (λ(x0, y0).  
  *shows-words* (map lfloat10 [x0, y0]) o *shows-space* o *shows-string* color)  
  (*polychain-of-segments* (*segments-of-aform* (prod-of-aforms (xs ! a) (xs ! b))))))

**abbreviation** *show-segments-of-aform* a b x c ≡ *shows-segments-of-aform* a b x c ""

**definition** *shows-box-of-aforms*— box and some further information

**where** *shows-box-of-aforms* (XS::real aform list) = (let  
  RS = map (Radius' 20) XS;  
  l = map (Inf-aform' 20) XS;  
  u = map (Sup-aform' 20) XS  
  in *shows-words*  
  (l @ u @ RS) o *shows-space* o  
  *shows* (card (⋃((λx. pdevs-domain (snd x)) ' (set XS))))  
)

**abbreviation** *show-box-of-aforms* x ≡ *shows-box-of-aforms* x ""

**definition** *pdevs-domains* ((XS::real aform list)) = (⋃((λx. pdevs-domain (snd x)) ' (set XS)))

**definition** *generators* XS =

(let  
  is = sorted-list-of-set (pdevs-domains XS);  
  rs = map (λi. (i, map (λx. pdevs-apply (snd x) i) XS)) is  
  in  
  (map fst XS, rs))

**definition** *shows-box-of-aforms-hr*— human readable

**where** *shows-box-of-aforms-hr* XS = (let  
  RS = map (Radius' 20) XS;  
  l = map (Inf-aform' 20) XS;  
  u = map (Sup-aform' 20) XS  
  in *shows-paren* (*shows-words* l) o *shows-string* " .. " o *shows-paren* (*shows-words*  
u) o  
  *shows-string* "; devs: " o *shows* (card (pdevs-domains XS)) o  
  *shows-string* "; tdev: " o *shows-paren* (*shows-words* RS)  
)

**abbreviation** *show-box-of-aforms-hr*  $x \equiv \text{shows-box-of-aforms-hr } x$  '''

**definition** *shows-aforms-hr*— human readable  
**where** *shows-aforms-hr*  $XS = \text{shows (generators } XS)$

**abbreviation** *show-aform-hr*  $x \equiv \text{shows-aforms-hr } x$  '''

**end**

## 15 Dyadic Rational Representation of Real

**theory** *Float-Real*

**imports**

*HOL-Library.Float*

*Optimize-Float*

**begin**

**code-datatype** *real-of-float*

**abbreviation**

*float-of-nat*  $:: \text{nat} \Rightarrow \text{float}$

**where**

*float-of-nat*  $\equiv \text{of-nat}$

**abbreviation**

*float-of-int*  $:: \text{int} \Rightarrow \text{float}$

**where**

*float-of-int*  $\equiv \text{of-int}$

Collapse nested embeddings

Operations

Undo code setup for *Ratreal*.

**lemma** *of-rat-numeral-eq* [*code-abbrev*]:

*real-of-float (numeral w) = Ratreal (numeral w)*

**by** *simp*

**lemma** *zero-real-code* [*code*]:

$0 = \text{real-of-float } 0$

**by** *simp*

**lemma** *one-real-code* [*code*]:

$1 = \text{real-of-float } 1$

**by** *simp*

**lemma** [*code-abbrev*]:

$(\text{real-of-float (of-int } a) :: \text{real}) = (\text{Ratreal (Rat.of-int } a) :: \text{real})$

**by** (*auto simp: Rat.of-int-def*)



**lemma** [code-abbrev]:  
*real-of-float 0*  $\equiv$  *Ratreal 0*  
**by** *simp*

**lemma** [code-abbrev]:  
*real-of-float 1* = *Ratreal 1*  
**by** *simp*

**lemmas** *compute-real-of-float*[code del]

**lemmas** [code del] =  
*real-equal-code*  
*real-less-eq-code*  
*real-less-code*  
*real-plus-code*  
*real-times-code*  
*real-uminus-code*  
*real-minus-code*  
*real-inverse-code*  
*real-divide-code*  
*real-floor-code*  
*Float.compute-truncate-down*  
*Float.compute-truncate-up*

**lemma** *real-equal-code* [code]:  
*HOL.equal (real-of-float x) (real-of-float y)*  $\longleftrightarrow$  *HOL.equal x y*  
**by** (*metis (poly-guards-query) equal real-of-float-inverse*)

**abbreviation** *FloatR::int $\Rightarrow$ int $\Rightarrow$ real* **where**  
*FloatR a b*  $\equiv$  *real-of-float (Float a b)*

**lemma** *real-less-eq-code'* [code]: *real-of-float x*  $\leq$  *real-of-float y*  $\longleftrightarrow$   $x \leq y$   
**and** *real-less-code'* [code]: *real-of-float x*  $<$  *real-of-float y*  $\longleftrightarrow$   $x < y$   
**and** *real-plus-code'* [code]: *real-of-float x* + *real-of-float y* = *real-of-float (x + y)*  
**and** *real-times-code'* [code]: *real-of-float x* \* *real-of-float y* = *real-of-float (x \* y)*  
**and** *real-uminus-code'* [code]:  $-$  *real-of-float x* = *real-of-float (- x)*  
**and** *real-minus-code'* [code]: *real-of-float x* - *real-of-float y* = *real-of-float (x - y)*  
**and** *real-inverse-code'* [code]: *inverse (FloatR a b)* =  
 (*if* *FloatR a b* = 2 *then* *FloatR 1 (-1)* *else*  
*if* *a* = 1 *then* *FloatR 1 (- b)* *else*  
*Code.abort (STR "inverse not of 2")* ( $\lambda$ -. *inverse (FloatR a b)*))  
**and** *real-divide-code'* [code]: *FloatR a b / FloatR c d* =  
 (*if* *FloatR c d* = 2 *then if* *a mod 2* = 0 *then* *FloatR (a div 2) b* *else* *FloatR a*  
 (*b - 1*) *else*  
*if* *c* = 1 *then* *FloatR a (b - d)* *else*  
*Code.abort (STR "division not by 2")* ( $\lambda$ -. *FloatR a b / FloatR c d*))  
**and** *real-floor-code'* [code]: *floor (real-of-float x)* = *int-floor-fl x*

**and** *real-abs-code'* [code]:  $\text{abs} (\text{real-of-float } x) = \text{real-of-float} (\text{abs } x)$   
**by** (*auto simp add: int-floor-fl.rep-eq powr-diff powr-minus inverse-eq-divide*)

**lemma** *compute-round-down*[code]:  $\text{round-down } \text{prec} (\text{real-of-float } f) = \text{real-of-float} (\text{float-down } \text{prec } f)$   
**by** *simp*

**lemma** *compute-round-up*[code]:  $\text{round-up } \text{prec} (\text{real-of-float } f) = \text{real-of-float} (\text{float-up } \text{prec } f)$   
**by** *simp*

**lemma** *compute-truncate-down*[code]:  
 $\text{truncate-down } \text{prec} (\text{real-of-float } f) = \text{real-of-float} (\text{float-round-down } \text{prec } f)$   
**by** (*simp add: Float.float-round-down.rep-eq truncate-down-def*)

**lemma** *compute-truncate-up*[code]:  
 $\text{truncate-up } \text{prec} (\text{real-of-float } f) = \text{real-of-float} (\text{float-round-up } \text{prec } f)$   
**by** (*simp add: float-round-up.rep-eq truncate-up-def*)

**lemma** [code]:  $\text{real-divl } p (\text{real-of-float } x) (\text{real-of-float } y) = \text{real-of-float} (\text{float-divl } p \ x \ y)$   
**by** (*simp add: float-divl.rep-eq real-divl-def*)

**lemma** [code]:  $\text{real-divr } p (\text{real-of-float } x) (\text{real-of-float } y) = \text{real-of-float} (\text{float-divr } p \ x \ y)$   
**by** (*simp add: float-divr.rep-eq real-divr-def*)

**lemmas** [code] = *real-of-float-inverse*

**end**

## 16 Examples

**theory** *Ex-Affine-Approximation*

**imports**

*Affine-Code*

*Print*

*Float-Real*

**begin**

**context** *includes floatarith-syntax* **begin**

**definition** *rotate-fas* =

$[\text{Cos} (\text{Rad-of } (\text{Var } 2)) * \text{Var } 0 - \text{Sin} (\text{Rad-of } (\text{Var } 2)) * \text{Var } 1,$

$\text{Sin} (\text{Rad-of } (\text{Var } 2)) * \text{Var } 0 + \text{Cos} (\text{Rad-of } (\text{Var } 2)) * \text{Var } 1]$

**definition** *rotate-slp* = *slp-of-fas rotate-fas*

**definition** *approx-rotate*  $p \ t \ X = \text{approx-slp-outer } p \ 3 \ \text{rotate-slp } X$

```

fun rotate-aform where
  rotate-aform x i = (let r = (((the o (λx. approx-rotate 30 (FloatR 1 (-3)) x))~i)
x) in
  (r ! 0) ×a (r ! 1) ×a (r ! 2))

value [code] rotate-aform (aforms-of-ivls [2, 1, 45] [3, 5, 45]) 70

definition translate-slp = slp-of-fas [Var 0 + Var 2, Var 1 + Var 2]
fun translatei where translatei x i = (((the o (λx. approx-slp-outer 7 3 translate-slp
x))~i) x)

value translatei (aforms-of-ivls [2, 1, 512] [3, 5, 512]) 50

end

hide-const rotate-fas rotate-slp approx-rotate rotate-aform translate-slp translatei

end

```

## 17 Examples on Proving Inequalities

```

theory Ex-Ineqs
  imports
    Affine-Code
    Print
    Float-Real
  begin

  definition plotcolors =
    [[(0, 1, "0x000000"),
      (0, 2, "0xff0000"),
      (1, 2, "0x7f0000"),

      (0, 3, "0x00ff00"),
      (1, 3, "0x00aa00"),
      (2, 3, "0x005500"),

      (1, 4, "0x0000ff"),
      (2, 4, "0x0000c0"),
      (3, 4, "0x00007f"),
      (0, 4, "0x00003f"),

      (0, 5, "0x00ffff"),
      (1, 5, "0x00cccc"),
      (2, 5, "0x009999"),
      (3, 5, "0x006666"),
      (4, 5, "0x003333")],

```

```

[(0, 6, "0xff00ff"),
 (1, 6, "0xd500d5"),
 (2, 6, "0xaa00aa"),
 (3, 6, "0x800080"),
 (4, 6, "0x550055"),
 (5, 6, "0x2a002a")]

```

```

primrec prove-pos::(nat * nat * string) list ⇒ nat ⇒ nat ⇒
  (nat ⇒ real aform list ⇒ real aform option) ⇒ real aform list list ⇒ bool where
  prove-pos prnt 0 p F X = (let - = if prnt ≠ [] then print (STR "# depth limit
  exceeded[↔]") else () in False)
| prove-pos prnt (Suc i) p F XXS =
  (case XXS of [] ⇒ True | (X#XS) ⇒
  let
    R = F p X;
    - = if prnt ≠ [] then print (String.implode ((shows "# " o shows-box-of-aforms-hr
  X) "[↔]")) else ();
    - = fold (λ(a, b, c) -. print (String.implode (shows-segments-of-aform a b X
  c "[↔]"))) prnt ()
  in
    if R ≠ None ∧ 0 < Inf-aform' p (the R)
    then let - = if prnt ≠ [] then print (STR "# Success[↔]") else () in prove-pos
  prnt i p F XS
    else let - = if prnt ≠ [] then print (STR "# Split[↔]") else () in case
  split-aforms-largest-uncond X of (a, b) ⇒
    prove-pos prnt i p F (a#b#XS))

```

**definition** *prove-pos-slp* prnt p fa i xs = (let slp = slp-of-fas [fa] in prove-pos prnt i p (λp xs. case approx-slp-outer p 1 slp xs of None ⇒ None | Some [x] ⇒ Some x | Some - ⇒ None) xs)

**experiment begin**

**unbundle** floatarith-syntax

The examples below are taken from [http://link.springer.com/chapter/10.1007/978-3-642-38088-4\\_26](http://link.springer.com/chapter/10.1007/978-3-642-38088-4_26), “Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations”, Alexey Solovyev, Thomas C. Hales, NASA Formal Methods 2013, LNCS 7871

**definition** *schwefel* =  
 $(5.8806 / 10 \wedge 10) + (\text{Var } 0 - (\text{Var } 1) \wedge_e 2) \wedge_e 2 + (\text{Var } 1 - 1) \wedge_e 2 + (\text{Var } 0 - (\text{Var } 2) \wedge_e 2) \wedge_e 2 + (\text{Var } 2 - 1) \wedge_e 2$

**lemma** *schwefel*:  
 $5.8806 / 10 \wedge 10 + (x0 - (x1)^2)^2 + (x1 - 1)^2 + (x0 - (x2)^2)^2 + (x2 - 1)^2$   
 =

*interpret-floatarith schwefel* [x0, x1, x2]  
**by** (*simp add: schwefel-def*)

**lemma** *prove-pos-slp* [] 30 *schwefel 100000* [*aforms-of-ivls* [-10,-10,-10] [10,10,10]]  
**unfolding** *schwefel-def*  
**by** *eval*

**definition** *delta6* = (Var 0 \* Var 3 \* (-Var 0 + Var 1 + Var 2 - Var 3 + Var 4 + Var 5) +  
Var 1 \* Var 4 \* (Var 0 - Var 1 + Var 2 + Var 3 - Var 4 + Var 5) +  
Var 2 \* Var 5 \* (Var 0 + Var 1 - Var 2 + Var 3 + Var 4 - Var 5) +  
- Var 1 \* Var 2 \* Var 3  
- Var 0 \* Var 2 \* Var 4  
- Var 0 \* Var 1 \* Var 5  
- Var 3 \* Var 4 \* Var 5)

**schematic-goal** *delta6*:

(x0 \* x3 \* (-x0 + x1 + x2 - x3 + x4 + x5) +  
x1 \* x4 \* (x0 - x1 + x2 + x3 - x4 + x5) +  
x2 \* x5 \* (x0 + x1 - x2 + x3 + x4 - x5)  
- x1 \* x2 \* x3  
- x0 \* x2 \* x4  
- x0 \* x1 \* x5  
- x3 \* x4 \* x5) = *interpret-floatarith delta6* [x0, x1, x2, x3, x4, x5]  
**by** (*simp add: delta6-def*)

**lemma** *prove-pos-slp* [] 20 *delta6 10000* [*aforms-of-ivls* (*replicate* 6 4) (*replicate* 6 (FloatR 104045 (-14)))]  
**unfolding** *delta6-def*  
**by** *eval*

**definition** *caprasse* = (3.1801 + - Var 0 \* (Var 2) <sup>e</sup> 3 + 4 \* Var 1 \* (Var 2) <sup>e</sup> 2 \* Var 3 +  
4 \* Var 0 \* Var 2 \* (Var 3) <sup>e</sup> 2 + 2 \* Var 1 \* (Var 3) <sup>e</sup> 3 + 4 \* Var 0 \*  
Var 2 + 4 \* (Var 2) <sup>e</sup> 2 - 10 \* Var 1 \* Var 3 +  
-10 \* (Var 3) <sup>e</sup> 2 + 2)

**schematic-goal** *caprasse*:

(3.1801 + - xs!0 \* (xs!2) <sup>e</sup> 3 + 4 \* xs!1 \* (xs!2)<sup>2</sup> \* xs!3 +  
4 \* xs!0 \* xs!2 \* (xs!3)<sup>2</sup> + 2 \* xs!1 \* (xs!3) <sup>e</sup> 3 + 4 \* xs!0 \* xs!2 + 4 \* (xs!2)<sup>2</sup>  
- 10 \* xs!1 \* xs!3 +  
-10 \* (xs!3)<sup>2</sup> + 2) = *interpret-floatarith caprasse xs*  
**by** (*simp add: caprasse-def*)

**lemma** *prove-pos-slp* [] 20 *caprasse 10000* [*aforms-of-ivls* (*replicate* 4 (1/2)) (*replicate* 4 (1/2))]  
**unfolding** *caprasse-def*  
**by** *eval*

```

definition magnetism =
  0.25001 + (Var 0) e2 + 2 * (Var 1) e2 + 2 * (Var 2) e2 + 2 * (Var 3) e2
+ 2 * (Var 4) e2 + 2 * (Var 5) e2 +
  2 * (Var 6) e2 - Var 0
schematic-goal magnetism:
  0.25001 + (xs!0)2 + 2 * (xs!1)2 + 2 * (xs!2)2 + 2 * (xs!3)2 + 2 * (xs!4)2 +
  2 * (xs!5)2 +
  2 * (xs!6)2 - xs!0 = interpret-floatarith magnetism xs
  by (simp add: magnetism-def)

end

end

```

## 18 Examples: Intersection of Zonotopes with Hyperplanes

```

theory Ex-Inter
  imports
    Intersection
    Affine-Code
    Print
begin

```

### 18.1 Example

```

definition zono1::(real*real*real) aform
  where zono1 = msum-aform 53 (aform-of-ivl ((0,0,0)::real*real*real) ((1,2,0)::real*real*real))
    (0, pdevs-of-list [(5, 10, 20)])

```

```

definition interzono1::(real*real*real) aform
  where interzono1 = the (inter-aform-plane-ortho 53 zono1 (0, 0, 1) 3)

```

10-dimensional zonotope with 50 generators

```

definition random-zono::(real*real*real*real*real*real*real*real*real*real) aform
where
  random-zono =
    (0, pdevs-of-list
      [(5, 9, 27, 12, 23, 3, 9, 10, 18, 2),
       (26, 4, 14, 15, 11, 7, 27, 5, 21, 16),
       (10, 17, 11, 27, 13, 14, 27, 14, 25, 23),
       (7, 6, 5, 30, 14, 10, 2, 1, 18, 25),
       (17, 5, 28, 6, 10, 22, 5, 18, 8, 11),
       (5, 7, 14, 14, 5, 11, 5, 17, 1, 22),
       (3, 6, 11, 20, 28, 13, 12, 10, 2, 23),
       (3, 1, 26, 15, 1, 3, 25, 23, 6, 18),
       (30, 8, 24, 16, 8, 20, 27, 25, 21, 17),

```

(30, 4, 8, 12, 8, 4, 22, 27, 23, 2),  
 (24, 21, 19, 15, 24, 22, 16, 15, 25, 6),  
 (20, 4, 1, 24, 2, 9, 19, 4, 21, 17),  
 (1, 12, 13, 7, 8, 8, 2, 11, 28, 6),  
 (26, 25, 19, 8, 6, 26, 27, 17, 27, 25),  
 (8, 8, 1, 4, 6, 2, 28, 13, 18, 28),  
 (14, 14, 12, 7, 26, 19, 9, 25, 21, 17),  
 (25, 14, 30, 17, 24, 17, 7, 25, 25, 5),  
 (27, 21, 29, 22, 30, 10, 13, 15, 23, 19),  
 (27, 5, 10, 4, 11, 12, 3, 20, 8, 23),  
 (29, 11, 19, 12, 2, 28, 30, 27, 27, 1),  
 (18, 7, 23, 1, 14, 6, 23, 22, 23, 19),  
 (7, 17, 3, 15, 28, 15, 9, 16, 23, 7),  
 (18, 25, 10, 13, 17, 14, 3, 24, 14, 7),  
 (28, 13, 6, 27, 8, 14, 7, 14, 5, 24),  
 (17, 5, 18, 9, 2, 11, 24, 17, 3, 2),  
 (13, 17, 15, 30, 27, 29, 29, 16, 27, 13),  
 (25, 21, 21, 17, 19, 3, 26, 27, 26, 2),  
 (5, 16, 21, 18, 23, 1, 19, 13, 10, 2),  
 (8, 27, 14, 16, 2, 11, 27, 27, 29, 2),  
 (10, 22, 1, 23, 2, 22, 17, 22, 19, 15),  
 (16, 8, 9, 27, 19, 23, 24, 30, 1, 3),  
 (2, 20, 9, 12, 19, 21, 30, 9, 19, 13),  
 (23, 21, 28, 26, 27, 17, 22, 9, 17, 13),  
 (24, 1, 19, 19, 28, 21, 4, 8, 10, 20),  
 (27, 19, 7, 23, 11, 30, 12, 10, 27, 20),  
 (4, 3, 23, 21, 17, 13, 25, 8, 13, 26),  
 (11, 25, 7, 2, 27, 10, 15, 14, 17, 23),  
 (25, 27, 28, 15, 11, 4, 30, 25, 16, 1),  
 (27, 26, 11, 21, 9, 14, 15, 11, 30, 18),  
 (3, 19, 13, 17, 13, 9, 22, 4, 20, 30),  
 (21, 26, 20, 8, 19, 1, 22, 9, 28, 15),  
 (22, 12, 5, 25, 29, 27, 13, 9, 2, 10),  
 (9, 24, 30, 6, 23, 13, 18, 15, 30, 20),  
 (13, 5, 7, 6, 21, 30, 7, 22, 26, 15),  
 (9, 3, 3, 1, 29, 16, 10, 2, 21, 25),  
 (3, 14, 22, 18, 21, 15, 16, 22, 27, 26),  
 (16, 25, 16, 22, 27, 18, 4, 15, 9, 21),  
 (30, 23, 29, 24, 20, 14, 15, 25, 3, 22),  
 (6, 18, 17, 14, 19, 25, 9, 22, 7, 26),  
 (24, 7, 30, 27, 9, 2, 8, 23, 24, 1)])

10-dimensional zonotope with 100 generators

**definition** *random-zono2::(real\*real\*real\*real\*real\*real\*real\*real\*real\*real) aform*  
**where**

*random-zono2 =*  
 (0, *pdevs-of-list*  
 [(17, 28, 12, 10, 18, 3, 14, 27, 21, 22),  
 (7, 17, 16, 26, 25, 4, 12, 20, 18, 28)],

(11, 8, 30, 20, 11, 17, 8, 13, 28, 18),  
 (18, 20, 26, 12, 25, 24, 23, 24, 22, 2),  
 (14, 27, 20, 12, 16, 7, 21, 5, 5, 20),  
 (4, 27, 8, 19, 11, 14, 9, 25, 8, 11),  
 (14, 29, 12, 28, 29, 21, 20, 6, 18, 6),  
 (20, 25, 8, 19, 30, 1, 21, 18, 7, 18),  
 (5, 6, 7, 25, 30, 2, 19, 7, 13, 19),  
 (11, 15, 16, 13, 17, 2, 9, 10, 29, 17),  
 (29, 1, 30, 6, 6, 27, 19, 24, 11, 12),  
 (27, 30, 8, 11, 30, 2, 19, 25, 5, 27),  
 (3, 26, 16, 18, 12, 11, 4, 8, 2, 4),  
 (16, 7, 11, 23, 29, 30, 22, 22, 5, 21),  
 (6, 12, 28, 24, 12, 4, 11, 27, 6, 13),  
 (30, 13, 16, 29, 22, 7, 10, 12, 3, 17),  
 (26, 22, 6, 4, 8, 11, 29, 23, 13, 17),  
 (30, 23, 20, 3, 4, 28, 25, 26, 25, 17),  
 (30, 27, 8, 20, 4, 1, 9, 6, 23, 16),  
 (10, 27, 15, 17, 14, 9, 19, 22, 7, 19),  
 (29, 5, 14, 23, 23, 29, 13, 19, 1, 14),  
 (7, 30, 29, 23, 27, 2, 3, 8, 10, 14),  
 (7, 10, 10, 10, 30, 5, 7, 29, 7, 23),  
 (2, 1, 11, 19, 23, 9, 14, 16, 13, 25),  
 (5, 10, 2, 24, 16, 21, 21, 30, 14, 12),  
 (25, 19, 9, 29, 21, 29, 10, 4, 19, 25),  
 (30, 18, 3, 8, 9, 6, 13, 17, 1, 19),  
 (7, 30, 18, 16, 25, 15, 10, 17, 18, 12),  
 (21, 10, 13, 2, 12, 25, 25, 2, 27, 19),  
 (17, 7, 18, 22, 24, 10, 8, 3, 26, 3),  
 (3, 22, 19, 23, 30, 20, 1, 25, 18, 27),  
 (8, 2, 15, 23, 28, 18, 4, 20, 7, 7),  
 (4, 8, 29, 22, 20, 8, 18, 29, 13, 2),  
 (20, 5, 8, 8, 20, 17, 2, 17, 29, 2),  
 (4, 27, 8, 20, 18, 2, 18, 21, 6, 16),  
 (8, 11, 24, 10, 20, 6, 16, 17, 13, 23),  
 (22, 8, 21, 25, 17, 13, 9, 21, 4, 19),  
 (18, 23, 22, 22, 2, 15, 25, 18, 30, 7),  
 (2, 5, 5, 21, 18, 6, 27, 5, 30, 6),  
 (28, 4, 17, 15, 27, 7, 27, 5, 9, 19),  
 (8, 7, 4, 28, 22, 1, 28, 10, 14, 8),  
 (6, 7, 30, 26, 5, 15, 21, 28, 1, 21),  
 (20, 11, 8, 18, 17, 1, 24, 11, 22, 6),  
 (23, 5, 29, 8, 10, 8, 28, 6, 5, 3),  
 (8, 8, 17, 23, 23, 10, 9, 27, 10, 20),  
 (3, 7, 29, 26, 1, 16, 1, 30, 5, 4),  
 (23, 22, 17, 2, 15, 16, 17, 7, 20, 13),  
 (1, 14, 3, 21, 14, 5, 24, 29, 5, 4),  
 (6, 14, 26, 18, 29, 7, 2, 19, 19, 24),  
 (24, 24, 10, 14, 22, 6, 17, 13, 3, 6),  
 (5, 17, 2, 30, 26, 6, 21, 13, 11, 7),



(11, 20, 15, 29, 20, 2, 23, 6, 28, 9),  
 (27, 10, 3, 16, 21, 22, 8, 5, 19, 14),  
 (21, 25, 23, 24, 7, 3, 30, 8, 21, 19),  
 (10, 9, 17, 15, 14, 2, 5, 19, 28, 9),  
 (1, 4, 3, 1, 22, 27, 15, 26, 1, 9),  
 (8, 19, 18, 12, 26, 18, 1, 5, 19, 16),  
 (6, 30, 11, 8, 22, 1, 24, 10, 30, 5),  
 (10, 11, 12, 14, 24, 27, 22, 8, 11, 27),  
 (8, 29, 17, 19, 20, 17, 4, 9, 3, 1),  
 (17, 15, 1, 17, 22, 30, 1, 22, 3, 23),  
 (1, 11, 15, 8, 6, 22, 4, 24, 18, 3),  
 (23, 21, 24, 2, 17, 14, 14, 7, 18, 27),  
 (30, 3, 25, 17, 25, 3, 5, 8, 4, 24),  
 (4, 29, 30, 7, 14, 27, 25, 11, 18, 19),  
 (2, 26, 15, 13, 16, 8, 7, 11, 21, 23),  
 (9, 22, 28, 29, 18, 9, 22, 25, 26, 20),  
 (21, 15, 29, 18, 24, 29, 20, 17, 2, 29),  
 (12, 17, 11, 9, 4, 6, 2, 4, 22, 25),  
 (17, 9, 9, 19, 3, 8, 6, 22, 12, 15),  
 (28, 19, 25, 28, 1, 15, 8, 7, 6, 4),  
 (17, 17, 22, 7, 1, 21, 25, 23, 22, 14),  
 (19, 1, 7, 3, 11, 9, 7, 24, 2, 4),  
 (17, 27, 18, 29, 8, 2, 17, 17, 13, 30),  
 (8, 14, 14, 11, 26, 20, 28, 25, 13, 17),  
 (10, 17, 7, 26, 24, 4, 10, 17, 2, 15),  
 (21, 9, 29, 7, 13, 10, 13, 17, 2, 2),  
 (16, 10, 18, 27, 26, 26, 3, 30, 14, 1),  
 (9, 15, 11, 9, 2, 11, 3, 13, 29, 20),  
 (18, 9, 22, 25, 15, 5, 21, 2, 13, 20),  
 (9, 22, 15, 11, 24, 27, 22, 12, 16, 6),  
 (4, 6, 20, 5, 25, 20, 3, 21, 26, 30),  
 (24, 7, 19, 19, 27, 26, 3, 9, 13, 13),  
 (27, 22, 8, 27, 13, 24, 23, 1, 26, 28),  
 (12, 29, 7, 6, 25, 17, 22, 10, 6, 24),  
 (2, 25, 30, 13, 10, 11, 20, 8, 10, 2),  
 (28, 14, 11, 23, 28, 26, 2, 28, 28, 24),  
 (8, 3, 24, 9, 10, 19, 11, 7, 5, 3),  
 (25, 11, 27, 7, 4, 18, 14, 17, 3, 8),  
 (2, 2, 20, 6, 26, 28, 7, 22, 2, 3),  
 (29, 15, 23, 30, 23, 30, 1, 13, 12, 3),  
 (18, 2, 4, 21, 23, 16, 17, 15, 9, 17),  
 (28, 22, 12, 16, 8, 20, 14, 8, 2, 10),  
 (28, 6, 18, 9, 4, 17, 11, 5, 19, 16),  
 (27, 15, 27, 2, 4, 21, 21, 9, 10, 13),  
 (5, 23, 13, 9, 28, 19, 5, 5, 14, 27),  
 (7, 15, 2, 12, 9, 6, 12, 23, 25, 25),  
 (7, 17, 17, 11, 20, 5, 13, 27, 27, 6),  
 (7, 30, 14, 22, 16, 16, 11, 30, 29, 8)]

a randomly generated 20-dimensional zonotope\* with 50 generators

**definition** *random-zono3*::

(*real\*real*) *aform*

**where**

*random-zono3* =

(0, *pdevs-of-list*

[(30, 22, 14, 3, 15, 10, 9, 9, 18, 22, 24, 27, 24, 5, 24, 18, 16, 4, 13, 21),  
 (30, 10, 25, 6, 5, 10, 7, 13, 14, 27, 30, 30, 6, 21, 12, 28, 1, 1, 24, 18),  
 (25, 14, 10, 30, 9, 5, 2, 11, 11, 11, 26, 8, 12, 18, 5, 10, 17, 15, 30, 24),  
 (30, 27, 21, 21, 27, 23, 7, 1, 22, 4, 13, 3, 20, 12, 4, 14, 13, 13, 4, 28),  
 (9, 22, 4, 13, 19, 26, 8, 19, 28, 24, 14, 1, 30, 14, 9, 20, 12, 12, 14, 1),  
 (7, 6, 13, 1, 21, 28, 23, 1, 26, 16, 6, 25, 12, 26, 17, 13, 30, 12, 28, 25),  
 (12, 12, 30, 23, 15, 11, 7, 8, 11, 20, 8, 17, 16, 20, 18, 9, 9, 11, 9, 18),  
 (9, 3, 13, 16, 28, 6, 28, 4, 1, 20, 23, 19, 12, 9, 11, 26, 2, 24, 8, 10),  
 (3, 9, 11, 22, 29, 17, 1, 16, 27, 6, 16, 3, 24, 20, 20, 14, 4, 14, 21, 11),  
 (16, 7, 9, 30, 14, 22, 1, 11, 7, 8, 18, 21, 24, 18, 27, 22, 17, 26, 21, 6),  
 (4, 4, 4, 24, 24, 22, 28, 24, 25, 14, 2, 22, 6, 24, 19, 14, 13, 11, 8, 1),  
 (30, 9, 12, 17, 23, 11, 18, 1, 19, 3, 18, 26, 19, 16, 21, 10, 23, 28, 17, 11),  
 (5, 5, 25, 22, 15, 24, 4, 17, 18, 23, 29, 12, 18, 20, 27, 13, 4, 29, 6, 23),  
 (29, 14, 14, 17, 20, 17, 1, 27, 5, 4, 3, 4, 7, 12, 12, 21, 14, 21, 13, 11),  
 (3, 21, 14, 3, 14, 27, 5, 22, 22, 3, 4, 1, 24, 17, 1, 7, 7, 24, 16, 6),  
 (14, 2, 24, 16, 10, 11, 23, 30, 14, 19, 16, 16, 22, 12, 28, 19, 12, 25, 17,

11),

(8, 23, 19, 25, 5, 30, 22, 13, 28, 28, 23, 7, 24, 29, 3, 13, 2, 7, 6, 10),  
 (4, 10, 13, 5, 15, 22, 11, 20, 4, 9, 11, 17, 16, 30, 1, 12, 29, 7, 20, 11),  
 (19, 6, 22, 17, 9, 3, 6, 13, 18, 21, 21, 27, 4, 23, 18, 5, 23, 16, 21, 1),  
 (2, 8, 16, 16, 8, 21, 19, 22, 10, 28, 7, 11, 21, 3, 18, 30, 15, 21, 3, 16),  
 (7, 8, 8, 19, 21, 13, 7, 7, 29, 16, 10, 5, 21, 28, 16, 19, 11, 21, 13, 23),  
 (26, 7, 26, 14, 9, 18, 10, 24, 20, 2, 5, 1, 15, 21, 29, 24, 27, 20, 24, 16),  
 (4, 14, 10, 8, 22, 20, 1, 4, 1, 25, 17, 15, 16, 2, 30, 10, 29, 11, 29, 17),  
 (21, 12, 16, 3, 28, 7, 3, 8, 12, 19, 24, 12, 6, 14, 18, 16, 24, 12, 21, 2),  
 (7, 30, 25, 20, 23, 14, 17, 17, 18, 27, 24, 17, 3, 19, 7, 10, 19, 14, 24, 6),  
 (12, 16, 26, 29, 27, 1, 18, 3, 14, 4, 27, 28, 24, 4, 18, 25, 25, 7, 12, 30),  
 (19, 30, 30, 15, 16, 4, 12, 16, 27, 24, 22, 28, 13, 14, 22, 17, 18, 21, 7,

19),

(9, 9, 23, 5, 1, 23, 9, 26, 23, 13, 19, 14, 29, 27, 23, 25, 2, 13, 18, 11),  
 (12, 8, 20, 14, 14, 23, 24, 11, 8, 6, 25, 27, 28, 3, 4, 15, 1, 22, 19, 22),  
 (19, 23, 28, 13, 2, 5, 17, 1, 17, 19, 30, 7, 6, 29, 7, 12, 11, 20, 30, 23),  
 (27, 10, 21, 19, 24, 17, 10, 22, 22, 26, 2, 25, 8, 1, 5, 9, 22, 18, 28, 6),  
 (9, 22, 9, 13, 20, 10, 6, 23, 7, 10, 29, 5, 28, 30, 22, 23, 8, 10, 14, 11),  
 (14, 16, 20, 4, 25, 1, 10, 20, 13, 29, 17, 14, 21, 30, 21, 16, 10, 19, 6, 16),  
 (25, 3, 6, 20, 18, 23, 3, 12, 14, 9, 2, 2, 30, 19, 12, 29, 23, 20, 29, 22),  
 (20, 15, 11, 23, 5, 17, 13, 2, 4, 20, 16, 7, 7, 24, 7, 10, 13, 22, 9, 15),  
 (8, 12, 30, 22, 11, 26, 25, 16, 27, 2, 9, 15, 15, 13, 30, 21, 4, 3, 1, 5),  
 (23, 26, 23, 29, 26, 24, 8, 15, 22, 5, 26, 6, 2, 3, 17, 5, 14, 25, 28, 10),  
 (20, 28, 25, 20, 9, 22, 1, 5, 24, 8, 10, 19, 3, 26, 21, 1, 13, 15, 3, 3),  
 (9, 24, 1, 5, 22, 11, 11, 22, 25, 25, 16, 25, 24, 28, 15, 26, 22, 1, 23, 9),  
 (13, 1, 11, 16, 6, 12, 11, 8, 29, 21, 23, 21, 21, 20, 5, 26, 2, 23, 2, 16),  
 (12, 13, 5, 24, 25, 19, 26, 4, 17, 5, 18, 6, 2, 29, 21, 3, 10, 20, 7, 5),

```

(26, 10, 13, 17, 29, 22, 3, 3, 28, 11, 5, 8, 11, 11, 17, 27, 19, 17, 23, 8),
(2, 4, 11, 17, 18, 23, 14, 22, 4, 29, 2, 29, 25, 3, 4, 13, 2, 14, 5, 15),
(12, 6, 16, 4, 25, 22, 29, 21, 2, 27, 17, 4, 11, 22, 2, 2, 5, 9, 28, 8),
(3, 26, 17, 3, 29, 17, 16, 24, 10, 9, 16, 4, 23, 14, 10, 12, 16, 28, 28, 28),
(7, 15, 28, 6, 25, 24, 11, 26, 22, 3, 28, 17, 10, 17, 19, 12, 20, 18, 29, 23),
(24, 7, 7, 26, 17, 23, 19, 29, 1, 28, 11, 30, 23, 25, 30, 2, 6, 21, 1, 16),
(6, 27, 22, 25, 9, 1, 16, 2, 12, 30, 23, 19, 12, 29, 20, 16, 16, 16, 6, 21),
(25, 12, 5, 28, 19, 9, 25, 12, 10, 27, 10, 26, 27, 15, 2, 4, 23, 12, 20, 27)]

```

**fun** *random-inter1* **where**

```

random-inter1 () =
  the (inter-aform-plane-ortho 53 random-zono (1, 15, 26, 8, 15, 23, 5, 14, 8,
8) 12)

```

**fun** *random-inter2* **where**

```

random-inter2 () =
  the (inter-aform-plane-ortho 53 random-zono2 (13, 23, 22, 30, 27, 19, 17, 11,
24, 29) 12)

```

**fun** *random-inter3* **where**

```

random-inter3 () =
  the (inter-aform-plane-ortho 53 random-zono3
(7, 10, 24, 12, 6, 14, 10, 14, 23, 13, 25, 27, 20, 2, 1, 9, 4, 17, 28, 19)
12)

```

**ML** <

```

val ri1 = @{code random-inter1}
val ri2 = @{code random-inter2}
val ri3 = @{code random-inter3}
>

```

Timings

**ML** <

```

fun iter f 0 = f ()
| iter f i = let val - = f () in iter f (i - 1) end
>

```

**ML** <*iter ri1 100*> — 0.7 s

**ML** <*iter ri2 100*> — 1.3 s

**ML** <*iter ri3 100*> — 1.3 s

**end**

**theory** *Affine-Arithmetic*

**imports**

```

Affine-Code
Intersection
Straight-Line-Program
Ex-Affine-Approximation
Ex-Ineqs
Ex-Inter

```

**begin**

**end**

## References

- [1] L. H. De Figueiredo and J. Stolfi. Affine arithmetic: concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004.
- [2] A. Girard. Reachability of uncertain linear systems using zonotopes. In *Hybrid systems: computation and control*, pages 291–305. Springer, 2005.
- [3] F. Immler. A verified algorithm for geometric zonotope/hyperplane intersection. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP '15*, pages 129–136, New York, NY, USA, 2015. ACM.