Affine Arithmetic

Fabian Immler

September 23, 2021

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

theory *Affine-Arithmetic-Auxiliarities*
imports *HOL−Analysis.Multivariate-Analysis*
begin

0.1 *sum-list*

lemma *sum-list-nth-eqI*:
fixes *xs ys*::'*a*::*monoid-add list*
shows
length xs = *length ys* ⟹ (⋀*x y*. (*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *x* = *y*) ⟹
sum-list xs = *sum-list ys*
⟨*proof*⟩

lemma *fst-sum-list*: *fst* (*sum-list xs*) = *sum-list* (*map fst xs*)
⟨*proof*⟩

lemma *snd-sum-list*: *snd* (*sum-list xs*) = *sum-list* (*map snd xs*)
⟨*proof*⟩

lemma *take-greater-eqI*: *take c xs* = *take c ys* ⟹ *c* ≥ *a* ⟹ *take a xs* = *take a ys*
⟨*proof*⟩

lemma *take-max-eqD*:
take (*max a b*) *xs* = *take* (*max a b*) *ys* ⟹ *take a xs* = *take a ys* ∧ *take b xs* = *take b ys*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *deg-of-monoI*: *x* ≤ *y* ⟹ *deg-of x* ≤ *deg-of y*
⟨*proof*⟩

lemma *rad-of-monoI*: *x* ≤ *y* ⟹ *rad-of x* ≤ *rad-of y*
⟨*proof*⟩

lemma *deg-of-strict-monoI*: *x* < *y* ⟹ *deg-of x* < *deg-of y*
⟨*proof*⟩

lemma *rad-of-strict-monoI*: *x* < *y* ⟹ *rad-of x* < *rad-of y*
⟨*proof*⟩

lemma *deg-of-mono*[*simp*]: *deg-of* $x \leq$ *deg-of* $y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *rad-of-mono*[*simp*]: *rad-of* $x \leq$ *rad-of* $y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *deg-of-strict-mono*[*simp*]: *deg-of* $x <$ *deg-of* $y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *rad-of-strict-mono*[*simp*]: *rad-of* $x <$ *rad-of* $y \longleftrightarrow x < y$
⟨*proof*⟩

lemma *rad-of-lt-iff*: *rad-of* $d < r \longleftrightarrow d <$ *deg-of* r
and *rad-of-gt-iff*: *rad-of* $d > r \longleftrightarrow d >$ *deg-of* r
and *rad-of-le-iff*: *rad-of* $d \leq r \longleftrightarrow d \leq$ *deg-of* r
and *rad-of-ge-iff*: *rad-of* $d \geq r \longleftrightarrow d \geq$ *deg-of* r
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma *eucl-truncate-down-zero*[*simp*]: *eucl-truncate-down p 0 = 0*
⟨*proof*⟩

lemma *eucl-truncate-up-zero*[*simp*]: *eucl-truncate-up p 0 = 0*
⟨*proof*⟩

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*⟩

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*⟩

end

lemma *eucl-truncate-down-Basis*[*simp*]:
i ∈ *Basis* ⟹ *eucl-truncate-down e x* · *i* = *truncate-down e* (*x* · *i*)
⟨*proof*⟩

lemma *eucl-truncate-down-correct*:
dist (*x*::*'a*::*executable-euclidean-space*) (*eucl-down e x*) ∈
{*0*..*sqrt* (*DIM*(*'a*)) ∗ *2 powr of-int* (− *e*)}
⟨*proof*⟩

lemma *eucl-down*: *eucl-down e* (*x*::*'a*::*executable-euclidean-space*) ≤ *x*
⟨*proof*⟩

lemma *eucl-truncate-down*: *eucl-truncate-down e* (*x*::*'a*::*executable-euclidean-space*)
≤ *x*
⟨*proof*⟩

lemma *eucl-truncate-down-le*:
x ≤ *y* ⟹ *eucl-truncate-down w x* ≤ (*y*::*'a*::*executable-euclidean-space*)
⟨*proof*⟩

lemma *eucl-truncate-up-Basis*[*simp*]: *i* ∈ *Basis* ⟹ *eucl-truncate-up e x* · *i* =
truncate-up e (*x* · *i*)
⟨*proof*⟩

lemma *eucl-truncate-up*: *x* ≤ *eucl-truncate-up e* (*x*::*'a*::*executable-euclidean-space*)
⟨*proof*⟩

lemma *eucl-truncate-up-le*: *x* ≤ *y* ⟹ *x* ≤ *eucl-truncate-up e* (*y*::*'a*::*executable-euclidean-space*)
⟨*proof*⟩

lemma *eucl-truncate-down-mono*:
fixes *x*::*'a*::*executable-euclidean-space*
shows *x* ≤ *y* ⟹ *eucl-truncate-down p x* ≤ *eucl-truncate-down p y*
⟨*proof*⟩

lemma *eucl-truncate-up-mono*:
fixes *x*::*'a*::*executable-euclidean-space*
shows *x* ≤ *y* ⟹ *eucl-truncate-up p x* ≤ *eucl-truncate-up p y*
⟨*proof*⟩

lemma *infnorm*[*code*]:
fixes *x*::*'a*::*executable-euclidean-space*
shows *infnorm x* = *fold max* (*map* (λ*i*. *abs* (*x* · *i*)) *Basis-list*) *0*
⟨*proof*⟩

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* ⇒ *real*]]
declare [[*code abort*: *Sup*::*real set* ⇒ *real*]]
declare [[*code abort*: *Inf*::(*'a*::*Inf* ∗ *'b*::*Inf*) *set* ⇒ *'a* ∗ *'b*]]
declare [[*code abort*: *Sup*::(*'a*::*Sup* ∗ *'b*::*Sup*) *set* ⇒ *'a* ∗ *'b*]]

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

1.3 Representation as list

lemma *nth-eq-iff-index*:
$distinct\ xs \Longrightarrow n < length\ xs \Longrightarrow xs\ !\ n = i \longleftrightarrow n = index\ xs\ i$
⟨*proof*⟩

lemma *in-Basis-index-Basis-list*: $i \in Basis \Longrightarrow i = Basis\text{-}list\ !\ index\ Basis\text{-}list\ i$
⟨*proof*⟩

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

lemma *sum-Basis-sum-nth-Basis-list*:
$(\sum i{\in}Basis.\ f\ i) = (\sum i{<}DIM('a::executable\text{-}euclidean\text{-}space).\ f\ ((Basis\text{-}list::'a\ list)\ !\ i))$
⟨*proof*⟩

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

lemma *eucl-of-list-nth*:
assumes *length xs* $= DIM('a)$
shows *eucl-of-list xs* $= (\sum i{<}DIM('a::executable\text{-}euclidean\text{-}space).\ (xs\ !\ i) *_R ((Basis\text{-}list::'a\ list)\ !\ i))$
⟨*proof*⟩

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

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

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

lemma *index-Basis-list-nth*[*simp*]:
$i < DIM('a::executable\text{-}euclidean\text{-}space) \Longrightarrow index\ Basis\text{-}list\ ((Basis\text{-}list::'a\ list)$

! i) = i
⟨*proof*⟩

lemma *list-of-eucl-eucl-of-list*[*simp*]:
length xs = *DIM*('*a*::*executable-euclidean-space*) ⟹ *list-of-eucl* (*eucl-of-list xs*::'*a*)
= *xs*
⟨*proof*⟩

lemma *eucl-of-list-list-of-eucl*[*simp*]:
eucl-of-list (*list-of-eucl x*) = *x*
⟨*proof*⟩

lemma *length-list-of-eucl*[*simp*]: *length* (*list-of-eucl* (*x*::'*a*::*executable-euclidean-space*))
= *DIM*('*a*)
⟨*proof*⟩

lemma *list-of-eucl-nth*[*simp*]: *n* < *DIM*('*a*::*executable-euclidean-space*) ⟹ *list-of-eucl*
x ! *n* = *x* ∙ (*Basis-list* ! *n*::'*a*)
⟨*proof*⟩

lemma *nth-ge-len*: *n* ≥ *length xs* ⟹ *xs* ! *n* = [] ! (*n* − *length xs*)
⟨*proof*⟩

lemma *list-of-eucl-nth-if*: *list-of-eucl x* ! *n* = (*if n* < *DIM*('*a*::*executable-euclidean-space*)
then x ∙ (*Basis-list* ! *n*::'*a*) *else* [] ! (*n* − *DIM*('*a*)))
⟨*proof*⟩

lemma *list-of-eucl-eq-iff*:
list-of-eucl (*x*::'*a*::*executable-euclidean-space*) = *list-of-eucl* (*y*::'*b*::*executable-euclidean-space*)
⟷
(*DIM*('*a*) = *DIM*('*b*) ∧ (∀ *i* < *DIM*('*b*). *x* ∙ *Basis-list* ! *i* = *y* ∙ *Basis-list* ! *i*))
⟨*proof*⟩

lemma *eucl-le-Basis-list-iff*:
(*x*::'*a*::*executable-euclidean-space*) ≤ *y* ⟷
(∀ *i*<*DIM*('*a*). *x* ∙ *Basis-list* ! *i* ≤ *y* ∙ *Basis-list* ! *i*)
⟨*proof*⟩

lemma *eucl-of-list-inj*: *length xs* = *DIM*('*a*::*executable-euclidean-space*) ⟹ *length*
ys = *DIM*('*a*) ⟹
(*eucl-of-list xs*::'*a*) = *eucl-of-list* (*ys*) ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *eucl-of-list-map-plus*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
shows (*eucl-of-list* (*map* (λ*x*. *f x* + *g x*) *xs*)::'*a*) =
eucl-of-list (*map f xs*) + *eucl-of-list* (*map g xs*)
⟨*proof*⟩

lemma *eucl-of-list-map-uminus*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
shows (*eucl-of-list* (*map* (λ*x*. − *f x*) *xs*)::'*a*) = − *eucl-of-list* (*map f xs*)
⟨*proof*⟩

lemma *eucl-of-list-map-mult-left*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
shows (*eucl-of-list* (*map* (λ*x*. *r* ∗ *f x*) *xs*)::'*a*) = *r* ∗$_R$ *eucl-of-list* (*map f xs*)
⟨*proof*⟩

lemma *eucl-of-list-map-mult-right*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
shows (*eucl-of-list* (*map* (λ*x*. *f x* ∗ *r*) *xs*)::'*a*) = *r* ∗$_R$ *eucl-of-list* (*map f xs*)
⟨*proof*⟩

lemma *eucl-of-list-map-divide-right*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
shows (*eucl-of-list* (*map* (λ*x*. *f x* / *r*) *xs*)::'*a*) = *eucl-of-list* (*map f xs*) /$_R$ *r*
⟨*proof*⟩

lemma *eucl-of-list-map-const*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
shows (*eucl-of-list* (*map* (λ*x*. *c*) *xs*)::'*a*) = *c* ∗$_R$ *One*
⟨*proof*⟩

lemma *replicate-eq-list-of-eucl-zero*: *replicate DIM*('*a*::*executable-euclidean-space*)
0 = *list-of-eucl* (*0*::'*a*)
⟨*proof*⟩

lemma *eucl-of-list-append-zeroes*[*simp*]: *eucl-of-list* (*xs* @ *replicate n 0*) = *eucl-of-list xs*
⟨*proof*⟩

lemma *Basis-prodD*:
assumes (*i*, *j*) ∈ *Basis*
shows *i* ∈ *Basis* ∧ *j* = *0* ∨ *i* = *0* ∧ *j* ∈ *Basis*
⟨*proof*⟩

lemma *eucl-of-list-take-DIM*[*simp*]:
assumes *d* = *DIM*('*b*::*executable-euclidean-space*)
shows (*eucl-of-list* (*take d xs*)::'*b*) = (*eucl-of-list xs*)
⟨*proof*⟩

lemma *eucl-of-list-eqI*:
assumes *take DIM*('*a*) (*xs* @ *replicate* (*DIM*('*a*) − *length xs*) *0*) =
take DIM('*a*) (*ys* @ *replicate* (*DIM*('*a*) − *length ys*) *0*)
shows *eucl-of-list xs* = (*eucl-of-list ys*::'*a*::*executable-euclidean-space*)
⟨*proof*⟩

lemma *eucl-of-list-replicate-zero*[*simp*]: *eucl-of-list* (*replicate E 0*) = *0*
⟨*proof*⟩

lemma *eucl-of-list-Nil*[*simp*]: *eucl-of-list* [] = *0*
⟨*proof*⟩

lemma *fst-eucl-of-list-prod*:
shows *fst* (*eucl-of-list xs*::*'b*::*executable-euclidean-space* × -) = (*eucl-of-list* (*take DIM*(*'b*) *xs*)::*'b*)
⟨*proof*⟩

lemma *index-zip-replicate1*[*simp*]: *index* (*zip* (*replicate d a*) *bs*) (*a*, *b*) = *index bs b*
if *d* = *length bs*
⟨*proof*⟩

lemma *index-zip-replicate2*[*simp*]: *index* (*zip as* (*replicate d b*)) (*a*, *b*) = *index as a*
if *d* = *length as*
⟨*proof*⟩

lemma *index-Basis-list-prod*[*simp*]:
fixes *a*::*'a*::*executable-euclidean-space* **and** *b*::*'b*::*executable-euclidean-space*
shows *a* ∈ *Basis* ⟹ *index Basis-list* (*a*, *0*::*'b*) = *index Basis-list a*
b ∈ *Basis* ⟹ *index Basis-list* (*0*::*'a*, *b*) = *DIM*(*'a*) + *index Basis-list b*
⟨*proof*⟩

lemma *eucl-of-list-eq-takeI*:
assumes (*eucl-of-list* (*take DIM*(*'a*::*executable-euclidean-space*) *xs*)::*'a*) = *x*
shows *eucl-of-list xs* = *x*
⟨*proof*⟩

lemma *eucl-of-list-inner-le*:
fixes *i*::*'a*::*executable-euclidean-space*
assumes *i*: *i* ∈ *Basis*
assumes *l*: *length xs* ≥ *DIM*(*'a*)
shows *eucl-of-list xs* • *i* = *xs* ! (*index Basis-list i*)
⟨*proof*⟩

lemma *eucl-of-list-prod-if*:
assumes *length xs* = *DIM*(*'a*::*executable-euclidean-space*) + *DIM*(*'b*::*executable-euclidean-space*)
shows *eucl-of-list xs* =
(*eucl-of-list* (*take DIM*(*'a*) *xs*)::*'a*, *eucl-of-list* (*drop DIM*(*'a*) *xs*)::*'b*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *eucl-of-list-real*[*simp*]: *eucl-of-list* [*x*] = (*x*::*real*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *list-allI*: (⋀*x*. *x* ∈ *set xs* ⟹ *P x*) ⟹ *list-all P xs*
⟨*proof*⟩

lemma
concat-map-nthI:
assumes ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set* (*f x*) ⟹ *P y*
assumes *j* < *length* (*concat* (*map f xs*))
shows *P* (*concat* (*map f xs*) ! *j*)
⟨*proof*⟩

lemma *map-nth-append1*:
assumes *length xs* = *d*
shows *map* ((!) (*xs* @ *ys*)) [*0*..<*d*] = *xs*
⟨*proof*⟩

lemma *map-nth-append2*:
assumes *length ys* = *d*
shows *map* ((!) (*xs* @ *ys*)) [*length xs*..<*length xs* + *d*] = *ys*
⟨*proof*⟩

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

lemma *map2-nth* [*simp*]: *map2 f xs ys* ! *n* = *f* (*xs* ! *n*) (*ys* ! *n*)
if *n* < *length xs n* < *length ys*
⟨*proof*⟩

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

lemma *list-of-eucl-inj*:
list-of-eucl z = list-of-eucl y ⟹ y = z
⟨*proof*⟩

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

lemma *Basis-list-nth-nonzero*:
i < length (*Basis-list*::*'a*::*executable-euclidean-space list*) ⟹ (*Basis-list*::*'a list*)
! *i* ≠ *0*
⟨*proof*⟩

lemma *nth-Basis-list-prod*:
i < DIM(*'a*) + *DIM*(*'b*) ⟹ (*Basis-list*::(*'a*::*executable-euclidean-space* × *'b*::*executable-euclidean-space*)
list) ! *i* =
(*if i < DIM*(*'a*) *then* (*Basis-list* ! *i*, *0*) *else* (*0*, *Basis-list* ! (*i* − *DIM*(*'a*))))
⟨*proof*⟩

lemma *eucl-of-list-if*:
assumes [*simp*]: *length xs = DIM*(*'a*::*executable-euclidean-space*) *distinct xs*
shows *eucl-of-list* (*map* (λ*xa*. *if xa = x then 1 else 0*) (*xs*::*nat list*)) =
(*if x* ∈ *set xs then Basis-list* ! *index xs x else 0*::*'a*)
⟨*proof*⟩

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

lemma *sum-list-Basis-list*[*simp*]: *sum-list* (*map f Basis-list*) = (∑ *b*∈*Basis*. *f b*)
⟨*proof*⟩

lemma *hd-Basis-list*[*simp*]: *hd Basis-list* ∈ *Basis*
⟨*proof*⟩

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

lemma *eucl-of-list-inner-eq*: (*eucl-of-list xs*::*'a*) · *eucl-of-list ys = inner-lv-rel xs ys*
if *length xs = DIM*(*'a*::*executable-euclidean-space*) *length ys = DIM*(*'a*)
⟨*proof*⟩

lemma *euclidean-vec-componentwise*:
(∑ (*xa*::*'a*::*euclidean-space*^*'b*::*finite*)∈*Basis*. *f xa*) = (∑ *a*∈*Basis*. (∑ *b*::*'b*∈*UNIV*.
f (*axis b a*)))
⟨*proof*⟩

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

$f\ (x\ \$\ i \cdot j) *_R j = (\sum xa \in UNIV.\ f\ (x\ \$\ xa \cdot j) *_R axis\ xa\ j)\ \$\ i$
⟨*proof*⟩

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

definition *Basis-list-vec* :: ('*a*, '*b*) *vec list* **where**
Basis-list-vec = *concat* (*map* (λ*n*. *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*⟩
end

lemma *concat-same-lengths-nth*:
assumes ⋀*xs*. *xs* ∈ *set XS* ⟹ *length xs* = *N*
assumes *i* < *length XS* ∗ *N N* > *0*
shows *concat XS* ! *i* = *XS* ! (*i div N*) ! (*i mod N*)
⟨*proof*⟩

lemma *concat-map-map-index*:
shows *concat* (*map* (λ*n*. *map* (*f n*) *xs*) *ys*) =
map (λ*i*. *f* (*ys* ! (*i div length xs*)) (*xs* ! (*i mod length xs*))) [*0*..<*length xs* ∗ *length ys*]
⟨*proof*⟩

lemma
sum-list-zip-map:
assumes *distinct xs*
shows (∑ (*x*, *y*)←*zip xs* (*map g xs*). *f x y*) = (∑ *x*∈*set xs*. *f x* (*g x*))
⟨*proof*⟩

lemma
sum-list-zip-map-of:
assumes *distinct bs*
assumes *length xs* = *length bs*
shows (∑ (*x*, *y*)←*zip xs bs*. *f x y*) = (∑ *x*∈*set bs*. *f* (*the* (*map-of* (*zip bs xs*) *x*)) *x*)
⟨*proof*⟩

lemma *vec-nth-matrix*:
vec-nth (*vec-nth* (*matrix y*) *i*) *j* = *vec-nth* (*y* (*axis j 1*)) *i*
⟨*proof*⟩

lemma *matrix-eqI*:
assumes ⋀*x*. *x* ∈ *Basis* ⟹ *A* ∗*v x* = *B* ∗*v x*
shows (*A*::*real*^′*n*^′*n*) = *B*
⟨*proof*⟩

lemma *matrix-columnI*:
assumes ⋀*i*. *column i A* = *column i B*
shows (*A*::*real*^′*n*^′*n*) = *B*
⟨*proof*⟩

lemma
vec-nth-Basis:
fixes *x*::*real*^′*n*
shows *x* ∈ *Basis* ⟹ *vec-nth x i* = (*if x* = *axis i 1 then 1 else 0*)
⟨*proof*⟩

lemma *vec-nth-eucl-of-list-eq*: *length M* = *CARD*(′*n*) ⟹
vec-nth (*eucl-of-list M*::*real*^′*n*::*enum*) *i* = *M* ! *index Basis-list* (*axis i* (*1*::*real*))
⟨*proof*⟩

lemma *index-Basis-list-axis1*: *index Basis-list* (*axis i* (*1*::*real*)) = *index enum-class.enum i*
⟨*proof*⟩

lemma *vec-nth-eq-list-of-eucl1*:
(*vec-nth* (*M*::*real*^′*n*::*enum*) *i*) = *list-of-eucl M* ! (*index enum-class.enum i*)
⟨*proof*⟩

lemma *enum-3*[*simp*]: (*enum-class.enum*::*3 list*) = [*0*, *1*, *2*]
⟨*proof*⟩

lemma *three-eq-zero*: (*3*::*3*) = *0* ⟨*proof*⟩

lemma *forall-3*′: (∀ *i*::*3*. *P i*) ⟷ *P 0* ∧ *P 1* ∧ *P 2*
⟨*proof*⟩

lemma *euclidean-eq-list-of-euclI*: *x* = *y* **if** *list-of-eucl x* = *list-of-eucl y*
⟨*proof*⟩

lemma *axis-one-neq-zero*[*simp*]: *axis xa* (*1*::′*a*::*zero-neq-one*) ≠ *0*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *Basis-list-R3*: *Basis-list* = [(*1*,*0*,*0*), (*0*, *1*, *0*), (*0*, *0*, *1*)::*R3*]
⟨*proof*⟩

lemma *Basis-list-vec3*: *Basis-list* = [*axis 0 1*::*real^3*, *axis 1 1*, *axis 2 1*]
⟨*proof*⟩

lemma *eucl-of-list3*[*simp*]: *eucl-of-list* [*a*, *b*, *c*] = (*a*, *b*, *c*)
⟨*proof*⟩

1.4 Bounded Linear Functions

1.5 bounded linear functions

locale *blinfun-syntax*
begin
no-notation *vec-nth* (**infixl** $ *90*)
notation *blinfun-apply* (**infixl** $ *999*)
end

lemma *bounded-linear-via-derivative*:
fixes *f*::*'a*::*real-normed-vector* ⇒ *'b*::*euclidean-space* ⇒$_L$ *'c*::*real-normed-vector*
— TODO: generalize?
assumes ⋀*i*. ((λ*x*. *blinfun-apply* (*f x*) *i*) *has-derivative* (λ*x*. *f' y x i*)) (*at y*)
shows *bounded-linear* (*f' y x*)
⟨*proof*⟩

definition *blinfun-scaleR*::(*'a*::*real-normed-vector* ⇒$_L$ *real*) ⇒ *'b*::*real-normed-vector*
⇒ (*'a* ⇒$_L$ *'b*)
where *blinfun-scaleR a b* = *blinfun-scaleR-left b* o$_L$ *a*

lemma *blinfun-scaleR-transfer*[*transfer-rule*]:
rel-fun (*pcr-blinfun* (=) (=)) (*rel-fun* (=) (*pcr-blinfun* (=) (=)))
(λ*a b c*. *a c* *$_R$ *b*) *blinfun-scaleR*
⟨*proof*⟩

lemma *blinfun-scaleR-rep-eq*[*simp*]:
blinfun-scaleR a b c = *a c* *$_R$ *b*
⟨*proof*⟩

lemma *bounded-linear-blinfun-scaleR*: *bounded-linear* (*blinfun-scaleR a*)
⟨*proof*⟩

lemma *blinfun-scaleR-has-derivative*[*derivative-intros*]:
assumes (*f has-derivative f'*) (*at x within s*)
shows ((λ*x*. *blinfun-scaleR a* (*f x*)) *has-derivative* (λ*x*. *blinfun-scaleR a* (*f' x*))) (*at x within s*)
⟨*proof*⟩

lemma *blinfun-componentwise*:
fixes *f*::'*a*::*real-normed-vector* ⇒ '*b*::*euclidean-space* ⇒$_L$ '*c*::*real-normed-vector*
shows *f* = (λ*x*. ∑ *i*∈*Basis*. *blinfun-scaleR* (*blinfun-inner-left i*) (*f x i*))
⟨*proof*⟩

lemma
blinfun-has-derivative-componentwiseI:
fixes *f*::'*a*::*real-normed-vector* ⇒ '*b*::*euclidean-space* ⇒$_L$ '*c*::*real-normed-vector*
assumes ⋀*i*. *i* ∈ *Basis* ⟹ ((λ*x*. *f x i*) *has-derivative blinfun-apply* (*f' i*)) (*at x*)
shows (*f has-derivative* (λ*x*. ∑ *i*∈*Basis*. *blinfun-scaleR* (*blinfun-inner-left i*) (*f' i x*))) (*at x*)
⟨*proof*⟩

lemma
has-derivative-BlinfunI:
fixes *f*::'*a*::*real-normed-vector* ⇒ '*b*::*euclidean-space* ⇒$_L$ '*c*::*real-normed-vector*
assumes ⋀*i*. ((λ*x*. *f x i*) *has-derivative* (λ*x*. *f' y x i*)) (*at y*)
shows (*f has-derivative* (λ*x*. *Blinfun* (*f' y x*))) (*at y*)
⟨*proof*⟩

lemma
has-derivative-Blinfun:
assumes (*f has-derivative f'*) *F*
shows (*f has-derivative Blinfun f'*) *F*
⟨*proof*⟩

lift-definition *flip-blinfun*::
('*a*::*real-normed-vector* ⇒$_L$ '*b*::*real-normed-vector* ⇒$_L$ '*c*::*real-normed-vector*) ⇒ '*b* ⇒$_L$ '*a* ⇒$_L$ '*c* **is**
λ*f x y*. *f y x*
⟨*proof*⟩

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

lemma *le-norm-blinfun*:
shows *norm* (*blinfun-apply f x*) / *norm x* ≤ *norm f*
⟨*proof*⟩

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

lemma *bounded-linear-flip-blinfun*[*bounded-linear*]: *bounded-linear flip-blinfun*
⟨*proof*⟩

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

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

lift-definition *blinfun-of-vmatrix*::(*real*^'*m*^'*n*) ⇒ ((*real*^('*m*::*finite*)) ⇒$_L$ (*real*^('*n*::*finite*)))
is
matrix-vector-mult:: ((*real*, '*m*) *vec*, '*n*) *vec* ⇒ ((*real*, '*m*) *vec* ⇒ (*real*, '*n*) *vec*)
⟨*proof*⟩

lemma *matrix-blinfun-of-vmatrix*[*simp*]: *matrix* (*blinfun-of-vmatrix M*) = *M*
⟨*proof*⟩

end

lemma *blinfun-apply-componentwise*:
B = (∑ *i*∈*Basis*. *blinfun-scaleR* (*blinfun-inner-left i*) (*blinfun-apply B i*))
⟨*proof*⟩

lemma *blinfun-apply-eq-sum*:
assumes [*simp*]: *length v* = *CARD*('*n*)
shows *blinfun-apply* (*B*::(*real*^'*n*::*enum*)⇒$_L$(*real*^'*m*::*enum*)) (*eucl-of-list v*) =
(∑ *i*<*CARD*('*m*). ∑ *j*<*CARD*('*n*). ((*B* (*Basis-list* ! *j*) ∙ *Basis-list* ! *i*) ∗ *v* ! *j*)
∗$_R$ (*Basis-list* ! *i*))
⟨*proof*⟩

lemma *in-square-lemma*[*intro*, *simp*]: *x* ∗ *C* + *y* < *D* ∗ *C* **if** *x* < *D y* < *C* **for**
x::*nat*
⟨*proof*⟩

lemma *less-square-imp-div-less*[*intro*, *simp*]: *i* < *E* ∗ *D* ⟹ *i div E* < *D* **for** *i*::*nat*
⟨*proof*⟩

lemma *in-square-lemma'*[*intro*, *simp*]: *i* < *L* ⟹ *n* < *N* ⟹ *i* ∗ *N* + *n* < *N* ∗ *L*
for *i n*::*nat*
⟨*proof*⟩

lemma
distinct-nth-eq-iff:
distinct xs ⟹ *x* < *length xs* ⟹ *y* < *length xs* ⟹ *xs* ! *x* = *xs* ! *y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *index-Basis-list-axis2*:
index Basis-list (*axis* (*j*::'*j*::*enum*) (*axis* (*i*::'*i*::*enum*) (*1*::*real*))) =
(*index enum-class.enum j*) ∗ *CARD*('*i*) + *index enum-class.enum i*

⟨*proof*⟩

lemma
vec-nth-Basis2:
fixes *x*::*real*^'*n*^'*m*
shows *x* ∈ *Basis* ⟹ *vec-nth* (*vec-nth x i*) *j* = ((*if x* = *axis i* (*axis j 1*) *then 1 else 0*))
⟨*proof*⟩

lemma *vec-nth-eucl-of-list-eq2*: *length M* = *CARD*('*n*) ∗ *CARD*('*m*) ⟹
vec-nth (*vec-nth* (*eucl-of-list M*::*real*^'*n*::*enum*^'*m*::*enum*) *i*) *j* = *M* ! *index Basis-list* (*axis i* (*axis j* (*1*::*real*)))
⟨*proof*⟩

lemma *vec-nth-eq-list-of-eucl2*:
vec-nth (*vec-nth* (*M*::*real*^'*n*::*enum*^'*m*::*enum*) *i*) *j* =
list-of-eucl M ! (*index enum-class.enum i* ∗ *CARD*('*n*) + *index enum-class.enum j*)
⟨*proof*⟩

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*^'*n*::*enum*^'*m*::*enum*) ∗*v eucl-of-list v* =
(∑ *i*<*CARD*('*m*)).
(∑ *j*<*CARD*('*n*)). *M* ! (*i* ∗ *CARD*('*n*) + *j*) ∗ *v* ! *j*) ∗$_R$ *Basis-list* ! *i*)
⟨*proof*⟩

lemma *index-enum-less*[*intro*, *simp*]: *index enum-class.enum* (*i*::'*n*::*enum*) < *CARD*('*n*)
⟨*proof*⟩

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

lemma *sum-index-enum-eq*:
(∑ (*k*::'*n*::*enum*)∈*UNIV*. *f* (*index enum-class.enum k*)) = (∑ *i*<*CARD*('*n*)). *f i*)
⟨*proof*⟩

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*::*ordered-ab-group-add list*
shows
length xs = *length ys* ⟹ (⋀*x y*. (*x*, *y*) ∈ *set* (*zip xs ys*) ⟹ *x* ≤ *y*) ⟹
sum-list xs ≤ *sum-list ys*
⟨*proof*⟩

lemma
fixes *xs*::*'a*::*ordered-comm-monoid-add list*
shows *sum-list-nonneg*: (⋀*x*. *x* ∈ *set xs* ⟹ *x* ≥ *0*) ⟹ *sum-list xs* ≥ *0*
⟨*proof*⟩

lemma *map-filter*:
map f (*filter* (λ*x*. *P* (*f x*)) *xs*) = *filter P* (*map f xs*)
⟨*proof*⟩

lemma
map-of-zip-upto2-length-eq-nth:
assumes *distinct B*
assumes *i* < *length B*
shows (*map-of* (*zip B* [*0*..<*length B*]) (*B* ! *i*)) = *Some i*
⟨*proof*⟩

lemma *distinct-map-fst-snd-eqD*:
distinct (*map fst xs*) ⟹ (*i*, *a*) ∈ *set xs* ⟹ (*i*, *b*) ∈ *set xs* ⟹ *a* = *b*
⟨*proof*⟩

lemma *length-filter-snd-zip*:
length ys = *length xs* ⟹ *length* (*filter* (*p* ∘ *snd*) (*zip ys xs*)) = *length* (*filter p xs*)
⟨*proof*⟩

lemma *filter-snd-nth*:
length ys = *length xs* ⟹ *n* < *length* (*filter p xs*) ⟹
snd (*filter* (*p* ∘ *snd*) (*zip ys xs*) ! *n*) = *filter p xs* ! *n*
⟨*proof*⟩

lemma *distinct-map-snd-fst-eqD*:
distinct (*map snd xs*) ⟹ (*i*, *a*) ∈ *set xs* ⟹ (*j*, *a*) ∈ *set xs* ⟹ *i* = *j*
⟨*proof*⟩

lemma *map-of-mapk-inj-on-SomeI*:
inj-on f (*fst* ‘ (*set t*)) ⟹ *map-of t k* = *Some x* ⟹
map-of (*map* (*case-prod* (λ*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
⟨*proof*⟩

lemma *map-abs-nonneg*[*simp*]:
fixes *xs*::*'a*::*ordered-ab-group-add-abs list*
shows *list-all* (λ*x*. *x* ≥ *0*) *xs* ⟹ *map abs xs* = *xs*
⟨*proof*⟩

lemma *the-inv-into-image-eq*: *inj-on f A* ⟹ *Y* ⊆ *f* ‘ *A* ⟹ *the-inv-into A f* ‘ *Y* = *f* −‘ *Y* ∩ *A*
⟨*proof*⟩

lemma *image-fst-zip*: *length ys* = *length xs* ⟹ *fst* ‘ *set* (*zip ys xs*) = *set ys*
⟨*proof*⟩

lemma *inj-on-fst-set-zip-distinct*[*simp*]:
distinct xs ⟹ *length xs* = *length ys* ⟹ *inj-on fst* (*set* (*zip xs ys*))
⟨*proof*⟩

lemma *mem-greaterThanLessThan-absI*:
fixes *x*::*real*
assumes *abs x* < *1*
shows *x* ∈ {−*1* <..< *1*}
⟨*proof*⟩

lemma *minus-one-less-divideI*: *b* > *0* ⟹ −*b* < *a* ⟹ −*1* < *a* / (*b*::*real*)
⟨*proof*⟩

lemma *divide-less-oneI*: *b* > *0* ⟹ *b* > *a* ⟹ *a* / (*b*::*real*) < *1*
⟨*proof*⟩

lemma *closed-segment-real*:
fixes *a b*::*real*
shows *closed-segment a b* = (*if a* ≤ *b then* {*a* .. *b*} *else* {*b* .. *a*}) (**is** *- = ?if*)
⟨*proof*⟩

2.2 Partial Deviations

typedef (**overloaded**) *'a pdevs* = {*x*::*nat* ⇒ *'a*::*zero. finite* {*i. x i* ≠ *0*}}
— TODO: unify with polynomials
morphisms *pdevs-apply Abs-pdev*
⟨*proof*⟩

setup-lifting *type-definition-pdevs*

lemma *pdevs-eqI*: (⋀*i. pdevs-apply x i* = *pdevs-apply y i*) ⟹ *x* = *y*
⟨*proof*⟩

definition *pdevs-val* :: (*nat* ⇒ *real*) ⇒ *'a*::*real-normed-vector pdevs* ⇒ *'a*
where *pdevs-val e x* = (∑ *i. e i* ∗$_R$ *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*: $x \in valuate\ f \longleftrightarrow (\exists\, e.\ (\forall\, i.\ e\ i \in \{-1\ ..\ 1\}) \wedge x = f\ e)$
⟨*proof*⟩

instantiation *pdevs* :: (*equal*) *equal*
begin

definition *equal-pdevs*::$'a\ pdevs \Rightarrow 'a\ pdevs \Rightarrow bool$
where $equal\text{-}pdevs\ a\ b \longleftrightarrow a = b$

instance ⟨*proof*⟩
end

2.3 Affine Forms

The data structure of affine forms represents particular sets, zonotopes

type-synonym $'a\ aform = 'a \times 'a\ pdevs$

2.4 Evaluation, Range, Joint Range

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

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

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

lemma *Joints-nthE*:
assumes $zs \in Joints\ ZS$
obtains e **where**
$\bigwedge i.\ i < length\ zs \Longrightarrow zs\ !\ i = aform\text{-}val\ e\ (ZS\ !\ i)$
$\bigwedge i.\ e\ i \in \{-1..1\}$
⟨*proof*⟩

lemma *Joints-mapE*:
assumes $ys \in Joints\ YS$
obtains e **where**
$ys = map\ (\lambda x.\ aform\text{-}val\ e\ x)\ YS$
$\bigwedge i.\ e\ i \in \{-1\ ..\ 1\}$
⟨*proof*⟩

lemma
zipped-subset-mapped-Elem:

assumes *xs* = *map* (*aform-val e*) *XS*
assumes *e*: ⋀*i*. *e i* ∈ {−*1* .. *1*}
assumes [*simp*]: *length xs* = *length XS*
assumes [*simp*]: *length ys* = *length YS*
assumes *set* (*zip ys YS*) ⊆ *set* (*zip xs XS*)
shows *ys* = *map* (*aform-val e*) *YS*
⟨*proof*⟩

lemma *Joints-set-zip-subset*:
assumes *xs* ∈ *Joints XS*
assumes *length xs* = *length XS*
assumes *length ys* = *length YS*
assumes *set* (*zip ys YS*) ⊆ *set* (*zip xs XS*)
shows *ys* ∈ *Joints YS*
⟨*proof*⟩

lemma *Joints-set-zip*:
assumes *ys* ∈ *Joints YS*
assumes *length xs* = *length XS*
assumes *length YS* = *length XS*
assumes *sets-eq*: *set* (*zip xs XS*) = *set* (*zip ys YS*)
shows *xs* ∈ *Joints XS*
⟨*proof*⟩

definition *Joints2* ::
'a::*real-normed-vector aform list* ⇒*'b*::*real-normed-vector aform* ⇒ (*'a list* × *'b*)
set
where *Joints2 XS Y* = *valuate* (λ*e*. (*map* (*aform-val e*) *XS*, *aform-val e Y*))

lemma *Joints2E*:
assumes *zs-y* ∈ *Joints2 ZS Y*
obtains *e* **where**
⋀*i*. *i* < *length* (*fst zs-y*) ⟹ (*fst zs-y*) ! *i* = *aform-val e* (*ZS* ! *i*)
snd (*zs-y*) = *aform-val e Y*
⋀*i*. *e i* ∈ {−*1*..*1*}
⟨*proof*⟩

lemma *nth-in-AffineI*:
assumes *xs* ∈ *Joints XS*
assumes *i* < *length XS*
shows *xs* ! *i* ∈ *Affine* (*XS* ! *i*)
⟨*proof*⟩

lemma *Cons-nth-in-Joints1*:
assumes *xs* ∈ *Joints XS*
assumes *i* < *length XS*
shows ((*xs* ! *i*) # *xs*) ∈ *Joints* ((*XS* ! *i*) # *XS*)
⟨*proof*⟩

lemma *Cons-nth-in-Joints2*:
assumes $xs \in Joints\ XS$
assumes $i < length\ XS$
assumes $j < length\ XS$
shows $((xs\ !\ i)\ \#(xs\ !\ j)\ \#\ xs) \in Joints\ ((XS\ !\ i)\#(XS\ !\ j)\ \#\ XS)$
⟨*proof*⟩

lemma *Joints-swap*:
$x\#y\#xs \in Joints\ (X\#Y\#XS) \longleftrightarrow y\#x\#xs \in Joints\ (Y\#X\#XS)$
⟨*proof*⟩

lemma *Joints-swap-Cons-append*:
$length\ xs = length\ XS \Longrightarrow x\#ys@xs \in Joints\ (X\#YS@XS) \longleftrightarrow ys@x\#xs \in Joints\ (YS@X\#XS)$
⟨*proof*⟩

lemma *Joints-ConsD*:
$x\#xs \in Joints\ (X\#XS) \Longrightarrow xs \in Joints\ XS$
⟨*proof*⟩

lemma *Joints-appendD1*:
$ys@xs \in Joints\ (YS@XS) \Longrightarrow length\ xs = length\ XS \Longrightarrow xs \in Joints\ XS$
⟨*proof*⟩

lemma *Joints-appendD2*:
$ys@xs \in Joints\ (YS@XS) \Longrightarrow length\ ys = length\ YS \Longrightarrow ys \in Joints\ YS$
⟨*proof*⟩

lemma *Joints-imp-length-eq*: $xs \in Joints\ XS \Longrightarrow length\ xs = length\ XS$
⟨*proof*⟩

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

2.5 Domain

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

lemma *finite-pdevs-domain*[*intro*, *simp*]: *finite* (*pdevs-domain* x)
⟨*proof*⟩

lemma *in-pdevs-domain*[*simp*]: $i \in pdevs\text{-}domain\ x \longleftrightarrow pdevs\text{-}apply\ x\ i \neq 0$
⟨*proof*⟩

2.6 Least Fresh Index

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

lemma *degree*[*rule-format*, *intro*, *simp*]:

shows $\forall j{\geq}degree\ x.\ pdevs\text{-}apply\ x\ j = 0$
⟨*proof*⟩

lemma *degree-le*:
assumes *d*: $\forall j \geq d.\ pdevs\text{-}apply\ x\ j = 0$
shows $degree\ x \leq d$
⟨*proof*⟩

lemma *degree-gt*: $pdevs\text{-}apply\ x\ j \neq 0 \Longrightarrow degree\ x > j$
⟨*proof*⟩

lemma *pdevs-val-pdevs-domain*: $pdevs\text{-}val\ e\ X = (\sum i{\in}pdevs\text{-}domain\ X.\ e\ i *_R pdevs\text{-}apply\ X\ i)$
⟨*proof*⟩

lemma *pdevs-val-sum-le*: $degree\ X \leq d \Longrightarrow pdevs\text{-}val\ e\ X = (\sum i < d.\ e\ i *_R pdevs\text{-}apply\ X\ i)$
⟨*proof*⟩

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

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

lemma *degree-eqI*:
assumes $pdevs\text{-}apply\ x\ d \neq 0$
assumes $\bigwedge j.\ j > d \Longrightarrow pdevs\text{-}apply\ x\ j = 0$
shows $degree\ x = Suc\ d$
⟨*proof*⟩

lemma *finite-degree-nonzero*[*intro*, *simp*]: $finite\ \{i.\ pdevs\text{-}apply\ x\ i \neq 0\}$
⟨*proof*⟩

lemma *degree-eq-Suc-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*⟩

lemma *pdevs-val-degree-cong*:
assumes $b = d$
assumes $\bigwedge i.\ i < degree\ b \Longrightarrow a\ i = c\ i$
shows $pdevs\text{-}val\ a\ b = pdevs\text{-}val\ c\ d$
⟨*proof*⟩

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

lemma *degree-cong*: $(\bigwedge i.\ (pdevs\text{-}apply\ x\ i = 0) = (pdevs\text{-}apply\ y\ i = 0)) \Longrightarrow degree\ x = degree\ y$

⟨*proof*⟩

lemma *Least-True-nat*[*intro*, *simp*]: (*LEAST i*::*nat*. *True*) = *0*
⟨*proof*⟩

lemma *sorted-list-of-pdevs-domain-eq*:
sorted-list-of-set (*pdevs-domain X*) = *filter* ((≠) *0 o pdevs-apply X*) [*0*..<*degree X*]
⟨*proof*⟩

2.7 Total Deviation

definition *tdev*::*'a*::*ordered-euclidean-space pdevs* ⇒ *'a* **where**
tdev x = (∑ *i*<*degree x*. |*pdevs-apply x i*|)

lemma *abs-pdevs-val-le-tdev*: *e* ∈ *UNIV* → {−*1* .. *1*} ⟹ |*pdevs-val e x*| ≤ *tdev x*
⟨*proof*⟩

2.8 Binary Pointwise Operations

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

lemma *nonzeros-binop-pdevs-subset*:
{*i*. *binop-pdevs-raw f x y i* ≠ *0*} ⊆ {*i*. *x i* ≠ *0*} ∪ {*i*. *y i* ≠ *0*}
⟨*proof*⟩

lift-definition *binop-pdevs*::
(*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a*::*zero pdevs* ⇒ *'b*::*zero pdevs* ⇒ *'c*::*zero pdevs*
is *binop-pdevs-raw*
⟨*proof*⟩

lemma *pdevs-apply-binop-pdevs*[*simp*]: *pdevs-apply* (*binop-pdevs f x y*) *i* =
(*if pdevs-apply x i* = *0* ∧ *pdevs-apply y i* = *0 then 0 else f* (*pdevs-apply x i*) (*pdevs-apply y i*))
⟨*proof*⟩

2.9 Addition

definition *add-pdevs*::*'a*::*real-vector pdevs* ⇒ *'a pdevs* ⇒ *'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*
⟨*proof*⟩

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*⟩

2.10 Total Deviation

lemma *tdev-eq-zero-iff*: **fixes** *X*::*real pdevs* **shows** *tdev X* = *0* ⟷ (∀ *e*. *pdevs-val e X* = *0*)
⟨*proof*⟩

lemma *tdev-nonneg*[*intro*, *simp*]: *tdev X* ≥ *0*
⟨*proof*⟩

lemma *tdev-nonpos-iff*[*simp*]: *tdev X* ≤ *0* ⟷ *tdev X* = *0*
⟨*proof*⟩

2.11 Unary Operations

definition *unop-pdevs-raw*::
('*a*::*zero* ⇒ '*b*::*zero*) ⇒ (*nat* ⇒ '*a*) ⇒ *nat* ⇒ '*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* ≠ *0*} ⊆ {*i*. *x i* ≠ *0*}
⟨*proof*⟩

lift-definition *unop-pdevs*::
('*a* ⇒ '*b*) ⇒ '*a*::*zero pdevs* ⇒ '*b*::*zero pdevs*
is *unop-pdevs-raw*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *pdevs-domain-unop-linear*:
assumes *linear f*
shows *pdevs-domain* (*unop-pdevs f x*) ⊆ *pdevs-domain x*
⟨*proof*⟩

lemma
pdevs-val-unop-linear:
assumes *linear f*
shows *pdevs-val e* (*unop-pdevs f x*) = *f* (*pdevs-val e x*)
⟨*proof*⟩

2.12 Pointwise Scaling of Partial Deviations

definition *scaleR-pdevs*::*real* ⇒ '*a*::*real-vector pdevs* ⇒ '*a pdevs*
where *scaleR-pdevs r x* = *unop-pdevs* ((∗$_R$) *r*) *x*

lemma *pdevs-apply-scaleR-pdevs*[*simp*]:
pdevs-apply (*scaleR-pdevs x Y*) *n* = *x* ∗$_R$ *pdevs-apply Y n*

⟨*proof*⟩

lemma *degree-scaleR-pdevs*[*simp*]: *degree* (*scaleR-pdevs r x*) = (*if r* = *0 then 0 else degree x*)
⟨*proof*⟩

lemma *pdevs-val-scaleR-pdevs*[*simp*]:
fixes *x*::*real* **and** *Y*::*'a*::*real-normed-vector pdevs*
shows *pdevs-val e* (*scaleR-pdevs x Y*) = *x* ∗$_R$ *pdevs-val e Y*
⟨*proof*⟩

2.13 Partial Deviations Scale Pointwise

definition *pdevs-scaleR*::*real pdevs* ⇒ *'a*::*real-vector* ⇒ *'a pdevs*
where *pdevs-scaleR r x* = *unop-pdevs* (λ*r*. *r* ∗$_R$ *x*) *r*

lemma *pdevs-apply-pdevs-scaleR*[*simp*]:
pdevs-apply (*pdevs-scaleR X y*) *n* = *pdevs-apply X n* ∗$_R$ *y*
⟨*proof*⟩

lemma *degree-pdevs-scaleR*[*simp*]: *degree* (*pdevs-scaleR r x*) = (*if x* = *0 then 0 else degree r*)
⟨*proof*⟩

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* ∗$_R$ *y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *degree-uminus-pdevs*[*simp*]: *degree* (*uminus-pdevs x*) = *degree x*
⟨*proof*⟩

lemma *pdevs-val-uminus-pdevs*[*simp*]: *pdevs-val e* (*uminus-pdevs x*) = − *pdevs-val e x*
⟨*proof*⟩

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

lemma *fst-uminus-aform*[*simp*]: *fst* (*uminus-aform Y*) = − *fst Y*
⟨*proof*⟩

lemma *aform-val-uminus-aform*[*simp*]: *aform-val e* (*uminus-aform X*) = − *aform-val e X*
⟨*proof*⟩

2.15 Constant

lift-definition *zero-pdevs*::*'a*::*zero pdevs* **is** λ-. *0* ⟨*proof*⟩

lemma *pdevs-apply-zero-pdevs*[*simp*]: *pdevs-apply zero-pdevs i* = *0*
⟨*proof*⟩

lemma *pdevs-val-zero-pdevs*[*simp*]: *pdevs-val e zero-pdevs* = *0*
⟨*proof*⟩

definition *num-aform f* = (*f*, *zero-pdevs*)

2.16 Inner Product

definition *pdevs-inner*::*'a*::*euclidean-space pdevs* ⇒ *'a* ⇒ *real pdevs*
where *pdevs-inner x b* = *unop-pdevs* (λ*x*. *x* ⋅ *b*) *x*

lemma *pdevs-apply-pdevs-inner*[*simp*]: *pdevs-apply* (*pdevs-inner p a*) *i* = *pdevs-apply p i* ⋅ *a*
⟨*proof*⟩

lemma *pdevs-val-pdevs-inner*[*simp*]: *pdevs-val e* (*pdevs-inner p a*) = *pdevs-val e p* ⋅ *a*
⟨*proof*⟩

definition *inner-aform*::*'a*::*euclidean-space aform* ⇒ *'a* ⇒ *real aform*
where *inner-aform X b* = (*fst X* ⋅ *b*, *pdevs-inner* (*snd X*) *b*)

2.17 Inner Product Pair

definition *inner2*::*'a*::*euclidean-space* ⇒ *'a* ⇒ *'a* ⇒ *real*∗*real*
where *inner2 x n l* = (*x* ⋅ *n*, *x* ⋅ *l*)

definition *pdevs-inner2*::*'a*::*euclidean-space pdevs* ⇒ *'a* ⇒ *'a* ⇒ (*real*∗*real*) *pdevs*
where *pdevs-inner2 X n l* = *unop-pdevs* (λ*x*. *inner2 x n l*) *X*

lemma *pdevs-apply-pdevs-inner2*[*simp*]: *pdevs-apply* (*pdevs-inner2 p a b*) *i* = (*pdevs-apply p i* ⋅ *a*, *pdevs-apply p i* ⋅ *b*)
⟨*proof*⟩

definition *inner2-aform*::*'a*::*euclidean-space aform* ⇒ *'a* ⇒ *'a* ⇒ (*real*∗*real*) *aform*
where *inner2-aform X a b* = (*inner2* (*fst X*) *a b*, *pdevs-inner2* (*snd X*) *a b*)

lemma *linear-inner2*[*intro*, *simp*]: *linear* (λ*x*. *inner2 x n i*)
⟨*proof*⟩

lemma *aform-val-inner2-aform*[*simp*]: *aform-val e* (*inner2-aform Z n i*) = *inner2* (*aform-val e Z*) *n i*
⟨*proof*⟩

2.18 Update

lemma *pdevs-val-upd*[*simp*]:
pdevs-val (*e*(*n* := *e'*)) *X* = *pdevs-val e X* − *e n* ∗ *pdevs-apply X n* + *e'* ∗ *pdevs-apply X n*
⟨*proof*⟩

lemma *nonzeros-fun-upd*:
{*i*. (*f*(*n* := *a*)) *i* ≠ *0*} ⊆ {*i*. *f i* ≠ *0*} ∪ {*n*}
⟨*proof*⟩

lift-definition *pdev-upd*::'*a*::*real-vector pdevs* ⇒ *nat* ⇒ '*a* ⇒ '*a pdevs*
is λ*x n a*. *x*(*n*:=*a*)
⟨*proof*⟩

lemma *pdevs-apply-pdev-upd*[*simp*]:
pdevs-apply (*pdev-upd X n x*) = (*pdevs-apply X*)(*n*:=*x*)
⟨*proof*⟩

lemma *pdevs-val-pdev-upd*[*simp*]:
pdevs-val e (*pdev-upd X n x*) = *pdevs-val e X* + *e n* ∗$_R$ *x* − *e n* ∗$_R$ *pdevs-apply X n*
⟨*proof*⟩

lemma *degree-pdev-upd*:
assumes *x* = *0* ⟷ *pdevs-apply X n* = *0*
shows *degree* (*pdev-upd X n x*) = *degree X*
⟨*proof*⟩

lemma *degree-pdev-upd-le*:
assumes *degree X* ≤ *n*
shows *degree* (*pdev-upd X n x*) ≤ *Suc n*
⟨*proof*⟩

2.19 Inf/Sup

definition *Inf-aform X* = *fst X* − *tdev* (*snd X*)

definition *Sup-aform X* = *fst X* + *tdev* (*snd X*)

lemma *Inf-aform*:
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
shows *Inf-aform X* ≤ *aform-val e X*
⟨*proof*⟩

lemma *Sup-aform*:

assumes $e \in UNIV \to \{-1 \mathinner{..} 1\}$
shows *aform-val e X* $\leq$ *Sup-aform X*
⟨*proof*⟩

2.20 Minkowski Sum

definition *msum-pdevs-raw*::*nat*⇒(*nat* ⇒ *'a*::*real-vector*)⇒(*nat* ⇒ *'a*)⇒(*nat*⇒*'a*)
where
msum-pdevs-raw n x y i = (*if i* < *n then x i else y* (*i* − *n*))

lemma *nonzeros-msum-pdevs-raw*:
{*i*. *msum-pdevs-raw n f g i* ≠ *0*} = ({*0*..<*n*} ∩ {*i*. *f i* ≠ *0*}) ∪ (+) *n* ' ({*i*. *g i* ≠ *0*})
⟨*proof*⟩

lift-definition *msum-pdevs*::*nat*⇒*'a*::*real-vector pdevs*⇒*'a pdevs*⇒*'a pdevs* **is** *msum-pdevs-raw*
⟨*proof*⟩

lemma *pdevs-apply-msum-pdevs*: *pdevs-apply* (*msum-pdevs n f g*) *i* =
(*if i* < *n then pdevs-apply f i else pdevs-apply g* (*i* − *n*))
⟨*proof*⟩

lemma *degree-least-nonzero*:
assumes *degree f* ≠ *0*
shows *pdevs-apply f* (*degree f* − *1*) ≠ *0*
⟨*proof*⟩

lemma *degree-leI*:
assumes (⋀*i*. *pdevs-apply y i* = *0* ⟹ *pdevs-apply x i* = *0*)
shows *degree x* ≤ *degree y*
⟨*proof*⟩

lemma *degree-msum-pdevs-ge1*:
shows *degree f* ≤ *n* ⟹ *degree f* ≤ *degree* (*msum-pdevs n f g*)
⟨*proof*⟩

lemma *degree-msum-pdevs-ge2*:
assumes *degree f* ≤ *n*
shows *degree g* ≤ *degree* (*msum-pdevs n f g*) − *n*
⟨*proof*⟩

lemma *degree-msum-pdevs-le*:
shows *degree* (*msum-pdevs n f g*) ≤ *n* + *degree g*
⟨*proof*⟩

lemma
sum-msum-pdevs-cases:
assumes *degree f* ≤ *n*
assumes [*simp*]: ⋀*i*. *e i 0* = *0*

shows
$(\sum i <_{degree}\ (msum\text{-}pdevs\ n\ f\ g).$
$e\ i\ (if\ i < n\ then\ pdevs\text{-}apply\ f\ i\ else\ pdevs\text{-}apply\ g\ (i - n))) =$
$(\sum i <_{degree} f.\ e\ i\ (pdevs\text{-}apply\ f\ i)) + (\sum i <_{degree} g.\ e\ (i + n)\ (pdevs\text{-}apply\ g\ i))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *tdev-msum-pdevs*: $degree\ f \leq n \Longrightarrow tdev\ (msum\text{-}pdevs\ n\ f\ g) = tdev\ f + tdev\ g$
⟨*proof*⟩

lemma *pdevs-val-msum-pdevs*:
$degree\ f \leq n \Longrightarrow pdevs\text{-}val\ e\ (msum\text{-}pdevs\ n\ f\ g) = pdevs\text{-}val\ e\ f + pdevs\text{-}val\ (\lambda i.\ e\ (i + n))\ g$
⟨*proof*⟩

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

lemma *fst-msum-aform*[*simp*]: $fst\ (msum\text{-}aform\ n\ f\ g) = fst\ f + fst\ g$
⟨*proof*⟩

lemma *snd-msum-aform*[*simp*]: $snd\ (msum\text{-}aform\ n\ f\ g) = msum\text{-}pdevs\ n\ (snd\ f)\ (snd\ g)$
⟨*proof*⟩

lemma *finite-nonzero-summable*: $finite\ \{i.\ f\ i \neq 0\} \Longrightarrow summable\ f$
⟨*proof*⟩

lemma *aform-val-msum-aform*:
assumes $degree\text{-}aform\ f \leq n$
shows $aform\text{-}val\ e\ (msum\text{-}aform\ n\ f\ g) = aform\text{-}val\ e\ f + aform\text{-}val\ (\lambda i.\ e\ (i + n))\ g$
⟨*proof*⟩

lemma *Inf-aform-msum-aform*:
$degree\text{-}aform\ X \leq n \Longrightarrow Inf\text{-}aform\ (msum\text{-}aform\ n\ X\ Y) = Inf\text{-}aform\ X + Inf\text{-}aform\ Y$
⟨*proof*⟩

lemma *Sup-aform-msum-aform*:
$degree\text{-}aform\ X \leq n \Longrightarrow Sup\text{-}aform\ (msum\text{-}aform\ n\ X\ Y) = Sup\text{-}aform\ X + Sup\text{-}aform\ Y$
⟨*proof*⟩

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

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

lemma *degree-zero-pdevs*[*simp*]: *degree zero-pdevs = 0*
⟨*proof*⟩

lemma *independent-aform-Joints*:
assumes $x \in$ *Affine X*
assumes $y \in$ *Affine Y*
shows $[x, y] \in$ *Joints* $[X,$ *independent-aform X Y*$]$
⟨*proof*⟩

lemma *msum-aform-Joints*:
assumes $d \geq$ *degree-aform X*
assumes $\bigwedge X.\ X \in$ *set XS* $\Longrightarrow d \geq$ *degree-aform X*
assumes $(x\#xs) \in$ *Joints* $(X\#XS)$
assumes $y \in$ *Affine Y*
shows $((x + y)\#x\#xs) \in$ *Joints* $(msum\text{-}aform\ d\ X\ Y\#X\#XS)$
⟨*proof*⟩

lemma *Joints-msum-aform*:
assumes $d \geq$ *degree-aform X*
assumes $\bigwedge Y.\ Y \in$ *set YS* $\Longrightarrow d \geq$ *degree-aform Y*
shows *Joints* $(msum\text{-}aform\ d\ X\ Y\#YS) = \{((x + y)\#ys)\ |x\ y\ ys.\ y \in$ *Affine Y* $\wedge\ x\#ys \in$ *Joints* $(X\#YS)\}$
⟨*proof*⟩

lemma *Joints-singleton-image*: *Joints* $[x] = (\lambda x.\ [x])$ ‘ *Affine x*
⟨*proof*⟩

lemma *Collect-extract-image*: $\{g\ (f\ x\ y)\ |x\ y.\ P\ x\ y\} = g$ ‘ $\{f\ x\ y\ |x\ y.\ P\ x\ y\}$
⟨*proof*⟩

lemma *inj-Cons*: *inj* $(\lambda x.\ x\#xs)$
⟨*proof*⟩

lemma *Joints-Nil*[*simp*]: *Joints* $[] = \{[]\}$
⟨*proof*⟩

lemma *msum-pdevs-zero-ident*[*simp*]: *msum-pdevs 0 zero-pdevs x = x*
⟨*proof*⟩

lemma *msum-aform-zero-ident*[*simp*]: *msum-aform 0* $(0,$ *zero-pdevs*$)$ *x = x*
⟨*proof*⟩

lemma *mem-Joints-singleton*: $(x \in$ *Joints* $[X]) = (\exists\, y.\ x = [y] \wedge y \in$ *Affine X*$)$
⟨*proof*⟩

lemma *singleton-mem-Joints*[*simp*]: $[x] \in$ *Joints* $[X] \longleftrightarrow x \in$ *Affine X*
⟨*proof*⟩

lemma *msum-aform-Joints-without-first*:
assumes $d \geq$ *degree-aform* X
assumes $\bigwedge X.\ X \in$ *set* $XS \Longrightarrow d \geq$ *degree-aform* X
assumes $(x\#xs) \in$ *Joints* $(X\#XS)$
assumes $y \in$ *Affine* Y
assumes $z = x + y$
shows $z\#xs \in$ *Joints* (*msum-aform* d X $Y\#XS$)
⟨*proof*⟩

lemma *Affine-msum-aform*:
assumes $d \geq$ *degree-aform* X
shows *Affine* (*msum-aform* d X Y) = $\{x + y \mid x\ y.\ x \in$ *Affine* $X \wedge y \in$ *Affine* $Y\}$
⟨*proof*⟩

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

lemma *Affine-independent-aform*:
Affine (*independent-aform* X Y) = *Affine* Y
⟨*proof*⟩

lemma
abs-diff-eq1:
fixes l u::'a::*ordered-euclidean-space*
shows $l \leq u \Longrightarrow |u - l| = u - l$
⟨*proof*⟩

lemma *compact-sum*:
fixes f :: 'a ⇒ 'b::*topological-space* ⇒ 'c::*real-normed-vector*
assumes *finite* I
assumes $\bigwedge i.\ i \in I \Longrightarrow$ *compact* $(S\ i)$
assumes $\bigwedge i.\ i \in I \Longrightarrow$ *continuous-on* $(S\ i)$ $(f\ i)$
assumes $I \subseteq J$
shows *compact* $\{\sum i{\in}I.\ f\ i\ (x\ i) \mid x.\ x \in Pi\ J\ S\}$
⟨*proof*⟩

lemma *compact-Affine*:
fixes X::'a::*ordered-euclidean-space aform*
shows *compact* (*Affine* X)
⟨*proof*⟩

lemma *Joints2-JointsI*:
$(xs,\ x) \in$ *Joints2* XS $X \Longrightarrow x\#xs \in$ *Joints* $(X\#XS)$
⟨*proof*⟩

2.21 Splitting

definition *split-aform* X i =

(*let xi = pdevs-apply (snd X) i /$_R$ 2*
in ((fst X − xi, pdev-upd (snd X) i xi), (fst X + xi, pdev-upd (snd X) i xi)))

lemma *split-aformE*:
assumes $e \in UNIV \to \{-1 .. 1\}$
assumes x = *aform-val e X*
obtains *err* **where** x = *aform-val* (*e*(*i*:=*err*)) (*fst* (*split-aform X i*)) *err* $\in \{-1 .. 1\}$
| *err* **where** x = *aform-val* (*e*(*i*:=*err*)) (*snd* (*split-aform X i*)) *err* $\in \{-1 .. 1\}$
⟨*proof*⟩

lemma *pdevs-val-add*: *pdevs-val* ($\lambda i.$ *e i* + *f i*) *xs* = *pdevs-val e xs* + *pdevs-val f xs*
⟨*proof*⟩

lemma *pdevs-val-minus*: *pdevs-val* ($\lambda i.$ *e i* − *f i*) *xs* = *pdevs-val e xs* − *pdevs-val f xs*
⟨*proof*⟩

lemma *pdevs-val-cmul*: *pdevs-val* ($\lambda i.$ *u* ∗ *e i*) *xs* = *u* ∗$_R$ *pdevs-val e xs*
⟨*proof*⟩

lemma *atLeastAtMost-absI*: − *a* ≤ *a* ⟹ |*x*::*real*| ≤ |*a*| ⟹ *x* ∈ *atLeastAtMost* (− *a*) *a*
⟨*proof*⟩

lemma *divide-atLeastAtMost-1-absI*: |*x*::*real*| ≤ |*a*| ⟹ *x*/*a* ∈ $\{-1 .. 1\}$
⟨*proof*⟩

lemma *convex-scaleR-aux*: *u* + *v* = 1 ⟹ *u* ∗$_R$ *x* + *v* ∗$_R$ *x* = (*x*::*'a*::*real-vector*)
⟨*proof*⟩

lemma *convex-mult-aux*: *u* + *v* = 1 ⟹ *u* ∗ *x* + *v* ∗ *x* = (*x*::*real*)
⟨*proof*⟩

lemma *convex-Affine*: *convex* (*Affine X*)
⟨*proof*⟩

lemma *segment-in-aform-val*:
assumes $e \in UNIV \to \{-1 .. 1\}$
assumes $f \in UNIV \to \{-1 .. 1\}$
shows *closed-segment* (*aform-val e X*) (*aform-val f X*) ⊆ *Affine X*
⟨*proof*⟩

2.22 From List of Generators

lift-definition *pdevs-of-list*::*'a*::*zero list* ⇒ *'a pdevs*
is λ*xs i. if i* < *length xs then xs* ! *i else 0*
⟨*proof*⟩

lemma *pdevs-apply-pdevs-of-list*:
pdevs-apply (*pdevs-of-list xs*) *i* = (*if i* < *length xs then xs* ! *i else 0*)
⟨*proof*⟩

lemma *pdevs-apply-pdevs-of-list-Nil*[*simp*]:
pdevs-apply (*pdevs-of-list* []) *i* = *0*
⟨*proof*⟩

lemma *pdevs-apply-pdevs-of-list-Cons*:
pdevs-apply (*pdevs-of-list* (*x* # *xs*)) *i* =
(*if i* = *0 then x else pdevs-apply* (*pdevs-of-list xs*) (*i* − *1*))
⟨*proof*⟩

lemma *pdevs-domain-pdevs-of-list-Cons*[*simp*]: *pdevs-domain* (*pdevs-of-list* (*x* # *xs*)) =
(*if x* = *0 then* {} *else* {*0*}) ∪ (+) *1* ‘ *pdevs-domain* (*pdevs-of-list xs*)
⟨*proof*⟩

lemma *pdevs-val-pdevs-of-list-eq*[*simp*]:
pdevs-val e (*pdevs-of-list* (*x* # *xs*)) = *e 0* ∗$_R$ *x* + *pdevs-val* (*e o* (+) *1*) (*pdevs-of-list xs*)
⟨*proof*⟩

lemma
less-degree-pdevs-of-list-imp-less-length:
assumes *i* < *degree* (*pdevs-of-list xs*)
shows *i* < *length xs*
⟨*proof*⟩

lemma *tdev-pdevs-of-list*[*simp*]: *tdev* (*pdevs-of-list xs*) = *sum-list* (*map abs xs*)
⟨*proof*⟩

lemma *pdevs-of-list-Nil*[*simp*]: *pdevs-of-list* [] = *zero-pdevs*
⟨*proof*⟩

lemma *pdevs-val-inj-sumI*:
fixes *K*::′*a set* **and** *g*::′*a* ⇒ *nat*
assumes *finite K*
assumes *inj-on g K*
assumes *pdevs-domain x* ⊆ *g* ‘ *K*
assumes ⋀*i*. *i* ∈ *K* ⟹ *g i* ∉ *pdevs-domain x* ⟹ *f i* = *0*
assumes ⋀*i*. *i* ∈ *K* ⟹ *g i* ∈ *pdevs-domain x* ⟹ *f i* = *e* (*g i*) ∗$_R$ *pdevs-apply x* (*g i*)
shows *pdevs-val e x* = (∑ *i*∈*K*. *f i*)
⟨*proof*⟩

lemma *pdevs-domain-pdevs-of-list-le*: *pdevs-domain* (*pdevs-of-list xs*) ⊆ {*0*..<*length xs*}

⟨*proof*⟩

lemma *pdevs-val-zip*: *pdevs-val e* (*pdevs-of-list xs*) = (∑ (*i*,*x*)←*zip* [*0*..<*length xs*] *xs*. *e i* ∗$_R$ *x*)
⟨*proof*⟩

lemma *pdevs-val-map*:
‹*pdevs-val e* (*pdevs-of-list xs*)
= (∑ *n*←[*0*..<*length xs*]. *e n* ∗$_R$ *xs* ! *n*)›
⟨*proof*⟩

lemma *scaleR-sum-list*:
fixes *xs*::'*a*::*real-vector list*
shows *a* ∗$_R$ *sum-list xs* = *sum-list* (*map* (*scaleR a*) *xs*)
⟨*proof*⟩

lemma *pdevs-val-const-pdevs-of-list*: *pdevs-val* (λ-. *c*) (*pdevs-of-list xs*) = *c* ∗$_R$ *sum-list xs*
⟨*proof*⟩

lemma *pdevs-val-partition*:
assumes *e* ∈ *UNIV* → *I*
obtains *f g* **where** *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*))
f ∈ *UNIV* → *I*
g ∈ *UNIV* → *I*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *degree-pdevs-of-list-le-length*[*intro*, *simp*]: *degree* (*pdevs-of-list xs*) ≤ *length xs*
⟨*proof*⟩

lemma *degree-pdevs-of-list-append*:
degree (*pdevs-of-list* (*xs* @ *ys*)) ≤ *length xs* + *degree* (*pdevs-of-list ys*)
⟨*proof*⟩

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*⟩

lemma
sum-general-mono:
fixes *f*::*'a*⇒(*'b*::*ordered-ab-group-add*)
assumes [*simp*,*intro*]: *finite s finite t*
assumes *f*: ⋀*x*. *x ∈ s − t* ⟹ *f x ≤ 0*
assumes *g*: ⋀*x*. *x ∈ t − s* ⟹ *g x ≥ 0*
assumes *fg*: ⋀*x*. *x ∈ s ∩ t* ⟹ *f x ≤ g x*
shows (∑ *x ∈ s*. *f x*) ≤ (∑ *x ∈ t*. *g x*)
⟨*proof*⟩

lemma *degree-pdevs-of-list-eq'*:
‹*degree* (*pdevs-of-list xs*) = *Min* {*n*. *n* ≤ *length xs* ∧ (∀ *m*. *n* ≤ *m* ⟶ *m* < *length xs* ⟶ *xs* ! *m* = *0*)}›
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *set-distinct-permI*: *set xs* = *set ys* ⟹ *distinct xs* ⟹ *distinct ys* ⟹ *xs* <~~> *ys*
⟨*proof*⟩

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

lemma *partition-permI*:
filter p xs @ *filter* (*Not o p*) *xs* <~~> *xs*
⟨*proof*⟩

lemma *pdevs-val-eqI*:
assumes $\bigwedge i.\ i \in$ *pdevs-domain* $y \Longrightarrow i \in$ *pdevs-domain* $x \Longrightarrow$
$e\ i *_R$ *pdevs-apply* $x\ i = f\ i *_R$ *pdevs-apply* $y\ i$
assumes $\bigwedge i.\ i \in$ *pdevs-domain* $y \Longrightarrow i \notin$ *pdevs-domain* $x \Longrightarrow f\ i *_R$ *pdevs-apply* $y\ i = 0$
assumes $\bigwedge i.\ i \in$ *pdevs-domain* $x \Longrightarrow i \notin$ *pdevs-domain* $y \Longrightarrow e\ i *_R$ *pdevs-apply* $x\ i = 0$
shows *pdevs-val* $e\ x =$ *pdevs-val* $f\ y$
⟨*proof*⟩

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

lemma *filter-pdevs-raw-nonzeros*: $\{i.$ *filter-pdevs-raw* $s\ f\ i \neq 0\} = \{i.\ f\ i \neq 0\} \cap \{x.\ s\ x\ (f\ x)\}$
⟨*proof*⟩

lift-definition *filter-pdevs*::$(nat \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a$::*real-vector pdevs* $\Rightarrow 'a$ *pdevs*
is *filter-pdevs-raw*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *degree-filter-pdevs-le*: *degree* (*filter-pdevs* $I\ x) \leq$ *degree* x
⟨*proof*⟩

lemma *pdevs-val-filter-pdevs*:
pdevs-val e (*filter-pdevs* $I\ x) =$
$(\sum i \in \{..<$*degree* $x\} \cap \{i.\ I\ i$ (*pdevs-apply* $x\ i)\}.\ e\ i *_R$ *pdevs-apply* $x\ i)$
⟨*proof*⟩

lemma *pdevs-val-filter-pdevs-dom*:
pdevs-val e (*filter-pdevs* $I\ x) =$
$(\sum i \in$ *pdevs-domain* $x \cap \{i.\ I\ i$ (*pdevs-apply* $x\ i)\}.\ e\ i *_R$ *pdevs-apply* $x\ i)$
⟨*proof*⟩

lemma *pdevs-val-filter-pdevs-eval*:
pdevs-val e (*filter-pdevs* $p\ x) =$ *pdevs-val* $(\lambda i.$ *if* $p\ i$ (*pdevs-apply* $x\ i$) *then* $e\ i$ *else* $0)\ x$
⟨*proof*⟩

definition *pdevs-applys* $X\ i =$ *map* $(\lambda x.$ *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* $(\lambda i$ -. $I\ i$ (*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 (λ-. *0*) *X*)
⟨*proof*⟩

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

abbreviation *degree-aforms X* ≡ *degrees* (*map snd X*)

lemma *degrees-leI*:
assumes ⋀*x*. *x* ∈ *set X* ⟹ *degree x* ≤ *K*
shows *degrees X* ≤ *K*
⟨*proof*⟩

lemma *degrees-leD*:
assumes *degrees X* ≤ *K*
shows ⋀*x*. *x* ∈ *set X* ⟹ *degree x* ≤ *K*
⟨*proof*⟩

lemma *degree-filter-pdevs-list-le*: *degrees* (*filter-pdevs-list I x*) ≤ *degrees x*
⟨*proof*⟩

definition *dense-list-of-pdevs x* = *map* (λ*i*. *pdevs-apply x i*) [*0*..<*degree x*]

2.22.1 (reverse) ordered coefficients as list

definition *list-of-pdevs x* =
map (λ*i*. (*i*, *pdevs-apply x i*)) (*rev* (*sorted-list-of-set* (*pdevs-domain x*)))

lemma *list-of-pdevs-zero-pdevs*[*simp*]: *list-of-pdevs zero-pdevs* = []
⟨*proof*⟩

lemma *sum-list-list-of-pdevs*: *sum-list* (*map snd* (*list-of-pdevs x*)) = *sum-list* (*dense-list-of-pdevs x*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *pdevs-of-list-dense-list-of-pdevs*: *pdevs-of-list* (*dense-list-of-pdevs x*) = *x*
⟨*proof*⟩

lemma *pdevs-val-sum-list*: *pdevs-val* (λ-. *c*) *X* = *c* ∗$_R$ *sum-list* (*map snd* (*list-of-pdevs*

X))
⟨*proof*⟩

lemma *list-of-pdevs-all-nonzero*: *list-all* (λ*x*. *x* ≠ *0*) (*map snd* (*list-of-pdevs xs*))
⟨*proof*⟩

lemma *list-of-pdevs-nonzero*: *x* ∈ *set* (*map snd* (*list-of-pdevs xs*)) ⟹ *x* ≠ *0*
⟨*proof*⟩

lemma *pdevs-of-list-scaleR-0*[*simp*]:
fixes *xs*::'*a*::*real-vector list*
shows *pdevs-of-list* (*map* ((∗$_R$) *0*) *xs*) = *zero-pdevs*
⟨*proof*⟩

lemma *degree-pdevs-of-list-scaleR*:
degree (*pdevs-of-list* (*map* ((∗$_R$) *c*) *xs*)) = (*if c* ≠ *0 then degree* (*pdevs-of-list xs*) *else 0*)
⟨*proof*⟩

lemma *list-of-pdevs-eq*:
rev (*list-of-pdevs X*) = (*filter* ((≠) *0 o snd*) (*map* (λ*i*. (*i*, *pdevs-apply X i*)) [*0*..<*degree X*]))
(**is** - = *filter ?P* (*map ?f ?xs*))
⟨*proof*⟩

lemma *sum-list-take-pdevs-val-eq*:
sum-list (*take d xs*) = *pdevs-val* (λ*i*. *if i* < *d then 1 else 0*) (*pdevs-of-list xs*)
⟨*proof*⟩

lemma *zero-in-range-pdevs-apply*[*intro*, *simp*]:
fixes *X*::'*a*::*real-vector pdevs* **shows** *0* ∈ *range* (*pdevs-apply X*)
⟨*proof*⟩

lemma *dense-list-in-range*: *x* ∈ *set* (*dense-list-of-pdevs X*) ⟹ *x* ∈ *range* (*pdevs-apply X*)
⟨*proof*⟩

lemma *not-in-dense-list-zeroD*:
assumes *pdevs-apply X i* ∉ *set* (*dense-list-of-pdevs X*)
shows *pdevs-apply X i* = *0*
⟨*proof*⟩

lemma *list-all-list-of-pdevsI*:
assumes ⋀*i*. *i* ∈ *pdevs-domain X* ⟹ *P* (*pdevs-apply X i*)
shows *list-all* (λ*x*. *P x*) (*map snd* (*list-of-pdevs X*))
⟨*proof*⟩

lemma *pdevs-of-list-map-scaleR*:
pdevs-of-list (*map* (*scaleR r*) *xs*) = *scaleR-pdevs r* (*pdevs-of-list xs*)

⟨*proof*⟩

lemma
map-permI:
assumes *xs* <~~> *ys*
shows *map f xs* <~~> *map f ys*
⟨*proof*⟩

lemma *rev-perm*: *rev xs* <~~> *ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

lemma *list-of-pdevs-perm-filter-nonzero*:
map snd (*list-of-pdevs X*) <~~> (*filter* ((≠) *0*) (*dense-list-of-pdevs X*))
⟨*proof*⟩

lemma *pdevs-val-filter*:
assumes *mem*: *e* ∈ *UNIV* → *I*
assumes *0* ∈ *I*
obtains *e′* **where**
pdevs-val e (*pdevs-of-list* (*filter p xs*)) = *pdevs-val e′* (*pdevs-of-list xs*)
e′ ∈ *UNIV* → *I*
⟨*proof*⟩

lemma
pdevs-val-of-list-of-pdevs:
assumes *e* ∈ *UNIV* → *I*
assumes *0* ∈ *I*
obtains *e′* **where**
pdevs-val e (*pdevs-of-list* (*map snd* (*list-of-pdevs X*))) = *pdevs-val e′ X*
e′ ∈ *UNIV* → *I*
⟨*proof*⟩

lemma
pdevs-val-of-list-of-pdevs2:
assumes *e* ∈ *UNIV* → *I*
obtains *e′* **where**
pdevs-val e X = *pdevs-val e′* (*pdevs-of-list* (*map snd* (*list-of-pdevs X*)))
e′ ∈ *UNIV* → *I*
⟨*proof*⟩

lemma *dense-list-of-pdevs-scaleR*:
r ≠ *0* ⟹ *map* ((∗$_R$) *r*) (*dense-list-of-pdevs x*) = *dense-list-of-pdevs* (*scaleR-pdevs r x*)
⟨*proof*⟩

lemma *degree-pdevs-of-list-eq*:
(⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*) ⟹ *degree* (*pdevs-of-list xs*) = *length xs*
⟨*proof*⟩

lemma *dense-list-of-pdevs-pdevs-of-list*:
(⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*) ⟹ *dense-list-of-pdevs* (*pdevs-of-list xs*) = *xs*
⟨*proof*⟩

lemma *pdevs-of-list-sum*:
assumes *distinct xs*
assumes *e* ∈ *UNIV* → *I*
obtains *f* **where** *f* ∈ *UNIV* → *I pdevs-val e* (*pdevs-of-list xs*) = (∑ *P*∈*set xs*. *f P* ∗$_R$ *P*)
⟨*proof*⟩

lemma *pdevs-domain-eq-pdevs-of-list*:
assumes *nz*: ⋀*x*. *x* ∈ *set* (*xs*) ⟹ *x* ≠ *0*
shows *pdevs-domain* (*pdevs-of-list xs*) = {*0*..<*length xs*}
⟨*proof*⟩

lemma *length-list-of-pdevs-pdevs-of-list*:
assumes *nz*: ⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*
shows *length* (*list-of-pdevs* (*pdevs-of-list xs*)) = *length xs*
⟨*proof*⟩

lemma *nth-list-of-pdevs-pdevs-of-list*:
assumes *nz*: ⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*
assumes *l*: *n* < *length xs*
shows *list-of-pdevs* (*pdevs-of-list xs*) ! *n* = ((*length xs* − *Suc n*), *xs* ! (*length xs* − *Suc n*))
⟨*proof*⟩

lemma *list-of-pdevs-pdevs-of-list-eq*:
(⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*) ⟹
list-of-pdevs (*pdevs-of-list xs*) = *zip* (*rev* [*0*..<*length xs*]) (*rev xs*)
⟨*proof*⟩

lemma *sum-list-filter-list-of-pdevs-of-list*:
fixes *xs*::'*a*::*comm-monoid-add list*
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*
shows *sum-list* (*filter p* (*map snd* (*list-of-pdevs* (*pdevs-of-list xs*)))) = *sum-list* (*filter p xs*)
⟨*proof*⟩

lemma
sum-list-partition:
fixes *xs*::'*a*::*comm-monoid-add list*
shows *sum-list* (*filter p xs*) + *sum-list* (*filter* (*Not o p*) *xs*) = *sum-list xs*
⟨*proof*⟩

2.23 2d zonotopes

definition *prod-of-pdevs x y* = *binop-pdevs Pair x y*

lemma *apply-pdevs-prod-of-pdevs*[*simp*]:
pdevs-apply (*prod-of-pdevs x y*) *i* = (*pdevs-apply x i*, *pdevs-apply y i*)
⟨*proof*⟩

lemma *pdevs-domain-prod-of-pdevs*[*simp*]:
pdevs-domain (*prod-of-pdevs x y*) = *pdevs-domain x* ∪ *pdevs-domain y*
⟨*proof*⟩

lemma *pdevs-val-prod-of-pdevs*[*simp*]:
pdevs-val e (*prod-of-pdevs x y*) = (*pdevs-val e x*, *pdevs-val e y*)
⟨*proof*⟩

definition *prod-of-aforms* (**infixr** ×$_a$ *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* ⇒ *'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*)..}
⟨*proof*⟩

lemma *nonzeros-One-pdevs-raw*:
{*i*. *One-pdevs-raw i* ≠ (*0*::*'a*::*executable-euclidean-space*)} = − {*length* (*Basis-list*::*'a list*)..}
⟨*proof*⟩

lift-definition *One-pdevs*::*'a*::*executable-euclidean-space pdevs* **is** *One-pdevs-raw*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Max-Collect-less-nat*: *Max* {*i*::*nat*. *i* < *k*} = (*if k* = *0 then Max* {} *else k* − *1*)
⟨*proof*⟩

lemma *degree-One-pdevs*[*simp*]: *degree* (*One-pdevs*::*'a pdevs*) =
length (*Basis-list*::*'a*::*executable-euclidean-space list*)
⟨*proof*⟩

definition *inner-scaleR-pdevs*::*'a*::*euclidean-space* ⇒ *'a pdevs* ⇒ *'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* · (*pdevs-apply x i*)) $*_R$ (*pdevs-apply x i*)
⟨*proof*⟩

lemma *degree-inner-scaleR-pdevs-le*:
degree (*inner-scaleR-pdevs* (*l*::*'a*::*executable-euclidean-space*) *One-pdevs*) ≤
degree (*One-pdevs*::*'a pdevs*)
⟨*proof*⟩

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*) ≤ *DIM*(*'a*)
⟨*proof*⟩

lemma *pdevs-apply-pdevs-of-ivl*:
defines *B* ≡ *Basis-list*::*'a*::*executable-euclidean-space list*
shows *pdevs-apply* (*pdevs-of-ivl l u*) *i* = (*if i* < *length B then* ((*u* − *l*)·(*B*!*i*)/*2*)$*_R$(*B*!*i*) *else 0*)
⟨*proof*⟩

lemma *deg-length-less-imp*[*simp*]:
k < *degree* (*pdevs-of-ivl l u*::*'a*::*executable-euclidean-space pdevs*) ⟹
k < *length* (*Basis-list*::*'a list*)
⟨*proof*⟩

lemma *tdev-pdevs-of-ivl*: *tdev* (*pdevs-of-ivl l u*) = |*u* − *l*| /$_R$ *2*
⟨*proof*⟩

definition *aform-of-ivl l u* = ((*l* + *u*)/$_R$*2*, *pdevs-of-ivl l u*)

definition *aform-of-point x* = *aform-of-ivl x x*

lemma *Elem-affine-of-ivl-le*:
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *l* ≤ *u*
shows *l* ≤ *aform-val e* (*aform-of-ivl l u*)
⟨*proof*⟩

lemma *Elem-affine-of-ivl-ge*:
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *l* ≤ *u*
shows *aform-val e* (*aform-of-ivl l u*) ≤ *u*
⟨*proof*⟩

lemma

map-of-zip-upto-length-eq-nth:
assumes $i < \mathit{length}\ B$
assumes $d = \mathit{length}\ B$
shows $(\mathit{map\text{-}of}\ (\mathit{zip}\ [0..<d]\ B)\ i) = \mathit{Some}\ (B\ !\ i)$
⟨*proof*⟩

lemma *in-ivl-affine-of-ivlE*:
assumes $k \in \{l\ ..\ u\}$
obtains e **where** $e \in \mathit{UNIV} \to \{-1\ ..\ 1\}\ k = \mathit{aform\text{-}val}\ e\ (\mathit{aform\text{-}of\text{-}ivl}\ l\ u)$
⟨*proof*⟩

lemma *Inf-aform-aform-of-ivl*:
assumes $l \leq u$
shows $\mathit{Inf\text{-}aform}\ (\mathit{aform\text{-}of\text{-}ivl}\ l\ u) = l$
⟨*proof*⟩

lemma *Sup-aform-aform-of-ivl*:
assumes $l \leq u$
shows $\mathit{Sup\text{-}aform}\ (\mathit{aform\text{-}of\text{-}ivl}\ l\ u) = u$
⟨*proof*⟩

lemma *Affine-aform-of-ivl*:
$a \leq b \Longrightarrow \mathit{Affine}\ (\mathit{aform\text{-}of\text{-}ivl}\ a\ b) = \{a\ ..\ b\}$
⟨*proof*⟩

end

3 Operations on Expressions

theory *Floatarith-Expression*
imports
HOL−Decision-Procs.Approximation
Affine-Arithmetic-Auxiliarities
Executable-Euclidean-Space
begin

Much of this could move to the distribution...

3.1 Approximating Expression*s*

unbundle *floatarith-notation*

primrec *interpret-floatariths* :: *floatarith list* ⇒ *real list* ⇒ *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*
⟨*proof*⟩

lemma *interpret-floatariths-nth*[*simp*]:
interpret-floatariths fas xs ! *n* = *interpret-floatarith* (*fas* ! *n*) *xs*
if *n* < *length fas*
⟨*proof*⟩

abbreviation *einterpret* ≡ λ*fas vs. eucl-of-list* (*interpret-floatariths fas vs*)

3.2 Syntax

syntax *interpret-floatarith*::*floatarith* ⇒ *real list* ⇒ *real*

instantiation *floatarith* :: {*plus*, *minus*, *uminus*, *times*, *inverse*, *zero*, *one*}
begin

definition − *f* = *Minus f*
lemma *interpret-floatarith-uminus*[*simp*]:
interpret-floatarith (− *f*) *xs* = − *interpret-floatarith f xs*
⟨*proof*⟩

definition *f* + *g* = *Add f g*
lemma *interpret-floatarith-plus*[*simp*]:
interpret-floatarith (*f* + *g*) *xs* = *interpret-floatarith f xs* + *interpret-floatarith g xs*
⟨*proof*⟩

definition *f* − *g* = *Add f* (*Minus g*)
lemma *interpret-floatarith-minus*[*simp*]:
interpret-floatarith (*f* − *g*) *xs* = *interpret-floatarith f xs* − *interpret-floatarith g xs*
⟨*proof*⟩

definition *inverse f* = *Inverse f*
lemma *interpret-floatarith-inverse*[*simp*]:
interpret-floatarith (*inverse f*) *xs* = *inverse* (*interpret-floatarith f xs*)
⟨*proof*⟩

definition *f* ∗ *g* = *Mult f g*
lemma *interpret-floatarith-times*[*simp*]:
interpret-floatarith (*f* ∗ *g*) *xs* = *interpret-floatarith f xs* ∗ *interpret-floatarith g xs*
⟨*proof*⟩

definition *f div g* = *f* ∗ *Inverse g*
lemma *interpret-floatarith-divide*[*simp*]:
interpret-floatarith (*f div g*) *xs* = *interpret-floatarith f xs* / *interpret-floatarith g xs*

⟨*proof*⟩

definition *1 = Num 1*
lemma *interpret-floatarith-one*[*simp*]:
interpret-floatarith 1 xs = 1
⟨*proof*⟩

definition *0 = Num 0*
lemma *interpret-floatarith-zero*[*simp*]:
interpret-floatarith 0 xs = 0
⟨*proof*⟩

instance ⟨*proof*⟩
end

3.3 Derived symbols

definition R_e *r* = (*case quotient-of r of* (*n*, *d*) ⇒ *Num* (*of-int n*) / *Num* (*of-int d*))
declare [[*coercion* R_e]]

lemma *interpret-*R_e[*simp*]: *interpret-floatarith* (R_e *x*) *xs* = *real-of-rat x*
⟨*proof*⟩

definition *Sin x* = *Cos* ((*Pi* ∗ (*Num* (*Float 1* (−*1*)))) − *x*)

lemma *interpret-floatarith-Sin*[*simp*]:
interpret-floatarith (*Sin x*) *vs* = *sin* (*interpret-floatarith x vs*)
⟨*proof*⟩

definition *Half x* = *Mult* (*Num* (*Float 1* (−*1*))) *x*
lemma *interpret-Half*[*simp*]: *interpret-floatarith* (*Half x*) *xs* = *interpret-floatarith x xs* / *2*
⟨*proof*⟩

definition *Tan x* = (*Sin x*) / (*Cos x*)

lemma *interpret-floatarith-Tan*[*simp*]:
interpret-floatarith (*Tan x*) *vs* = *tan* (*interpret-floatarith x vs*)
⟨*proof*⟩

primrec Sum_e **where**
Sum_e *f* [] = *0*
| Sum_e *f* (*x*#*xs*) = *f x* + Sum_e *f xs*

lemma *interpret-floatarith-*Sum_e[*simp*]:
interpret-floatarith (Sum_e *f x*) *vs* = (∑ *i*←*x*. *interpret-floatarith* (*f i*) *vs*)
⟨*proof*⟩

definition *Norm* **where** *Norm is* = *Sqrt* (*Sum*$_e$ (λ*i*. *i* ∗ *i*) *is*)

lemma *interpret-floatarith-norm*[*simp*]:
assumes [*simp*]: *length x* = *DIM*('*a*)
shows *interpret-floatarith* (*Norm x*) *vs* = *norm* (*einterpret x vs*::'*a*::*executable-euclidean-space*)
⟨*proof*⟩

notation *floatarith.Power* (**infixr** $\widehat{}_e$ *80*)

3.4 Constant Folding

fun *dest-Num-fa* **where**
dest-Num-fa (*floatarith.Num x*) = *Some x*
| *dest-Num-fa* - = *None*

fun-cases *dest-Num-fa-None*: *dest-Num-fa fa* = *None*
and *dest-Num-fa-Some*: *dest-Num-fa fa* = *Some x*

fun *fold-const-fa* **where**
fold-const-fa (*Add 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 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
  ⟨proof⟩
```

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*,*max-Var-floatarith b*, *max-Var-floatarith c*}
| *max-Var-form* (*Assign a b c*) = *linorder-class.Max* {*max-Var-floatarith a*,*max-Var-floatarith b*, *max-Var-form c*}

lemma
interpret-floatarith-eq-take-max-VarI:
assumes *take* (*max-Var-floatarith ra*) *ys* = *take* (*max-Var-floatarith ra*) *zs*
shows *interpret-floatarith ra ys* = *interpret-floatarith ra zs*
⟨*proof*⟩

lemma
interpret-floatariths-eq-take-max-VarI:
assumes *take* (*max-Var-floatariths ea*) *ys* = *take* (*max-Var-floatariths ea*) *zs*
shows *interpret-floatariths ea ys* = *interpret-floatariths ea zs*
⟨*proof*⟩

lemma *Max-Image-distrib*:
includes *no-floatarith-notation*
assumes *finite X X* ≠ {}
shows *Max* ((λ*x*. *max* (*f1 x*) (*f2 x*)) ‘ *X*) = *max* (*Max* (*f1* ‘ *X*)) (*Max* (*f2* ‘ *X*))
⟨*proof*⟩

lemma *max-Var-floatarith-simps*[*simp*]:
max-Var-floatarith (*a* / *b*) = *max* (*max-Var-floatarith a*) (*max-Var-floatarith b*)
max-Var-floatarith (*a* ∗ *b*) = *max* (*max-Var-floatarith a*) (*max-Var-floatarith b*)
max-Var-floatarith (*a* + *b*) = *max* (*max-Var-floatarith a*) (*max-Var-floatarith b*)
max-Var-floatarith (*a* − *b*) = *max* (*max-Var-floatarith a*) (*max-Var-floatarith b*)

max-Var-floatarith (− *b*) = (*max-Var-floatarith b*)
⟨*proof*⟩

lemma *max-Var-floatariths-Max*:
max-Var-floatariths xs = (*if set xs* = {} *then 0 else linorder-class.Max* (*max-Var-floatarith* ‘ *set xs*))
⟨*proof*⟩

lemma *max-Var-floatariths-map-plus*[*simp*]:
max-Var-floatariths (*map* (λ*i*. *fa1 i* + *fa2 i*) *xs*) = *max* (*max-Var-floatariths* (*map fa1 xs*)) (*max-Var-floatariths* (*map fa2 xs*))
⟨*proof*⟩

lemma *max-Var-floatariths-map-times*[*simp*]:
max-Var-floatariths (*map* (λ*i*. *fa1 i* ∗ *fa2 i*) *xs*) = *max* (*max-Var-floatariths* (*map fa1 xs*)) (*max-Var-floatariths* (*map fa2 xs*))
⟨*proof*⟩

lemma *max-Var-floatariths-map-divide*[*simp*]:
max-Var-floatariths (*map* (λ*i*. *fa1 i* / *fa2 i*) *xs*) = *max* (*max-Var-floatariths* (*map fa1 xs*)) (*max-Var-floatariths* (*map fa2 xs*))
⟨*proof*⟩

lemma *max-Var-floatariths-map-uminus*[*simp*]:
max-Var-floatariths (*map* (λ*i*. − *fa1 i*) *xs*) = *max-Var-floatariths* (*map fa1 xs*)
⟨*proof*⟩

lemma *max-Var-floatariths-map-const*[*simp*]:
max-Var-floatariths (*map* (λ*i*. *fa*) *xs*) = (*if xs* = [] *then 0 else max-Var-floatarith fa*)
⟨*proof*⟩

lemma *max-Var-floatariths-map-minus*[*simp*]:
max-Var-floatariths (*map* (λ*i*. *fa1 i* − *fa2 i*) *xs*) = *max* (*max-Var-floatariths* (*map fa1 xs*)) (*max-Var-floatariths* (*map fa2 xs*))
⟨*proof*⟩

primrec *fresh-floatarith* **where**
fresh-floatarith (*Var y*) *x* ⟷ (*x* ≠ *y*)
| *fresh-floatarith* (*Num a*) *x* ⟷ *True*
| *fresh-floatarith Pi x* ⟷ *True*
| *fresh-floatarith* (*Cos a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Abs a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Arctan a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Sqrt a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Exp a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Floor a*) *x* ⟷ *fresh-floatarith a x*

| *fresh-floatarith* (*Power a n*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Minus a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Ln a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Inverse a*) *x* ⟷ *fresh-floatarith a x*
| *fresh-floatarith* (*Add a b*) *x* ⟷ *fresh-floatarith a x* ∧ *fresh-floatarith b x*
| *fresh-floatarith* (*Mult a b*) *x* ⟷ *fresh-floatarith a x* ∧ *fresh-floatarith b x*
| *fresh-floatarith* (*Max a b*) *x* ⟷ *fresh-floatarith a x* ∧ *fresh-floatarith b x*
| *fresh-floatarith* (*Min a b*) *x* ⟷ *fresh-floatarith a x* ∧ *fresh-floatarith b x*
| *fresh-floatarith* (*Powr a b*) *x* ⟷ *fresh-floatarith a x* ∧ *fresh-floatarith b x*

lemma *fresh-floatarith-subst*:
fixes *v*::*float*
assumes *fresh-floatarith e x*
assumes *x* < *length vs*
shows *interpret-floatarith e* (*vs*[*x*:=*v*]) = *interpret-floatarith e vs*
⟨*proof*⟩

lemma *fresh-floatarith-max-Var*:
assumes *max-Var-floatarith ea* ≤ *i*
shows *fresh-floatarith ea i*
⟨*proof*⟩

primrec *fresh-floatariths* **where**
fresh-floatariths [] *x* ⟷ *True*
| *fresh-floatariths* (*a*#*as*) *x* ⟷ *fresh-floatarith a x* ∧ *fresh-floatariths as x*

lemma *fresh-floatariths-max-Var*:
assumes *max-Var-floatariths ea* ≤ *i*
shows *fresh-floatariths ea i*
⟨*proof*⟩

lemma
interpret-floatariths-take-eqI:
assumes *take n ys* = *take n zs*
assumes *max-Var-floatariths ea* ≤ *n*
shows *interpret-floatariths ea ys* = *interpret-floatariths ea zs*
⟨*proof*⟩

lemma
interpret-floatarith-fresh-eqI:
assumes ⋀*i*. *fresh-floatarith ea i* ∨ (*i* < *length ys* ∧ *i* < *length zs* ∧ *ys* ! *i* = *zs*
! *i*)
shows *interpret-floatarith ea ys* = *interpret-floatarith ea zs*
⟨*proof*⟩

lemma
interpret-floatariths-fresh-eqI:
assumes ⋀*i*. *fresh-floatariths ea i* ∨ (*i* < *length ys* ∧ *i* < *length zs* ∧ *ys* ! *i* = *zs*
! *i*)

shows *interpret-floatariths ea ys = interpret-floatariths ea zs*
⟨*proof*⟩

lemma
interpret-floatarith-max-Var-cong:
assumes $\bigwedge i.\ i < \textit{max-Var-floatarith}\ f \Longrightarrow \textit{xs}\ !\ i = \textit{ys}\ !\ i$
shows *interpret-floatarith f ys = interpret-floatarith f xs*
⟨*proof*⟩

lemma
interpret-floatarith-fresh-cong:
assumes $\bigwedge i.\ \neg \textit{fresh-floatarith}\ f\ i \Longrightarrow \textit{xs}\ !\ i = \textit{ys}\ !\ i$
shows *interpret-floatarith f ys = interpret-floatarith f xs*
⟨*proof*⟩

lemma *max-Var-floatarith-le-max-Var-floatariths*:
$\textit{fa} \in \textit{set fas} \Longrightarrow \textit{max-Var-floatarith fa} \leq \textit{max-Var-floatariths fas}$
⟨*proof*⟩

lemma *max-Var-floatarith-le-max-Var-floatariths-nth*:
$n < \textit{length fas} \Longrightarrow \textit{max-Var-floatarith}\ (\textit{fas}\ !\ n) \leq \textit{max-Var-floatariths fas}$
⟨*proof*⟩

lemma *max-Var-floatariths-leI*:
assumes $\bigwedge i.\ i < \textit{length xs} \Longrightarrow \textit{max-Var-floatarith}\ (\textit{xs}\ !\ i) \leq F$
shows $\textit{max-Var-floatariths xs} \leq F$
⟨*proof*⟩

lemma *fresh-floatariths-map-Var*[*simp*]:
$\textit{fresh-floatariths}\ (\textit{map floatarith.Var xs})\ i \longleftrightarrow i \notin \textit{set xs}$
⟨*proof*⟩

lemma *max-Var-floatarith-fold-const-fa*:
$\textit{max-Var-floatarith}\ (\textit{fold-const-fa fa}) \leq \textit{max-Var-floatarith fa}$
⟨*proof*⟩

lemma *max-Var-floatariths-fold-const-fa*:
$\textit{max-Var-floatariths}\ (\textit{map fold-const-fa xs}) \leq \textit{max-Var-floatariths xs}$
⟨*proof*⟩

lemma *interpret-form-max-Var-cong*:
assumes $\bigwedge i.\ i < \textit{max-Var-form}\ f \Longrightarrow \textit{xs}\ !\ i = \textit{ys}\ !\ i$
shows *interpret-form f xs = interpret-form f ys*
⟨*proof*⟩

lemma *max-Var-floatariths-lessI*: $i < \textit{max-Var-floatarith}\ (\textit{fas}\ !\ j) \Longrightarrow j < \textit{length fas} \Longrightarrow i < \textit{max-Var-floatariths fas}$
⟨*proof*⟩

lemma *interpret-floatariths-max-Var-cong*:
assumes $\bigwedge i.\ i < \textit{max-Var-floatariths}\ f \Longrightarrow \textit{xs}\ !\ i = \textit{ys}\ !\ i$
shows *interpret-floatariths f ys* = *interpret-floatariths f xs*
⟨*proof*⟩

lemma *max-Var-floatarithimage-Var*[*simp*]: *max-Var-floatarith* ‘ *Var* ‘ *X* = *Suc* ‘ *X* ⟨*proof*⟩

lemma *max-Var-floatariths-map-Var*[*simp*]:
max-Var-floatariths (*map Var xs*) = (*if xs* = [] *then 0 else Suc* (*linorder-class.Max* (*set xs*)))
⟨*proof*⟩

lemma *Max-atLeastLessThan-nat*[*simp*]: $a < b \Longrightarrow \textit{linorder-class.Max}\ \{a..<b\} = b - 1$ **for** *a b*::*nat*
⟨*proof*⟩

3.6 Derivatives

lemma *isDERIV-Power-iff*: *isDERIV j* (*Power fa n*) *xs* = (*if n* = *0 then True else isDERIV j fa xs*)
⟨*proof*⟩

lemma *isDERIV-max-Var-floatarithI*:
assumes *isDERIV n f ys*
assumes $\bigwedge i.\ i < \textit{max-Var-floatarith}\ f \Longrightarrow \textit{xs}\ !\ i = \textit{ys}\ !\ i$
shows *isDERIV n f xs*
⟨*proof*⟩

definition *isFDERIV* **where** *isFDERIV n xs fas vs* ⟷
$(\forall\, i{<}n.\ \forall\, j{<}n.\ \textit{isDERIV}\ (\textit{xs}\ !\ i)\ (\textit{fas}\ !\ j)\ \textit{vs}) \wedge \textit{length fas} = n \wedge \textit{length xs} = n$

lemma *isFDERIV-I*: $(\bigwedge i\ j.\ i < n \Longrightarrow j < n \Longrightarrow \textit{isDERIV}\ (\textit{xs}\ !\ i)\ (\textit{fas}\ !\ j)\ \textit{vs}) \Longrightarrow$
length fas = *n* ⟹ *length xs* = *n* ⟹ *isFDERIV n xs fas vs*
⟨*proof*⟩

lemma *isFDERIV-isDERIV-D*: *isFDERIV n xs fas vs* ⟹ $i < n \Longrightarrow j < n \Longrightarrow$ *isDERIV* (*xs* ! *i*) (*fas* ! *j*) *vs*
⟨*proof*⟩

lemma *isFDERIV-lengthD*: *length fas* = *n length xs* = *n* **if** *isFDERIV n xs fas vs*
⟨*proof*⟩

lemma *isFDERIV-uptD*: *isFDERIV n* [*0*..<*n*] *fas vs* ⟹ $i < n \Longrightarrow j < n \Longrightarrow$ *isDERIV i* (*fas* ! *j*) *vs*
⟨*proof*⟩

lemma *isFDERIV-max-Var-congI*: *isFDERIV n xs fas ws*
if *f*: *isFDERIV n xs fas vs* **and** *c*: $(\bigwedge i.\ i < max\text{-}Var\text{-}floatariths\ fas \Longrightarrow vs\ !\ i = ws\ !\ i)$
⟨*proof*⟩

lemma *isFDERIV-max-Var-cong*: *isFDERIV n xs fas ws* $\longleftrightarrow$ *isFDERIV n xs fas vs*
if *c*: $(\bigwedge i.\ i < max\text{-}Var\text{-}floatariths\ fas \Longrightarrow vs\ !\ i = ws\ !\ i)$
⟨*proof*⟩

lemma *isDERIV-max-VarI*:
$i \geq max\text{-}Var\text{-}floatarith\ fa \Longrightarrow isDERIV\ j\ fa\ xs \Longrightarrow isDERIV\ i\ fa\ xs$
⟨*proof*⟩

lemmas *max-Var-floatarith-le-max-Var-floatariths-nthI* =
max-Var-floatarith-le-max-Var-floatariths-nth[*THEN order-trans*]

lemma
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*
⟨*proof*⟩

lemma *interpret-floatariths-Var*[*simp*]:
interpret-floatariths (*map floatarith.Var xs*) *vs* = (*map* (*nth vs*) *xs*)
⟨*proof*⟩

lemma *max-Var-floatariths-append*[*simp*]: *max-Var-floatariths* $(xs\ @\ ys) = max$ (*max-Var-floatariths xs*) (*max-Var-floatariths ys*)
⟨*proof*⟩

lemma *map-nth-append-upt*[*simp*]:
assumes $a \geq length\ xs$
shows *map* ((!) $(xs\ @\ ys)$) $[a..<b] = map\ ((!)\ ys)\ [a - length\ xs..<b - length\ xs]$
⟨*proof*⟩

lemma *map-nth-Cons-upt*[*simp*]:
assumes $a > 0$
shows *map* ((!) $(x\ \#\ ys)$) $[a..<b] = map\ ((!)\ ys)\ [a - Suc\ 0..<b - Suc\ 0]$
⟨*proof*⟩

lemma *map-nth-eq-self*[*simp*]:
shows *length fas* $= l \Longrightarrow$ (*map* ((!) *fas*) $[0..<l]$) = *fas*
⟨*proof*⟩

lemma
isFDERIV-appendI1:
assumes *isFDERIV J* [*0*..<*J*] (*es*) *xs*
assumes $\bigwedge i\ j.\ i < J + K \Longrightarrow j < K \Longrightarrow$ *isDERIV i* (*rs* ! *j*) *xs*
assumes *length es* = *J*
assumes *length rs* = *K*
assumes *max-Var-floatariths es* ≤ *J*
shows *isFDERIV* (*J* + *K*) [*0*..<*J* + *K*] (*es* @ *rs*) *xs*
⟨*proof*⟩

lemma *matrix-matrix-mult-zero*[*simp*]:
a ** *0* = *0* *0* ** *a* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma
matrix-blinfun-compose:
fixes *A* *B*::(*real* ^ 'n) ⇒$_L$ (*real* ^ 'n)
shows *matrix* (*A* *o_L *B*) = (*matrix A*) ** (*matrix B*)
⟨*proof*⟩

lemma *matrix-add-rdistrib*: ((*B* + *C*) ** *A*) = (*B* ** *A*) + (*C* ** *A*)
⟨*proof*⟩

lemma *matrix-scaleR-right*: *r* *$_R$ (*a*::'*a*::*real-algebra-1*^'*n*^'*m*) ** *b* = *r* *$_R$ (*a* ** *b*)
⟨*proof*⟩

lemma *matrix-scaleR-left*: (*a*::'*a*::*real-algebra-1*^'*n*^'*m*) ** *r* *$_R$ *b* = *r* *$_R$ (*a* ** *b*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *norm-axis*: *norm* (*axis ia* 1::'*a*::{*real-normed-algebra-1*}^'*n*) = *1*
⟨*proof*⟩

lemma *abs-vec-nth-blinfun-apply-lemma*:
fixes *x*::(*real*^'*n*) ⇒$_L$ (*real*^'*m*)

shows *abs* (*vec-nth* (*blinfun-apply x* (*axis ia 1*)) *i*) ≤ *norm x*
⟨*proof*⟩

lemma *bounded-linear-matrix-blinfun-apply*: *bounded-linear* (λ*x*::(*real*^'*n*) ⇒$_L$ (*real*^'*m*). *matrix* (*blinfun-apply x*))
⟨*proof*⟩

lemma *matrix-has-derivative*:
shows ((λ*x*::(*real*^'*n*)⇒$_L$(*real*^'*n*). *matrix* (*blinfun-apply x*)) *has-derivative* (λ*h*. *matrix* (*blinfun-apply h*))) (*at x*)
⟨*proof*⟩

lemma
matrix-comp-has-derivative[*derivative-intros*]:
fixes *f*::'*a*::*real-normed-vector* ⇒ ((*real*^'*n*)⇒$_L$(*real*^'*n*))
assumes (*f has-derivative f'*) (*at x within S*)
shows ((λ*x*. *matrix* (*blinfun-apply* (*f x*))) *has-derivative* (λ*x*. *matrix* (*f'* *x*))) (*at x within S*)
⟨*proof*⟩

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* =
(∑ *i*<*length ys*. (*interpret-floatariths xs vs* ! *i*) ∗ (*interpret-floatariths ys vs* ! *i*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

definition *FDERIV-floatarith* **where**
FDERIV-floatarith fa xs d = *inner-floatariths* (*map* (λ*x*. *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* (λ*x*. *interpret-floatarith* (*f x*) *vs*) *xs*
⟨*proof*⟩

lemma *max-Var-floatarith-DERIV-floatarith*:
max-Var-floatarith (*DERIV-floatarith x fa*) ≤ *max-Var-floatarith fa*
⟨*proof*⟩

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*)
⟨*proof*⟩

definition *FDERIV-floatariths* **where**
FDERIV-floatariths fas xs d = *map* (λ*fa*. *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*)
⟨*proof*⟩

lemma *length-FDERIV-floatariths*[*simp*]:
length (*FDERIV-floatariths fas xs ds*) = *length fas*
⟨*proof*⟩

lemma *FDERIV-floatariths-nth*[*simp*]:
i < *length fas* ⟹ *FDERIV-floatariths fas xs ds* ! *i* = *FDERIV-floatarith* (*fas* ! *i*) *xs ds*
⟨*proof*⟩

definition *FDERIV-n-floatariths fas xs ds n* = ((λ*x*. *FDERIV-floatariths x xs ds*)^^*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-floatariths* (*FDERIV-n-floatariths fa xs ds n*) *xs ds*
⟨*proof*⟩

lemma *length-FDERIV-n-floatariths*[*simp*]: *length* (*FDERIV-n-floatariths fa xs ds n*) = *length fa*
⟨*proof*⟩

lemma *max-Var-floatarith-FDERIV-n-floatariths*:
length xs = length d ⟹ max-Var-floatariths (FDERIV-n-floatariths fa xs d n)
≤ max (max-Var-floatariths fa) (max-Var-floatariths d)
⟨*proof*⟩

lemma *interpret-floatarith-FDERIV-floatarith-cong*:
assumes *rq*: *⋀i. i < max-Var-floatarith f ⟹ 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
⟨*proof*⟩

theorem
matrix-vector-mult-eq-list-of-eucl-nth:
*(M::real^'n::enum^'m::enum) *v v =*
(∑ i<CARD('m).
*(∑ j<CARD('n). list-of-eucl M ! (i * CARD('n) + j) * list-of-eucl v ! j) *R*
Basis-list ! i)
⟨*proof*⟩

definition *mmult-fa l m n AS BS =*
concat (map (λi. map (λk. inner-floatariths
*(map (λj. AS ! (i * m + j)) [0..<m]) (map (λj. BS ! (j * n + k)) [0..<m]))*
[0..<n]) [0..<l])

lemma *length-mmult-fa*[*simp*]: *length (mmult-fa l m n AS BS) = l * n*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *isDERIV-Power*: *isDERIV x (fa) vs* $\Longrightarrow$ *isDERIV x (fa* $\hat{}_e$ *n) vs*
⟨*proof*⟩

lemma *isDERIV-mmult-fa-nth*:
assumes $\bigwedge j.\ j < D * E \Longrightarrow$ *isDERIV i (A ! j) xs*
assumes $\bigwedge j.\ j < E * F \Longrightarrow$ *isDERIV i (B ! j) xs*
assumes [*simp*]: *length A = D * E length B = E * F j < D * F*
shows *isDERIV i (mmult-fa D E F A B ! j) xs*
⟨*proof*⟩

definition *mvmult-fa n m AS B =*
*map (λi. inner-floatariths (map (λj. AS ! (i * m + j)) [0..<m]) (map (λj. B ! j) [0..<m])) [0..<n]*

lemma *einterpret-mvmult-fa*:
assumes [*simp*]: *Dn = CARD('n::enum) Dm = CARD('m::enum)*
*length A = CARD('n)*CARD('m) length B = CARD('m)*
shows *einterpret (mvmult-fa Dn Dm A B) vs*::((*real*, *'m*::*enum*) *vec*, *'n*) *vec*) $*v$ (*einterpret B vs*::(*real*, *'m*) *vec*)
⟨*proof*⟩

lemma *max-Var-floatariths-mvult-fa*:
assumes [*simp*]: *length A = D * E length B = E*
shows *max-Var-floatariths (mvmult-fa D E A B)* $\leq$ *max (max-Var-floatariths A) (max-Var-floatariths B)*
⟨*proof*⟩

lemma *isDERIV-mvmult-fa-nth*:
assumes $\bigwedge j.\ j < D * E \Longrightarrow$ *isDERIV i (A ! j) xs*
assumes $\bigwedge j.\ j < E \Longrightarrow$ *isDERIV i (B ! j) xs*
assumes [*simp*]: *length A = D * E length B = E j < D*
shows *isDERIV i (mvmult-fa D E A B ! j) xs*
⟨*proof*⟩

lemma *max-Var-floatariths-mapI*:
assumes $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow$ *max-Var-floatarith (f x)* $\leq m$
shows *max-Var-floatariths (map f xs)* $\leq m$
⟨*proof*⟩

lemma
max-Var-floatariths-list-updateI:
assumes *max-Var-floatariths xs* $\leq m$
assumes *max-Var-floatarith v* $\leq m$
assumes *i < length xs*
shows *max-Var-floatariths (xs[i := v])* $\leq m$
⟨*proof*⟩

lemma

max-Var-floatariths-replicateI:
assumes *max-Var-floatarith v* ≤ *m*
shows *max-Var-floatariths* (*replicate n v*) ≤ *m*
⟨*proof*⟩

definition *FDERIV-n-floatarith fa xs ds n* = ((λ*x*. *FDERIV-floatarith x xs ds*) ^^ *n*) *fa*
lemma *FDERIV-n-floatariths-nth*: *i* < *length fas* ⟹ *FDERIV-n-floatariths fas xs ds n* ! *i* = *FDERIV-n-floatarith* (*fas* ! *i*) *xs ds n*
⟨*proof*⟩

lemma *einterpret-fold-const-fa*[*simp*]:
(*einterpret* (*map* (λ*i*. *fold-const-fa* (*fa i*)) *xs*) *vs*::*'a*::*executable-euclidean-space*) =
einterpret (*map fa xs*) *vs* **if** *length xs* = *DIM*(*'a*)
⟨*proof*⟩

lemma *einterpret-plus*[*simp*]:
shows (*einterpret* (*map* (λ*i*. *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*
⟨*proof*⟩

lemma *einterpret-uminus*[*simp*]:
shows (*einterpret* (*map* (λ*i*. − *fa1 i*) [*0*..<*DIM*(*'a*)]) *vs*::*'a*::*executable-euclidean-space*) =
− *einterpret* (*map fa1* [*0*..<*DIM*(*'a*)]) *vs*
⟨*proof*⟩

lemma *diff-floatarith-conv-add-uminus*: *a* − *b* = *a* + − *b* **for** *a b*::*floatarith*
⟨*proof*⟩

lemma *einterpret-minus*[*simp*]:
shows (*einterpret* (*map* (λ*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*
⟨*proof*⟩

lemma *einterpret-scaleR*[*simp*]:
shows (*einterpret* (*map* (λ*i*. *fa1* ∗ *fa2 i*) [*0*..<*DIM*(*'a*)]) *vs*::*'a*::*executable-euclidean-space*) =
interpret-floatarith (*fa1*) *vs* ∗$_R$ *einterpret* (*map fa2* [*0*..<*DIM*(*'a*)]) *vs*
⟨*proof*⟩

lemma *einterpret-nth*[*simp*]:
assumes [*simp*]: *length xs* = *DIM*(*'a*)
shows (*einterpret* (*map* ((!) *xs*) [*0*..<*DIM*(*'a*)]) *vs*::*'a*::*executable-euclidean-space*)

$= einterpret\ xs\ vs$
⟨*proof*⟩

type-synonym $'n\ rvec = (real,\ 'n)\ vec$

lemma *length-mvmult-fa*[*simp*]: $length\ (mvmult\text{-}fa\ D\ E\ xs\ ys) = D$
⟨*proof*⟩

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 $interpret\text{-}floatarith\ (mvmult\text{-}fa\ D\ E\ xs\ ys\ !\ n)\ vs =$
$((einterpret\ xs\ vs::((real,\ 'm)\ vec,\ 'n)\ vec) *v einterpret\ ys\ vs) \cdot (Basis\text{-}list\ !\ n)$
⟨*proof*⟩

lemmas [*simp del*] = *fold-const-fa.simps*

lemma *take-eq-map-nth*: $n < length\ xs \Longrightarrow take\ n\ xs = map\ ((!)\ xs)\ [0..<n]$
⟨*proof*⟩

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*: $interpret\text{-}floatarith\ (floatarith.Num\ f)\ vs = (real\text{-}of\text{-}float\ f)$
⟨*proof*⟩

⟨*ML*⟩

schematic-goal *reify-example*:
$[xs!i * xs!j,\ xs!i + xs!j\ powr\ (sin\ (xs!0)),\ xs!k + (2\ /\ 3 * xs!i * xs!j)] = interpret\text{-}floatariths\ ?fas\ xs$
⟨*proof*⟩

⟨*ML*⟩

lemma *eucl-of-list-interpret-floatariths-cong*:
fixes $y::'a::executable\text{-}euclidean\text{-}space$
assumes $\bigwedge b.\ b \in Basis \Longrightarrow interpret\text{-}floatarith\ (fa\ (index\ Basis\text{-}list\ b))\ vs = y \cdot b$
assumes $length\ xs = DIM('a)$
shows $eucl\text{-}of\text{-}list\ (interpret\text{-}floatariths\ (map\ fa\ [0..<DIM('a)])\ vs) = y$

⟨*proof*⟩

lemma *interpret-floatariths-fold-const-fa*[*simp*]:
interpret-floatariths (*map fold-const-fa ds*) = *interpret-floatariths ds*
⟨*proof*⟩

fun *subst-floatarith* **where**
subst-floatarith s (*Add a b*) = *Add* (*subst-floatarith s a*) (*subst-floatarith s b*)
|
subst-floatarith s (*Mult a b*) = *Mult* (*subst-floatarith s a*) (*subst-floatarith s b*) |
subst-floatarith s (*Minus a*) = *Minus* (*subst-floatarith s a*) |
subst-floatarith s (*Inverse a*) = *Inverse* (*subst-floatarith s a*) |
subst-floatarith s (*Cos a*) = *Cos* (*subst-floatarith s a*) |
subst-floatarith s (*Arctan a*) = *Arctan* (*subst-floatarith s a*) |
subst-floatarith s (*Min a b*) = *Min* (*subst-floatarith s a*) (*subst-floatarith s b*) |
subst-floatarith s (*Max a b*) = *Max* (*subst-floatarith s a*) (*subst-floatarith s b*) |
subst-floatarith s (*Abs a*) = *Abs* (*subst-floatarith s a*) |
subst-floatarith s Pi = *Pi* |
subst-floatarith s (*Sqrt a*) = *Sqrt* (*subst-floatarith s a*) |
subst-floatarith s (*Exp a*) = *Exp* (*subst-floatarith s a*) |
subst-floatarith s (*Powr a b*) = *Powr* (*subst-floatarith s a*) (*subst-floatarith s b*) |
subst-floatarith s (*Ln a*) = *Ln* (*subst-floatarith s a*) |
subst-floatarith s (*Power a i*) = *Power* (*subst-floatarith s a*) *i* |
subst-floatarith s (*Floor a*) = *Floor* (*subst-floatarith s a*) |
subst-floatarith s (*Num f*) = *Num f* |
subst-floatarith s (*Var n*) = *s n*

lemma *interpret-floatarith-subst-floatarith*:
assumes *max-Var-floatarith fa* ≤ *D*
shows *interpret-floatarith* (*subst-floatarith s fa*) *vs* =
interpret-floatarith fa (*map* (λ*i*. *interpret-floatarith* (*s i*) *vs*) [*0*..<*D*])
⟨*proof*⟩

lemma *max-Var-floatarith-subst-floatarith-le*[*THEN order-trans*]:
assumes *length xs* ≥ *max-Var-floatarith fa*
shows *max-Var-floatarith* (*subst-floatarith* ((!) *xs*) *fa*) ≤ *max-Var-floatariths xs*
⟨*proof*⟩

lemma *max-Var-floatariths-subst-floatarith-le*[*THEN order-trans*]:
assumes *length xs* ≥ *max-Var-floatariths fas*
shows *max-Var-floatariths* (*map* (*subst-floatarith* ((!) *xs*)) *fas*) ≤ *max-Var-floatariths xs*
⟨*proof*⟩

fun *continuous-on-floatarith* :: *floatarith* ⇒ *bool* **where**

continuous-on-floatarith (*Add a b*) = (*continuous-on-floatarith a* ∧ *continuous-on-floatarith b*) |
continuous-on-floatarith (*Mult a b*) = (*continuous-on-floatarith a* ∧ *continuous-on-floatarith b*) |
continuous-on-floatarith (*Minus a*) = *continuous-on-floatarith a* |
continuous-on-floatarith (*Inverse a*) = *False* |
continuous-on-floatarith (*Cos a*) = *continuous-on-floatarith a* |
continuous-on-floatarith (*Arctan a*) = *continuous-on-floatarith a* |
continuous-on-floatarith (*Min a b*) = (*continuous-on-floatarith a* ∧ *continuous-on-floatarith b*) |
continuous-on-floatarith (*Max a b*) = (*continuous-on-floatarith a* ∧ *continuous-on-floatarith b*) |
continuous-on-floatarith (*Abs a*) = *continuous-on-floatarith a* |
continuous-on-floatarith Pi = *True* |
continuous-on-floatarith (*Sqrt a*) = *False* |
continuous-on-floatarith (*Exp a*) = *continuous-on-floatarith a* |
continuous-on-floatarith (*Powr a b*) = *False* |
continuous-on-floatarith (*Ln a*) = *False* |
continuous-on-floatarith (*Floor a*) = *False* |
continuous-on-floatarith (*Power a n*) = (*if n = 0 then True else continuous-on-floatarith a*) |
continuous-on-floatarith (*Num f*) = *True* |
continuous-on-floatarith (*Var n*) = *True*

definition $Maxs_e$ *xs* = *fold* (λ*a b*. *floatarith.Max a b*) *xs*
definition $norm2_e$ *n* = $Maxs_e$ (*map* (λ*j*. *Norm* (*map* (λ*i*. *Var* (*Suc j* ∗ *n* + *i*)) [*0*..<*n*])) [*0*..<*n*]) (*Num 0*)

definition N_r *l* = *Num* (*float-of l*)

lemma *interpret-floatarith-Norm*:
interpret-floatarith (*Norm xs*) *vs* = *L2-set* (λ*i*. *interpret-floatarith* (*xs* ! *i*) *vs*) {*0*..<*length xs*}
⟨*proof*⟩

lemma *interpret-floatarith-Nr*[*simp*]: *interpret-floatarith* (N_r *U*) *vs* = *real-of-float* (*float-of U*)
⟨*proof*⟩

fun *list-updates* **where**
list-updates [] - *xs* = *xs*
| *list-updates* - [] *xs* = *xs*
| *list-updates* (*i*#*is*) (*u*#*us*) *xs* = *list-updates is us* (*xs*[*i*:=*u*])

lemma *list-updates-nth-notmem*:
assumes *length xs* = *length ys*
assumes *i* ∉ *set xs*

shows *list-updates xs ys vs* ! *i* = *vs* ! *i*
⟨*proof*⟩

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* ∈ *set xs then ys* ! (*index xs i*) *else vs* ! *i*)
⟨*proof*⟩

lemma *length-list-updates*[*simp*]: *length* (*list-updates xs ys vs*) = *length vs*
⟨*proof*⟩

lemma *list-updates-nth-ge*[*simp*]:
x ≥ *length vs* ⟹ *length xs* = *length ys* ⟹ *list-updates xs ys vs* ! *x* = *vs* ! *x*
⟨*proof*⟩

lemma
list-updates-nth:
assumes [*simp*]: *length xs* = *length ys distinct xs*
shows *list-updates xs ys vs* ! *i* = (*if i* < *length vs* ∧ *i* ∈ *set xs then ys* ! *index xs i else vs* ! *i*)
⟨*proof*⟩

lemma *list-of-eucl-coord-update*:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*)
assumes [*simp*]: *distinct xs*
assumes [*simp*]: *i* ∈ *Basis*
assumes [*simp*]: ⋀*n*. *n* ∈ *set xs* ⟹ *n* < *length vs*
shows *list-updates xs* (*list-of-eucl* (*x* + (*p* − *x* ∙ *i*) ∗$_R$ *i*::'*a*)) *vs* =
(*list-updates xs* (*list-of-eucl x*) *vs*)[*xs* ! *index Basis-list i* := *p*]
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nth-nth-eucl-of-env-inner*:
b ∈ *Basis* ⟹ *length is* = *DIM*('*a*) ⟹ *vs* ! (*is* ! *index Basis-list b*) = *eucl-of-env is vs* ∙ *b*
for *b*::'*a*::*executable-euclidean-space*
⟨*proof*⟩

lemma *list-updates-idem*[*simp*]:
assumes (⋀*i*. *i* ∈ *set X0* ⟹ *i* < *length vs*)
shows (*list-updates X0* (*map* ((!) *vs*) *X0*) *vs*) = *vs*
⟨*proof*⟩

lemma *pairwise-orthogonal-Basis*[*intro*, *simp*]: *pairwise orthogonal Basis*
⟨*proof*⟩

primrec *freshs-floatarith* **where**
freshs-floatarith (*Var y*) *x* ⟷ (*y* ∉ *set x*)
| *freshs-floatarith* (*Num a*) *x* ⟷ *True*
| *freshs-floatarith Pi x* ⟷ *True*
| *freshs-floatarith* (*Cos a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Abs a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Arctan a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Sqrt a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Exp a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Floor a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Power a n*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Minus a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Ln a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Inverse a*) *x* ⟷ *freshs-floatarith a x*
| *freshs-floatarith* (*Add a b*) *x* ⟷ *freshs-floatarith a x* ∧ *freshs-floatarith b x*
| *freshs-floatarith* (*Mult a b*) *x* ⟷ *freshs-floatarith a x* ∧ *freshs-floatarith b x*
| *freshs-floatarith* (*floatarith.Max a b*) *x* ⟷ *freshs-floatarith a x* ∧ *freshs-floatarith b x*
| *freshs-floatarith* (*floatarith.Min a b*) *x* ⟷ *freshs-floatarith a x* ∧ *freshs-floatarith b x*
| *freshs-floatarith* (*Powr a b*) *x* ⟷ *freshs-floatarith a x* ∧ *freshs-floatarith b x*

lemma *freshs-floatarith*[*simp*]:
assumes *freshs-floatarith fa ds length ds* = *length xs*
shows *interpret-floatarith fa* (*list-updates ds xs vs*) = *interpret-floatarith fa vs*
⟨*proof*⟩

lemma *freshs-floatarith-max-Var-floatarithI*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *max-Var-floatarith f* ≤ *x*
shows *freshs-floatarith f xs*
⟨*proof*⟩

definition *freshs-floatariths fas xs* = (∀ *fa*∈*set fas*. *freshs-floatarith fa xs*)

lemma *freshs-floatariths-max-Var-floatarithsI*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *max-Var-floatariths f* ≤ *x*
shows *freshs-floatariths f xs*
⟨*proof*⟩

lemma *freshs-floatariths-freshs-floatarithI*:
assumes ⋀*fa*. *fa* ∈ *set fas* ⟹ *freshs-floatarith fa xs*
shows *freshs-floatariths fas xs*
⟨*proof*⟩

lemma *fresh-floatariths-fresh-floatarithI*:
assumes *freshs-floatariths fas xs*
assumes *fa* ∈ *set fas*
shows *freshs-floatarith fa xs*
⟨*proof*⟩

lemma *fresh-floatariths-fresh-floatarith*[*simp*]:
fresh-floatariths (*fas*) *i* ⟹ *fa* ∈ *set fas* ⟹ *fresh-floatarith fa i*
⟨*proof*⟩

lemma *interpret-floatariths-fresh-cong*:
assumes ⋀*i*. ¬*fresh-floatariths f i* ⟹ *xs* ! *i* = *ys* ! *i*
shows *interpret-floatariths f ys* = *interpret-floatariths f xs*
⟨*proof*⟩

fun *subterms* :: *floatarith* ⇒ *floatarith set* **where**
subterms (*Add a b*) = *insert* (*Add a b*) (*subterms a* ∪ *subterms b*) |
subterms (*Mult a b*) = *insert* (*Mult a b*) (*subterms a* ∪ *subterms b*) |
subterms (*Min a b*) = *insert* (*Min a b*) (*subterms a* ∪ *subterms b*) |
subterms (*floatarith.Max a b*) = *insert* (*floatarith.Max a b*) (*subterms a* ∪ *subterms b*) |
subterms (*Powr a b*) = *insert* (*Powr a b*) (*subterms a* ∪ *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* ∈ *subterms fa2*
⟨*proof*⟩

lemma *interpret-floatarith-FDERIV-floatarith-eucl-of-env*:— TODO: cleanup, reduce to DERIV?!
assumes *iD*: ⋀*i*. *i* < *DIM*('*a*) ⟹ *isDERIV* (*xs* ! *i*) *fa vs*
assumes *ds-fresh*: *freshs-floatarith fa ds*
assumes [*simp*]: *length xs* = *DIM* ('*a*) *length ds* = *DIM* ('*a*)
⋀*i*. *i* ∈ *set xs* ⟹ *i* < *length vs distinct xs*
⋀*i*. *i* ∈ *set ds* ⟹ *i* < *length vs distinct ds*
shows ((λ*x*::'*a*::*executable-euclidean-space*.
(*interpret-floatarith fa* (*list-updates xs* (*list-of-eucl x*) *vs*))) *has-derivative*
(λ*d*. *interpret-floatarith* (*FDERIV-floatarith fa xs* (*map Var ds*)) (*list-updates*

ds (*list-of-eucl d*) *vs*))
) (*at* (*eucl-of-env xs vs*))
⟨*proof*⟩

lemma *interpret-floatarith-FDERIV-floatarith-append*:
assumes *iD*: ⋀*i j*. *i* < *DIM*('*a*) ⟹ *isDERIV i* (*fa*) (*list-of-eucl x* @ *params*)
assumes *m*: *max-Var-floatarith fa* ≤ *DIM*('*a*) + *length params*
shows ((λ*x*::'*a*::*executable-euclidean-space*.
interpret-floatarith fa (*list-of-eucl x* @ *params*)) *has-derivative*
(λ*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*⟩

lemma *interpret-floatarith-FDERIV-floatarith*:
assumes *iD*: ⋀*i j*. *i* < *DIM*('*a*) ⟹ *isDERIV i* (*fa*) (*list-of-eucl x*)
assumes *m*: *max-Var-floatarith fa* ≤ *DIM*('*a*)
shows ((λ*x*::'*a*::*executable-euclidean-space*.
interpret-floatarith fa (*list-of-eucl x*)) *has-derivative*
(λ*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*)
⟨*proof*⟩

lemma *interpret-floatarith-eventually-isDERIV*:
assumes *iD*: ⋀*i j*. *i* < *DIM*('*a*) ⟹ *isDERIV i fa* (*list-of-eucl x* @ *params*)
assumes *m*: *max-Var-floatarith fa* ≤ *DIM*('*a*::*executable-euclidean-space*) + *length params*
shows ∀ *i* < *DIM*('*a*). ∀ *F* (*x*::'*a*) *in at x*. *isDERIV i fa* (*list-of-eucl x* @ *params*)
⟨*proof*⟩

lemma *eventually-isFDERIV*:
assumes *iD*: *isFDERIV DIM*('*a*) [*0*..<*DIM*('*a*)] *fas* (*list-of-eucl x*@*params*)
assumes *m*: *max-Var-floatariths fas* ≤ *DIM*('*a*::*executable-euclidean-space*) + *length params*
shows ∀ *F* (*x*::'*a*) *in at x*. *isFDERIV DIM*('*a*) [*0*..<*DIM*('*a*)] *fas* (*list-of-eucl x* @ *params*)
⟨*proof*⟩

lemma *isFDERIV-eventually-isFDERIV*:
assumes *iD*: *isFDERIV DIM*('*a*) [*0*..<*DIM*('*a*)] *fas* (*list-of-eucl x*@*params*)
assumes *m*: *max-Var-floatariths fas* ≤ *DIM*('*a*::*executable-euclidean-space*) + *length params*
shows ∀ *F* (*x*::'*a*) *in at x*. *isFDERIV DIM*('*a*) [*0*..<*DIM*('*a*)] *fas* (*list-of-eucl x* @ *params*)
⟨*proof*⟩

lemma *interpret-floatarith-FDERIV-floatariths-eucl-of-env*:

assumes *iD*: *isFDERIV DIM('a) xs fas vs*
assumes *fresh*: *freshs-floatariths (fas) ds*
assumes [*simp*]: *length ds = DIM ('a)*
⋀*i*. *i ∈ set xs ⟹ i < length vs distinct xs*
⋀*i*. *i ∈ set ds ⟹ i < length vs distinct ds*
shows ((λ*x*::'*a*::*executable-euclidean-space*.
eucl-of-list
(*interpret-floatariths fas* (*list-updates xs* (*list-of-eucl x*) *vs*))::'*a*) *has-derivative*
(λ*d*. *eucl-of-list* (*interpret-floatariths*
(*FDERIV-floatariths fas xs* (*map Var ds*))
(*list-updates ds* (*list-of-eucl d*) *vs*)))) (*at* (*eucl-of-env xs vs*))
⟨*proof*⟩

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 ≤ DIM('a) + length ramsch*
assumes [*simp*]: *length fas = DIM('a)*
shows ((λ*x*::'*a*::*executable-euclidean-space*.
eucl-of-list
(*interpret-floatariths fas* (*list-of-eucl x*@*ramsch*))::'*a*) *has-derivative*
(λ*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*⟩

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 ((λ*x*::'*a*::*executable-euclidean-space*.
eucl-of-list
(*interpret-floatariths fas* (*list-of-eucl x*))::'*a*) *has-derivative*
(λ*d*. *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*)
⟨*proof*⟩

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* (λ*x*. *min* (*f x*) (*g x*))
⟨*proof*⟩

lemmas [*continuous-intros*] = *continuous-on-max*
lemma *continuous-on-if-const*[*continuous-intros*]:
continuous-on s f ⟹ continuous-on s g ⟹ continuous-on s (λ*x*. *if p then f x else g x*)
⟨*proof*⟩

lemma *continuous-on-floatarith*:
assumes *continuous-on-floatarith fa length xs = DIM('a) distinct xs*
shows *continuous-on UNIV* (λ*x*. *interpret-floatarith fa* (*list-updates xs* (*list-of-eucl* (*x*::*'a*::*executable-euclidean-space*)) *vs*))
⟨*proof*⟩

fun *open-form* :: *form* ⇒ *bool* **where**
open-form (*Bound x a b f*) = *False* |
open-form (*Assign x a f*) = *False* |
open-form (*Less a b*) ⟷ *continuous-on-floatarith a* ∧ *continuous-on-floatarith b* |
open-form (*LessEqual a b*) = *False* |
open-form (*AtLeastAtMost x a b*) = *False* |
open-form (*Conj f g*) ⟷ *open-form f* ∧ *open-form g* |
open-form (*Disj f g*) ⟷ *open-form f* ∧ *open-form g*

lemma *open-form*:
assumes *open-form f length xs = DIM('a*::*executable-euclidean-space*) *distinct xs*
shows *open* (*Collect* (λ*x*::*'a*. *interpret-form f* (*list-updates xs* (*list-of-eucl x*) *vs*)))
⟨*proof*⟩

primrec *isnFDERIV* **where**
isnFDERIV N fas xs ds vs 0 = *True*
| *isnFDERIV N fas xs ds vs* (*Suc n*) ⟷
isFDERIV N xs (*FDERIV-n-floatariths fas xs* (*map Var ds*) *n*) *vs* ∧
isnFDERIV N fas xs ds vs n

lemma *one-add-square-eq-0*: *1* + (*x*)^2 ≠ (*0*::*real*)
⟨*proof*⟩

lemma *isDERIV-fold-const-fa*[*intro*]:
assumes *isDERIV x fa vs*
shows *isDERIV x* (*fold-const-fa fa*) *vs*
⟨*proof*⟩

lemma *isDERIV-fold-const-fa-minus*[*intro*!]:
assumes *isDERIV x* (*fold-const-fa fa*) *vs*
shows *isDERIV x* (*fold-const-fa* (*Minus fa*)) *vs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isDERIV-fold-const-fa-power*[*intro*!]:
assumes *isDERIV x* (*fold-const-fa fa*) *vs*
shows *isDERIV x* (*fold-const-fa* (*fa* $\widehat{}_e$ *n*)) *vs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-square-ne-zero*[*simp*]: (*y*::'*a*::*linordered-idom*) > *0* $\Longrightarrow$ $y + x^2 \neq 0$
⟨*proof*⟩

lemma *isDERIV-FDERIV-floatarith*:
assumes *isDERIV x fa vs* $\bigwedge i$. *i* < *length ds* $\Longrightarrow$ *isDERIV x* (*ds* ! *i*) *vs*
assumes [*simp*]: *length xs* = *length ds*
shows *isDERIV x* (*FDERIV-floatarith fa xs ds*) *vs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isFDERIV-imp-isFDERIV-FDERIV-n*:
assumes *length fas* = *length ds*
shows *isFDERIV N xs fas vs* $\Longrightarrow$ *isFDERIV N xs ds vs* $\Longrightarrow$
isFDERIV N xs (*FDERIV-n-floatariths fas xs ds n*) *vs*
⟨*proof*⟩

lemma *isFDERIV-map-Var*:
assumes [*simp*]: *length ds* = *N length xs* = *N*
shows *isFDERIV N xs* (*map Var ds*) *vs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: *max-Var-floatariths fas* ≤ *2* ∗ *DIM*('*a*::*executable-euclidean-space*)
shows ∀ *F* (*x*::'*a*) *in at x*. *isnFDERIV DIM*('*a*) *fas* [*0*..<*DIM*('*a*)] [*DIM*('*a*)..<*2*∗*DIM*('*a*)] (*list-of-eucl x* @ *list-of-eucl d*) *n*
⟨*proof*⟩

lemma *isFDERIV-open*:
assumes *max-Var-floatariths fas* ≤ *DIM*('*a*)
shows *open* {*x*::'*a*. *isFDERIV DIM*('*a*::*executable-euclidean-space*) [*0*..<*DIM*('*a*)] *fas* (*list-of-eucl x*)}
(**is** *open* (*Collect ?s*))
⟨*proof*⟩

lemma *interpret-floatarith-FDERIV-floatarith-eq*:
assumes [*simp*]: *length xs* = *DIM*('*a*::*executable-euclidean-space*) *length ds* = *DIM*('*a*)
shows *interpret-floatarith* (*FDERIV-floatarith fa xs ds*) *vs* =
einterpret (*map* (λ*x*. *DERIV-floatarith x fa*) *xs*) *vs* · (*einterpret ds vs*::'*a*)
⟨*proof*⟩

lemma
interpret-floatariths-FDERIV-floatariths-cong:
assumes [*simp*]: *length d1s* = *DIM*('*a*::*executable-euclidean-space*) *length d2s* = *DIM*('*a*) *length fas1* = *length fas2*
assumes *fresh1*: *freshs-floatariths fas1 d1s*
assumes *fresh2*: *freshs-floatariths fas2 d2s*
assumes *eq1*: ⋀*i*. *i* < *length fas1* ⟹ *interpret-floatariths* (*map* (λ*x*. *DERIV-floatarith x* (*fas1* ! *i*)) [*0*..<*DIM*('*a*)]) *xs1* =
interpret-floatariths (*map* (λ*x*. *DERIV-floatarith x* (*fas2* ! *i*)) [*0*..<*DIM*('*a*)]) *xs2*
assumes *eq2*: ⋀*i*. *i* < *DIM*('*a*) ⟹ *xs1* ! (*d1s* ! *i*) = *xs2* ! (*d2s* ! *i*)
shows *interpret-floatariths* (*FDERIV-floatariths fas1* [*0*..<*DIM*('*a*)] (*map floatarith.Var d1s*)) *xs1* =
interpret-floatariths (*FDERIV-floatariths fas2* [*0*..<*DIM*('*a*)] (*map floatarith.Var d2s*)) *xs2*
⟨*proof*⟩

lemma *subst-floatarith-Var-DERIV-floatarith*:
assumes ⋀*x*. *x* = *n* ⟷ *s x* = *n*
shows *subst-floatarith* (λ*x*. *Var* (*s x*)) (*DERIV-floatarith n fa*) =
DERIV-floatarith n (*subst-floatarith* (λ*x*. *Var* (*s x*)) *fa*)
⟨*proof*⟩

lemma *subst-floatarith-inner-floatariths*[*simp*]:
assumes *length fs* = *length gs*
shows *subst-floatarith s* (*inner-floatariths fs gs*) =
inner-floatariths (*map* (*subst-floatarith s*) *fs*) (*map* (*subst-floatarith s*) *gs*)
⟨*proof*⟩

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* (λ*x*. *Var* (*s x*)) *fa* ⟷ *fa* = *Num x*
⟨*proof*⟩

lemma *Add-eq-subst-VarE*:
assumes *Add fa1 fa2* = *subst-floatarith* (λ*x*. *Var* (*s x*)) *fa*
obtains *a1 a2* **where** *fa* = *Add a1 a2 fa1* = *subst-floatarith* (λ*x*. *Var* (*s x*)) *a1*
fa2 = *subst-floatarith* (λ*x*. *Var* (*s x*)) *a2*
⟨*proof*⟩

lemma *subst-floatarith-eq-self*[*simp*]: *subst-floatarith s f* = *f* **if** *max-Var-floatarith f* = *0*
⟨*proof*⟩

lemma *fold-const-fa-unique*: *False* **if** (⋀*x*. *f* = *Num x*)
⟨*proof*⟩

lemma *zero-unique*: *False* **if** (⋀*x*::*float*. *x* = *0*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subst-floatarith-Var-fold-const-fa*[*symmetric*]:
fold-const-fa (*subst-floatarith* (λ*x*. *Var* (*s x*)) *fa*) =
subst-floatarith (λ*x*. *Var* (*s x*)) (*fold-const-fa fa*)
⟨*proof*⟩

lemma *subst-floatarith-eq-Num*[*simp*]: (*subst-floatarith* (λ*x*. *Var* (*s x*)) *fa* = *Num x*) ⟷ *fa* = *Num x*
⟨*proof*⟩

lemma *fold-const-fa-subst-eq-Num0-iff*[*simp*]:
fold-const-fa (*subst-floatarith* (λ*x*. *Var* (*s x*)) *fa*) = *Num x* ⟷ *fold-const-fa fa* = *Num x*
⟨*proof*⟩

lemma *subst-floatarith-Var-FDERIV-floatarith*:
assumes *len*: *length xs* = *DIM*('*a*::*executable-euclidean-space*) **and** [*simp*]: *length ds* = *DIM*('*a*)
assumes *eq*: $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow (y = x) = (s\ y = x)$
shows *subst-floatarith* ($\lambda x.\ Var\ (s\ x)$) (*FDERIV-floatarith fa xs ds*) =
(*FDERIV-floatarith* (*subst-floatarith* ($\lambda x.\ Var\ (s\ x)$) *fa*) *xs* (*map* (*subst-floatarith* ($\lambda x.\ Var\ (s\ x)$)) *ds*))
⟨*proof*⟩

lemma *subst-floatarith-Var-FDERIV-n-nth*:
assumes *len*: *length xs* = *DIM*('*a*::*executable-euclidean-space*) **and** [*simp*]: *length ds* = *DIM*('*a*)
assumes *eq*: $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow (y = x) = (s\ y = x)$
assumes [*simp*]: $i < length\ fas$
shows *subst-floatarith* ($\lambda x.\ Var\ (s\ x)$) (*FDERIV-n-floatariths fas xs ds n* ! *i*) =
(*FDERIV-n-floatariths* (*map* (*subst-floatarith* ($\lambda x.\ Var\ (s\ x)$)) *fas*) *xs* (*map* (*subst-floatarith* ($\lambda x.\ Var\ (s\ x)$)) *ds*) *n* ! *i*)
⟨*proof*⟩

lemma *subst-floatarith-Var-max-Var-floatarith*:
assumes $\bigwedge i.\ i < max\text{-}Var\text{-}floatarith\ fa \Longrightarrow s\ i = i$
shows *subst-floatarith* ($\lambda i.\ Var\ (s\ i)$) *fa* = *fa*
⟨*proof*⟩

lemma *interpret-floatarith-subst-floatarith-idem*:
assumes *mv*: *max-Var-floatarith fa* ≤ *length vs*
assumes *idem*: $\bigwedge j.\ j < max\text{-}Var\text{-}floatarith\ fa \Longrightarrow vs\ !\ s\ j = vs\ !\ j$
shows *interpret-floatarith* (*subst-floatarith* ($\lambda i.\ Var\ (s\ i)$) *fa*) *vs* = *interpret-floatarith fa vs*
⟨*proof*⟩

lemma *isDERIV-subst-Var-floatarith*:
assumes *mv*: *max-Var-floatarith fa* ≤ *length vs*
assumes *idem*: $\bigwedge j.\ j < max\text{-}Var\text{-}floatarith\ fa \Longrightarrow vs\ !\ s\ j = vs\ !\ j$
assumes $\bigwedge j.\ s\ j = i \longleftrightarrow j = i$
shows *isDERIV i* (*subst-floatarith* ($\lambda i.\ Var\ (s\ i)$) *fa*) *vs* = *isDERIV i fa vs*
⟨*proof*⟩

lemma *isFDERIV-subst-Var-floatarith*:
assumes *mv*: *max-Var-floatariths fas* ≤ *length vs*
assumes *idem*: $\bigwedge j.\ j < max\text{-}Var\text{-}floatariths\ fas \Longrightarrow vs\ !\ (s\ j) = vs\ !\ j$
assumes $\bigwedge i\ j.\ i \in set\ xs \Longrightarrow s\ j = i \longleftrightarrow j = i$
shows *isFDERIV n xs* (*map* (*subst-floatarith* ($\lambda i.\ Var\ (s\ i)$)) *fas*) *vs* = *isFDERIV n xs fas vs*
⟨*proof*⟩

lemma *interpret-floatariths-append*[*simp*]:
interpret-floatariths (*xs* @ *ys*) *vs* = *interpret-floatariths xs vs* @ *interpret-floatariths ys vs*

⟨*proof*⟩

lemma *not-fresh-inner-floatariths*:
assumes *length xs* = *length ys*
shows ¬ *fresh-floatarith* (*inner-floatariths xs ys*) *i* ⟷ ¬*fresh-floatariths xs i* ∨ ¬*fresh-floatariths ys i*
⟨*proof*⟩

lemma *fresh-inner-floatariths*:
assumes *length xs* = *length ys*
shows *fresh-floatarith* (*inner-floatariths xs ys*) *i* ⟷ *fresh-floatariths xs i* ∧ *fresh-floatariths ys i*
⟨*proof*⟩

lemma *not-fresh-floatariths-map*:
¬ *fresh-floatariths* (*map f xs*) *i* ⟷ (∃ *x* ∈ *set xs*. ¬*fresh-floatarith* (*f x*) *i*)
⟨*proof*⟩

lemma *fresh-floatariths-map*:
fresh-floatariths (*map f xs*) *i* ⟷ (∀ *x* ∈ *set xs*. *fresh-floatarith* (*f x*) *i*)
⟨*proof*⟩

lemma *fresh-floatarith-fold-const-fa*: *fresh-floatarith fa i* ⟹ *fresh-floatarith* (*fold-const-fa fa*) *i*
⟨*proof*⟩

lemma *fresh-floatarith-fold-const-fa-Add*[*intro*!]:
assumes *fresh-floatarith* (*fold-const-fa a*) *i fresh-floatarith* (*fold-const-fa b*) *i*
shows *fresh-floatarith* (*fold-const-fa* (*Add a b*)) *i*
⟨*proof*⟩

lemma *fresh-floatarith-fold-const-fa-Mult*[*intro*!]:
assumes *fresh-floatarith* (*fold-const-fa a*) *i fresh-floatarith* (*fold-const-fa b*) *i*
shows *fresh-floatarith* (*fold-const-fa* (*Mult a b*)) *i*
⟨*proof*⟩

lemma *fresh-floatarith-fold-const-fa-Minus*[*intro*!]:
assumes *fresh-floatarith* (*fold-const-fa b*) *i*
shows *fresh-floatarith* (*fold-const-fa* (*Minus b*)) *i*
⟨*proof*⟩

lemma *fresh-FDERIV-floatarith*:
fresh-floatarith ode-e i ⟹ *fresh-floatariths ds i*
⟹ *length ds* = *DIM*('*a*)
⟹ *fresh-floatarith* (*FDERIV-floatarith ode-e* [*0*..<*DIM*('*a*::*executable-euclidean-space*)] *ds*) *i*
⟨*proof*⟩

lemma *not-fresh-FDERIV-floatarith*:

$\neg$ *fresh-floatarith* (*FDERIV-floatarith ode-e* [*0..<DIM*('*a*::*executable-euclidean-space*)] *ds*) *i*
$\Longrightarrow$ *length ds* = *DIM*('*a*)
$\Longrightarrow$ $\neg$*fresh-floatarith ode-e i* $\vee$ $\neg$*fresh-floatariths ds i*
⟨*proof*⟩

lemma *not-fresh-FDERIV-floatariths*:
$\neg$ *fresh-floatariths* (*FDERIV-floatariths ode-e* [*0..<DIM*('*a*::*executable-euclidean-space*)] *ds*) *i* $\Longrightarrow$
length ds = *DIM*('*a*) $\Longrightarrow$ $\neg$*fresh-floatariths ode-e i* $\vee$ $\neg$*fresh-floatariths ds i*
⟨*proof*⟩

lemma *isDERIV-FDERIV-floatarith-linear*:
fixes *x h*::'*a*::*executable-euclidean-space*
assumes $\bigwedge$*k*. *k* < *DIM*('*a*) $\Longrightarrow$ *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*)
⟨*proof*⟩

lemma
isFDERIV-FDERIV-floatariths-linear:
fixes *x h*::'*a*::*executable-euclidean-space*
assumes $\bigwedge$*i j k*.
i < *DIM*('*a*) $\Longrightarrow$
j < *DIM*('*a*) $\Longrightarrow$ *k* < *DIM*('*a*) $\Longrightarrow$ *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*)
⟨*proof*⟩

definition *isFDERIV-approx* **where**
isFDERIV-approx p n xs fas vs =
((∀ *i*<*n*. ∀ *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 $\Longrightarrow$ *isFDERIV-approx prec n xs fas VS* $\Longrightarrow$ *isFDERIV n xs fas vs*
⟨*proof*⟩

primrec *isnFDERIV-approx* **where**

isnFDERIV-approx p N fas xs ds vs 0 = True
| *isnFDERIV-approx p N fas xs ds vs (Suc n) ⟷*
isFDERIV-approx p N xs (FDERIV-n-floatariths fas xs (map Var ds) n) vs ∧
isnFDERIV-approx p N fas xs ds vs n

lemma *isnFDERIV-approx*:
bounded-by vs VS ⟹ isnFDERIV-approx prec N fas xs ds VS n ⟹ isnFDERIV N fas xs ds vs n
⟨*proof*⟩

fun *plain-floatarith*::*nat ⇒ floatarith ⇒ bool* **where**
plain-floatarith N (floatarith.Add a b) ⟷ plain-floatarith N a ∧ plain-floatarith N b
| *plain-floatarith N (floatarith.Mult a b) ⟷ plain-floatarith N a ∧ plain-floatarith N b*
| *plain-floatarith N (floatarith.Minus a) ⟷ plain-floatarith N a*
| *plain-floatarith N (floatarith.Pi) ⟷ True*
| *plain-floatarith N (floatarith.Num n) ⟷ True*
| *plain-floatarith N (floatarith.Var i) ⟷ i < N*
| *plain-floatarith N (floatarith.Max a b) ⟷ plain-floatarith N a ∧ plain-floatarith N b*
| *plain-floatarith N (floatarith.Min a b) ⟷ plain-floatarith N a ∧ plain-floatarith N b*
| *plain-floatarith N (floatarith.Power a n) ⟷ plain-floatarith N a*
| *plain-floatarith N (floatarith.Cos a) ⟷ False* — TODO: should be plain!
| *plain-floatarith N (floatarith.Arctan a) ⟷ False* — TODO: should be plain!
| *plain-floatarith N (floatarith.Abs a) ⟷ plain-floatarith N a*
| *plain-floatarith N (floatarith.Exp a) ⟷ False* — TODO: should be plain!
| *plain-floatarith N (floatarith.Sqrt a) ⟷ False* — TODO: should be plain!
| *plain-floatarith N (floatarith.Floor a) ⟷ plain-floatarith N a*

| *plain-floatarith N (floatarith.Powr a b) ⟷ False*
| *plain-floatarith N (floatarith.Inverse a) ⟷ False*
| *plain-floatarith N (floatarith.Ln a) ⟷ False*

lemma *plain-floatarith-approx-not-None*:
assumes *plain-floatarith N fa N ≤ length XS* ⋀*i. i < N ⟹ XS ! i ≠ None*
shows *approx p fa XS ≠ None*
⟨*proof*⟩

definition *Rad-of w = w ∗ (Pi / Num 180)*
lemma *interpret-Rad-of*[*simp*]: *interpret-floatarith (Rad-of w) xs = rad-of (interpret-floatarith w xs)*
⟨*proof*⟩

definition *Deg-of w = Num 180 ∗ w / Pi*
lemma *interpret-Deg-of*[*simp*]: *interpret-floatarith (Deg-of w) xs = deg-of (interpret-floatarith w xs)*

⟨*proof*⟩

unbundle *no-floatarith-notation*

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-notation*

derive (*linorder*) *compare-order float*

derive *linorder floatarith*

4.1 Definition

type-synonym *slp* = *floatarith list*

primrec *interpret-slp*::*slp* ⇒ *real list* ⇒ *real list* **where**
interpret-slp [] = (λ*xs*. *xs*)
| *interpret-slp* (*ea* # *eas*) = (λ*xs*. *interpret-slp eas* (*interpret-floatarith ea xs*#*xs*))

4.2 Reification as straight line program (with common subexpression elimination)

definition *slp-index vs i* = (*length vs* − *Suc i*)

definition *slp-index-lookup vs M a* = *slp-index vs* (*the* (*Mapping.lookup M a*))

definition
slp-of-fa-bin Binop a b M slp M2 slp2 =
(*case Mapping.lookup M* (*Binop a b*) *of*
Some i ⇒ (*Mapping.update* (*Binop a b*) (*length slp*) *M*, *slp*@[*Var* (*slp-index slp i*)])
| *None* ⇒ (*Mapping.update* (*Binop a b*) (*length slp2*) *M2*,
slp2@[*Binop* (*Var* (*slp-index-lookup slp2 M2 a*)) (*Var* (*slp-index-lookup slp2 M2 b*))]))

definition
slp-of-fa-un Unop a M slp M1 slp1 =
(*case Mapping.lookup M* (*Unop a*) *of*

Some i ⇒ (*Mapping.update* (*Unop a*) (*length slp*) *M*, *slp*@[*Var* (*slp-index slp i*)])
| *None* ⇒ (*Mapping.update* (*Unop a*) (*length slp1*) *M1*,
slp1@[*Unop* (*Var* (*slp-index-lookup slp1 M1 a*))]))

definition
slp-of-fa-cnst Const Const′ M vs =
(*Mapping.update Const* (*length vs*) *M*,
vs @ [*case Mapping.lookup M Const of Some i* ⇒ *Var* (*slp-index vs i*) | *None* ⇒ *Const′*])

fun *slp-of-fa* :: *floatarith* ⇒ (*floatarith*, *nat*) *mapping* ⇒ *floatarith list* ⇒
((*floatarith*, *nat*) *mapping* × *floatarith list*) **where**
slp-of-fa (*Add a b*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp*; (*M2*, *slp2*) = *slp-of-fa b M1 slp1 in*
slp-of-fa-bin Add a b M slp M2 slp2)
| *slp-of-fa* (*Mult a b*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp*; (*M2*, *slp2*) = *slp-of-fa b M1 slp1 in*
slp-of-fa-bin Mult a b M slp M2 slp2)
| *slp-of-fa* (*Min a b*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp*; (*M2*, *slp2*) = *slp-of-fa b M1 slp1 in*
slp-of-fa-bin Min a b M slp M2 slp2)
| *slp-of-fa* (*Max a b*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp*; (*M2*, *slp2*) = *slp-of-fa b M1 slp1 in*
slp-of-fa-bin Max a b M slp M2 slp2)
| *slp-of-fa* (*Powr a b*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp*; (*M2*, *slp2*) = *slp-of-fa b M1 slp1 in*
slp-of-fa-bin Powr a b M slp M2 slp2)
| *slp-of-fa* (*Inverse a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Inverse a M slp M1 slp1*)
| *slp-of-fa* (*Cos a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Cos a M slp M1 slp1*)
| *slp-of-fa* (*Arctan a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Arctan a M slp M1 slp1*)
| *slp-of-fa* (*Abs a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Abs a M slp M1 slp1*)
| *slp-of-fa* (*Sqrt a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Sqrt a M slp M1 slp1*)
| *slp-of-fa* (*Exp a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Exp a M slp M1 slp1*)
| *slp-of-fa* (*Ln a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Ln a M slp M1 slp1*)
| *slp-of-fa* (*Minus a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Minus a M slp M1 slp1*)
| *slp-of-fa* (*Floor a*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un Floor a M slp M1 slp1*)
| *slp-of-fa* (*Power a n*) *M slp* =
(*let* (*M1*, *slp1*) = *slp-of-fa a M slp in slp-of-fa-un* (λ*a. Power a n*) *a M slp M1 slp1*)

| *slp-of-fa Pi M slp = slp-of-fa-cnst Pi Pi M slp*
| *slp-of-fa (Var v) M slp = slp-of-fa-cnst (Var v) (Var (v + length slp)) M slp*
| *slp-of-fa (Num n) M slp = slp-of-fa-cnst (Num n) (Num n) M slp*

lemma *interpret-slp-snoc*[*simp*]:
interpret-slp (slp @ [fa]) xs = interpret-floatarith fa (interpret-slp slp xs)#interpret-slp slp xs
⟨*proof*⟩

lemma
binop-slp-of-fa-induction-step:
assumes
Binop-IH1:
⋀*M slp M' slp'*. *slp-of-fa fa1 M slp = (M', slp')* ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *subterms f* ⊆ *Mapping.keys M*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *the* (*Mapping.lookup M f*) < *length slp*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *interpret-slp slp xs* ! *slp-index-lookup slp M f = interpret-floatarith f xs*) ⟹
subterms fa1 ⊆ *Mapping.keys M'* ∧
Mapping.keys M ⊆ *Mapping.keys M'* ∧
(∀ *f*∈*Mapping.keys M'*. *subterms f* ⊆ *Mapping.keys M'* ∧
the (*Mapping.lookup M' f*) < *length slp'* ∧
interpret-slp slp' xs ! *slp-index-lookup slp' M' f = interpret-floatarith f xs*)
and
Binop-IH2:
⋀*M slp M' slp'*. *slp-of-fa fa2 M slp = (M', slp')* ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *subterms f* ⊆ *Mapping.keys M*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *the* (*Mapping.lookup M f*) < *length slp*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *interpret-slp slp xs* ! *slp-index-lookup slp M f = interpret-floatarith f xs*) ⟹
subterms fa2 ⊆ *Mapping.keys M'* ∧
Mapping.keys M ⊆ *Mapping.keys M'* ∧
(∀ *f*∈*Mapping.keys M'*. *subterms f* ⊆ *Mapping.keys M'* ∧
the (*Mapping.lookup M' f*) < *length slp'* ∧
interpret-slp slp' xs ! *slp-index-lookup slp' M' f = interpret-floatarith f xs*)
and *Binop-prems*:
(*case slp-of-fa fa1 M slp of*
(*M1*, *slp1*) ⇒
case slp-of-fa fa2 M1 slp1 of (*x*, *xa*) ⇒ *slp-of-fa-bin Binop fa1 fa2 M slp x xa*) = (*M'*, *slp'*)
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *subterms f* ⊆ *Mapping.keys M*
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *the* (*Mapping.lookup M f*) < *length slp*
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *interpret-slp slp xs* ! *slp-index-lookup slp M f = interpret-floatarith f xs*
assumes *subterms-Binop*[*simp*]:
⋀*a b*. *subterms* (*Binop a b*) = *insert* (*Binop a b*) (*subterms a* ∪ *subterms b*)
assumes *interpret-Binop*[*simp*]:
⋀*a b xs*. *interpret-floatarith* (*Binop a b*) *xs* = *binop* (*interpret-floatarith a xs*) (*interpret-floatarith b xs*)

shows *insert* (*Binop fa1 fa2*) (*subterms fa1* ∪ *subterms fa2*) ⊆ *Mapping.keys M′*
∧
Mapping.keys M ⊆ *Mapping.keys M′* ∧
(∀ *f*∈*Mapping.keys M′*. *subterms f* ⊆ *Mapping.keys M′* ∧
the (*Mapping.lookup M′ f*) < *length slp′* ∧
interpret-slp slp′ xs ! *slp-index-lookup slp′ M′ f* = *interpret-floatarith f xs*)
⟨*proof*⟩

lemma
unop-slp-of-fa-induction-step:
assumes
Unop-IH1:
⋀*M slp M′ slp′*. *slp-of-fa fa1 M slp* = (*M′*, *slp′*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *subterms f* ⊆ *Mapping.keys M*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *the* (*Mapping.lookup M f*) < *length slp*) ⟹
(⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *interpret-slp slp xs* ! *slp-index-lookup slp M f* =
interpret-floatarith f xs) ⟹
subterms fa1 ⊆ *Mapping.keys M′* ∧
Mapping.keys M ⊆ *Mapping.keys M′* ∧
(∀ *f*∈*Mapping.keys M′*. *subterms f* ⊆ *Mapping.keys M′* ∧
the (*Mapping.lookup M′ f*) < *length slp′* ∧
interpret-slp slp′ xs ! *slp-index-lookup slp′ M′ f* = *interpret-floatarith f xs*)
and *Unop-prems*:
(*case slp-of-fa fa1 M slp of* (*M1*, *slp1*) ⇒ *slp-of-fa-un Unop fa1 M slp M1 slp1*)
= (*M′*, *slp′*)
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *subterms f* ⊆ *Mapping.keys M*
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *the* (*Mapping.lookup M f*) < *length slp*
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *interpret-slp slp xs* ! *slp-index-lookup slp M f* =
interpret-floatarith f xs
assumes *subterms-Unop*[*simp*]:
⋀*a b*. *subterms* (*Unop a*) = *insert* (*Unop a*) (*subterms a*)
assumes *interpret-Unop*[*simp*]:
⋀*a b xs*. *interpret-floatarith* (*Unop a*) *xs* = *unop* (*interpret-floatarith a xs*)
shows *insert* (*Unop fa1*) (*subterms fa1*) ⊆ *Mapping.keys M′* ∧
Mapping.keys M ⊆ *Mapping.keys M′* ∧
(∀ *f*∈*Mapping.keys M′*. *subterms f* ⊆ *Mapping.keys M′* ∧
the (*Mapping.lookup M′ f*) < *length slp′* ∧
interpret-slp slp′ xs ! *slp-index-lookup slp′ M′ f* = *interpret-floatarith f xs*)
⟨*proof*⟩

lemma
cnst-slp-of-fa-induction-step:
assumes ∗:
slp-of-fa-cnst Unop Unop′ M slp = (*M′*, *slp′*)
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *subterms f* ⊆ *Mapping.keys M*
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *the* (*Mapping.lookup M f*) < *length slp*
⋀*f*. *f* ∈ *Mapping.keys M* ⟹ *interpret-slp slp xs* ! *slp-index-lookup slp M f* =
interpret-floatarith f xs
assumes *subterms-Unop*[*simp*]:

$\bigwedge a\ b.\ subterms\ (Unop) = \{Unop\}$
assumes *interpret-Unop*[*simp*]:
$interpret\text{-}floatarith\ Unop\ xs = unop\ xs$
$interpret\text{-}floatarith\ Unop'\ (interpret\text{-}slp\ slp\ xs) = unop\ xs$
assumes *ui*: $unop\ (interpret\text{-}slp\ slp\ xs) = unop\ xs$
shows $\{Unop\} \subseteq Mapping.keys\ M' \wedge$
$Mapping.keys\ M \subseteq Mapping.keys\ M' \wedge$
$(\forall f{\in}Mapping.keys\ M'.\ subterms\ f \subseteq Mapping.keys\ M' \wedge$
$the\ (Mapping.lookup\ M'\ f) < length\ slp' \wedge$
$interpret\text{-}slp\ slp'\ xs\ !\ slp\text{-}index\text{-}lookup\ slp'\ M'\ f = interpret\text{-}floatarith\ f\ xs)$
⟨*proof*⟩

lemma *interpret-slp-nth*:
$n \geq length\ slp \Longrightarrow interpret\text{-}slp\ slp\ xs\ !\ n = xs\ !\ (n - length\ slp)$
⟨*proof*⟩

theorem
interpret-slp-of-fa:
assumes $slp\text{-}of\text{-}fa\ fa\ M\ slp = (M', slp')$
assumes $\bigwedge f.\ f \in Mapping.keys\ M \Longrightarrow subterms\ f \subseteq Mapping.keys\ M$
assumes $\bigwedge f.\ f \in Mapping.keys\ M \Longrightarrow (the\ (Mapping.lookup\ M\ f)) < length\ slp$
assumes $\bigwedge f.\ f \in Mapping.keys\ M \Longrightarrow interpret\text{-}slp\ slp\ xs\ !\ slp\text{-}index\text{-}lookup\ slp\ M\ f = interpret\text{-}floatarith\ f\ xs$
shows $subterms\ fa \subseteq Mapping.keys\ M' \wedge Mapping.keys\ M \subseteq Mapping.keys\ M' \wedge$
$(\forall f \in Mapping.keys\ M'.$
$subterms\ f \subseteq Mapping.keys\ M' \wedge$
$the\ (Mapping.lookup\ M'\ f) < length\ slp' \wedge$
$(interpret\text{-}slp\ slp'\ xs\ !\ slp\text{-}index\text{-}lookup\ slp'\ M'\ f = interpret\text{-}floatarith\ f\ xs))$
⟨*proof*⟩

primrec *slp-of-fas'* **where**
$slp\text{-}of\text{-}fas'\ []\ M\ slp = (M, slp)$
$|\ slp\text{-}of\text{-}fas'\ (fa\#fas)\ M\ slp = (let\ (M, slp) = slp\text{-}of\text{-}fa\ fa\ M\ slp\ in\ slp\text{-}of\text{-}fas'\ fas\ M\ slp)$

theorem
interpret-slp-of-fas':
assumes $slp\text{-}of\text{-}fas'\ fas\ M\ slp = (M', slp')$
assumes $\bigwedge f.\ f \in Mapping.keys\ M \Longrightarrow subterms\ f \subseteq Mapping.keys\ M$
assumes $\bigwedge f.\ f \in Mapping.keys\ M \Longrightarrow the\ (Mapping.lookup\ M\ f) < length\ slp$
assumes $\bigwedge f.\ f \in Mapping.keys\ M \Longrightarrow interpret\text{-}slp\ slp\ xs\ !\ slp\text{-}index\text{-}lookup\ slp\ M\ f = interpret\text{-}floatarith\ f\ xs$
shows $\bigcup (subterms\ `\ set\ fas) \subseteq Mapping.keys\ M' \wedge Mapping.keys\ M \subseteq Mapping.keys\ M' \wedge$
$(\forall f \in Mapping.keys\ M'.\ subterms\ f \subseteq Mapping.keys\ M' \wedge$
$(the\ (Mapping.lookup\ M'\ f) < length\ slp') \wedge$
$(interpret\text{-}slp\ slp'\ xs\ !\ slp\text{-}index\text{-}lookup\ slp'\ M'\ f = interpret\text{-}floatarith\ f\ xs))$
⟨*proof*⟩

definition *slp-of-fas fas* =
(*let*
(*M*, *slp*) = *slp-of-fas′ fas Mapping.empty* [];
fasi = *map* (*the o Mapping.lookup M*) *fas*;
fasi′ = *map* (λ(*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*
⟨*proof*⟩

lemma *length-interpret-floatariths*[*simp*]:
length (*interpret-floatariths slp xs*) = *length slp*
⟨*proof*⟩

lemma *interpret-slp-append*[*simp*]:
interpret-slp (*slp1* @ *slp2*) *xs* =
interpret-slp slp2 (*interpret-slp slp1 xs*)
⟨*proof*⟩

lemma *interpret-slp* (*map Var* [*a* + *0*, *b* + *1*, *c* + *2*, *d* + *3*]) *xs* =
(*rev* (*map* (λ(*i*, *e*). *xs* ! (*e* − *i*)) (*zip* [*0*..<*4*] [*a* + *0*, *b* + *1*, *c* + *2*, *d* + *3*])))@*xs*
⟨*proof*⟩

lemma *aC-eq-aa*: *xs* @ *y* # *zs* = (*xs* @ [*y*]) @ *zs*
⟨*proof*⟩

lemma
interpret-slp-map-Var:
assumes ⋀*i*. *i* < *length is* ⟹ *is* ! *i* ≥ *i*
assumes ⋀*i*. *i* < *length is* ⟹ (*is* ! *i* − *i*) < *length xs*
shows *interpret-slp* (*map Var is*) *xs* =
(*rev* (*map* (λ(*i*, *e*). *xs* ! (*e* − *i*)) (*zip* [*0*..<*length is*] *is*)))
@
xs
⟨*proof*⟩

theorem *slp-of-fas*:
take (*length fas*) (*interpret-slp* (*slp-of-fas fas*) *xs*) = *interpret-floatariths fas xs*
⟨*proof*⟩

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 ⇒ (floatarith, nat) mapping ⇒ floatarith list ⇒ nat ⇒*
((floatarith, nat) mapping × floatarith list × 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 slpl1*)
| *slp-of-fa-rev* (*Power a n*) *M slp slpl* =
(*let* (*M1*, *slp1*, *slpl1*) = *slp-of-fa-rev a M slp slpl in slp-of-fa-rev-un* (*λa. Power a n*) *a M slp slpl M1 slp1 slpl1*)
| *slp-of-fa-rev Pi M slp slpl* = *slp-of-fa-rev-cnst Pi Pi M slp slpl*
| *slp-of-fa-rev* (*Var v*) *M slp slpl* = *slp-of-fa-rev-cnst* (*Var v*) (*Var* (*v* + *slpl*)) *M slp slpl*
| *slp-of-fa-rev* (*Num n*) *M slp slpl* = *slp-of-fa-rev-cnst* (*Num n*) (*Num n*) *M slp slpl*

lemma *slp-indexl-length*[*simp*]: *slp-indexl* (*length xs*) *i* = *slp-index xs i*
⟨*proof*⟩

lemma *slp-indexl-lookup-length*[*simp*]: *slp-indexl-lookup* (*length xs*) *i* = *slp-index-lookup xs i*
⟨*proof*⟩

lemma *slp-index-rev*[*simp*]: *slp-index* (*rev xs*) *i* = *slp-index xs i*
⟨*proof*⟩

lemma *slp-index-lookup-rev*[*simp*]: *slp-index-lookup* (*rev xs*) *i* = *slp-index-lookup xs i*
⟨*proof*⟩

lemma *slp-of-fa-bin-slp-of-fa-rev-bin*:
slp-of-fa-rev-bin Binop a b M slp (*length slp*) *M2 slp2* (*length slp2*) =
(*let* (*M*, *slp′*) = *slp-of-fa-bin Binop a b M* (*rev slp*) *M2* (*rev slp2*) *in* (*M*, *rev*

slp', *length slp'*))
⟨*proof*⟩

lemma *slp-of-fa-un-slp-of-fa-rev-un*:
slp-of-fa-rev-un Binop a M slp (*length slp*) *M2 slp2* (*length slp2*) =
(*let* (*M*, *slp'*) = *slp-of-fa-un Binop a M* (*rev slp*) *M2* (*rev slp2*) *in* (*M*, *rev slp'*, *length slp'*))
⟨*proof*⟩

lemma *slp-of-fa-cnst-slp-of-fa-rev-cnst*:
slp-of-fa-rev-cnst Cnst Cnst' M slp (*length slp*) =
(*let* (*M*, *slp'*) = *slp-of-fa-cnst Cnst Cnst' M* (*rev slp*) *in* (*M*, *rev slp'*, *length slp'*))
⟨*proof*⟩

lemma *slp-of-fa-rev*:
slp-of-fa-rev fa M slp (*length slp*) = (*let* (*M*, *slp'*) = *slp-of-fa fa M* (*rev slp*) *in* (*M*, *rev slp'*, *length slp'*))
⟨*proof*⟩

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'*))
⟨*proof*⟩

definition *norm2-slp n* = *slp-of-fas* [*floatarith.Inverse* (*norm2*$_e$ *n*)]

unbundle *no-floatarith-notation*

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 ⟦*convex-on ?A ?f*; *connected ?A*; *?c* ∈ *interior ?A*; *?x* ∈ *?A*; (*?f has-real-derivative ?f'*) (*at ?c within ?A*)⟧
⟹ *?f'* ∗ (*?x* − *?c*) ≤ *?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 $f\ x - f\ c \geq f' * (x - c)$
⟨*proof*⟩

Approximate operations on affine forms.

lemma *Affine-notempty*[*intro*, *simp*]: *Affine X* $\neq \{\}$
⟨*proof*⟩

lemma *truncate-up-lt*: $x < y \Longrightarrow x <$ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-up-pos-eq*[*simp*]: $0 <$ *truncate-up p x* $\longleftrightarrow 0 < x$
⟨*proof*⟩

lemma *inner-scaleR-pdevs-0*: *inner-scaleR-pdevs 0 One-pdevs* $=$ *zero-pdevs*
⟨*proof*⟩

lemma *Affine-aform-of-point-eq*[*simp*]: *Affine* (*aform-of-point p*) $= \{p\}$
⟨*proof*⟩

lemma *mem-Affine-aform-of-point*: $x \in$ *Affine* (*aform-of-point x*)
⟨*proof*⟩

lemma
aform-val-aform-of-ivl-innerE:
assumes $e \in$ *UNIV* $\to \{-1\ ..\ 1\}$
assumes $a \leq b\ c \in$ *Basis*
obtains *f* **where** *aform-val e* (*aform-of-ivl a b*) $\cdot\ c =$ *aform-val f* (*aform-of-ivl* $(a \cdot c)\ (b \cdot c)$)
$f \in$ *UNIV* $\to \{-1\ ..\ 1\}$
⟨*proof*⟩

lift-definition *coord-pdevs*::*nat* $\Rightarrow$ *real pdevs* **is** $\lambda n\ i.$ *if* $i = n$ *then 1 else 0* ⟨*proof*⟩

lemma *pdevs-apply-coord-pdevs* [*simp*]: *pdevs-apply* (*coord-pdevs i*) $x =$ (*if* $x = i$ *then 1 else 0*)
⟨*proof*⟩

lemma *degree-coord-pdevs*[*simp*]: *degree* (*coord-pdevs i*) $=$ *Suc i*
⟨*proof*⟩

lemma *pdevs-val-coord-pdevs*[*simp*]: *pdevs-val e* (*coord-pdevs i*) $= e\ i$
⟨*proof*⟩

definition *aforms-of-ivls ls us* $=$ *map*
(λ(*i*, (*l*, *u*)). ((*l* + *u*)/*2*, *scaleR-pdevs* ((*u* − *l*)/*2*) (*coord-pdevs i*)))
(*zip* [*0*..<*length ls*] (*zip ls us*))

lemma
aforms-of-ivls:

assumes *length ls = length us length xs = length ls*
assumes $\bigwedge i.\ i < length\ xs \Longrightarrow xs\ !\ i \in \{ls\ !\ i\ ..\ us\ !\ i\}$
shows *xs ∈ Joints (aforms-of-ivls ls us)*
⟨*proof*⟩

5.1 Approximate Operations

definition *max-pdev x = fold (λx y. if infnorm (snd x) ≥ infnorm (snd y) then x 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) = (points-of-list (x0 + x) xs @ points-of-list (x0 − x) xs)*

primrec *points-of-aform* **where**
points-of-aform (x, xs) = points-of-list x (list-of-pdevs xs)

5.1.2 Approximate total deviation

definition *sum-list'*::*nat* ⇒ *'a list* ⇒ *'a*::*executable-euclidean-space*
where *sum-list' p xs = fold (λa b. eucl-truncate-up p (a + b)) xs 0*

definition *tdev' p x = sum-list' p (map (abs o snd) (list-of-pdevs x))*

lemma
eucl-fold-mono:
fixes *f*::*'a*::*ordered-euclidean-space*⇒*'a*⇒*'a*
assumes *mono*: $\bigwedge w\ x\ y\ z.\ w \le x \Longrightarrow y \le z \Longrightarrow f\ w\ y \le f\ x\ z$
shows $x \le y \Longrightarrow fold\ f\ xs\ x \le fold\ f\ xs\ y$
⟨*proof*⟩

lemma *sum-list-add-le-fold-eucl-truncate-up*:
fixes *z*::*'a*::*executable-euclidean-space*
shows *sum-list xs + z ≤ fold (λx y. eucl-truncate-up p (x + y)) xs z*
⟨*proof*⟩

lemma *sum-list-le-sum-list'*:
sum-list xs ≤ sum-list' p xs
⟨*proof*⟩

lemma *sum-list'-sum-list-le*:
$y \le sum\text{-}list\ xs \Longrightarrow y \le sum\text{-}list'\ p\ xs$
⟨*proof*⟩

lemma *tdev'*: *tdev x ≤ tdev' p x*
⟨*proof*⟩

lemma *tdev'-le*: $x \leq tdev\ y \Longrightarrow x \leq tdev'\ p\ y$
⟨*proof*⟩

lemmas *abs-pdevs-val-le-tdev'* = *tdev'-le*[*OF abs-pdevs-val-le-tdev*]

lemma *tdev'-uminus-pdevs*[*simp*]: $tdev'\ p\ (uminus\text{-}pdevs\ x) = tdev'\ p\ x$
⟨*proof*⟩

abbreviation *Radius*::'*a*::*ordered-euclidean-space aform* ⇒ '*a*
where *Radius X* ≡ *tdev* (*snd X*)

abbreviation *Radius'*::*nat*⇒'*a*::*executable-euclidean-space aform* ⇒ '*a*
where *Radius' p X* ≡ *tdev' p* (*snd X*)

lemma *Radius'-uminus-aform*[*simp*]: $Radius'\ p\ (uminus\text{-}aform\ X) = Radius'\ p\ X$
⟨*proof*⟩

5.1.3 truncate partial deviations

definition *trunc-pdevs-raw*::*nat* ⇒ (*nat* ⇒ '*a*) ⇒ *nat* ⇒ '*a*::*executable-euclidean-space*
where *trunc-pdevs-raw p x i* = *eucl-truncate-down p* (*x i*)

lemma *nonzeros-trunc-pdevs-raw*:
$\{i.\ trunc\text{-}pdevs\text{-}raw\ r\ x\ i \neq 0\} \subseteq \{i.\ x\ i \neq 0\}$
⟨*proof*⟩

lift-definition *trunc-pdevs*::*nat* ⇒ '*a*::*executable-euclidean-space pdevs* ⇒ '*a pdevs*
is *trunc-pdevs-raw*
⟨*proof*⟩

definition *trunc-err-pdevs-raw*::*nat* ⇒ (*nat* ⇒ '*a*) ⇒ *nat* ⇒ '*a*::*executable-euclidean-space*
where *trunc-err-pdevs-raw p x i* = *trunc-pdevs-raw p x i* − *x i*

lemma *nonzeros-trunc-err-pdevs-raw*:
$\{i.\ trunc\text{-}err\text{-}pdevs\text{-}raw\ r\ x\ i \neq 0\} \subseteq \{i.\ x\ i \neq 0\}$
⟨*proof*⟩

lift-definition *trunc-err-pdevs*::*nat* ⇒ '*a*::*executable-euclidean-space pdevs* ⇒ '*a pdevs*
is *trunc-err-pdevs-raw*
⟨*proof*⟩

term *float-plus-down*

lemma *pdevs-apply-trunc-pdevs*[*simp*]:
fixes *x y*::'*a*::*euclidean-space*
shows $pdevs\text{-}apply\ (trunc\text{-}pdevs\ p\ X)\ n = eucl\text{-}truncate\text{-}down\ p\ (pdevs\text{-}apply\ X\ n)$
⟨*proof*⟩

lemma *pdevs-apply-trunc-err-pdevs*[*simp*]:
fixes *x y*::*'a*::*euclidean-space*
shows *pdevs-apply* (*trunc-err-pdevs p X*) *n* =
eucl-truncate-down p (*pdevs-apply X n*) − (*pdevs-apply X n*)
⟨*proof*⟩

lemma *pdevs-val-trunc-pdevs*:
fixes *x y*::*'a*::*euclidean-space*
shows *pdevs-val e* (*trunc-pdevs p X*) = *pdevs-val e X* + *pdevs-val e* (*trunc-err-pdevs p X*)
⟨*proof*⟩

lemma *pdevs-val-trunc-err-pdevs*:
fixes *x y*::*'a*::*euclidean-space*
shows *pdevs-val e* (*trunc-err-pdevs p X*) = *pdevs-val e* (*trunc-pdevs p X*) − *pdevs-val e X*
⟨*proof*⟩

definition *truncate-aform*::*nat* ⇒ *'a aform* ⇒ *'a*::*executable-euclidean-space aform*
where *truncate-aform p x* = (*eucl-truncate-down p* (*fst x*), *trunc-pdevs p* (*snd x*))

definition *truncate-error-aform*::*nat* ⇒ *'a aform* ⇒ *'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* ∈ *UNIV* → {− *1..1*}
shows *abs* (*aform-val e X*) ≤ *eucl-truncate-up p* (|*fst X*| + *tdev' p* (*snd X*))
⟨*proof*⟩

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*| ≤ *snd* (*trunc-bound-eucl p x*)
fst (*trunc-bound-eucl p x*) = *x* + *err*
⟨*proof*⟩

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-apply* (*fst* (*trunc-bound-pdevs* p x)) =
pdevs-apply (*trunc-pdevs* p x)
⟨*proof*⟩

lemma *trunc-bound-pdevsE*:
assumes $e \in UNIV \to \{-1..1\}$
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*⟩

lemma
degree-add-pdevs-le:
assumes *degree* $X \leq n$
assumes *degree* $Y \leq n$
shows *degree* (*add-pdevs* X Y) $\leq n$
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *error-absE*:
assumes *abs err* $\leq k$
obtains e::*real* **where** *err* = $e * k$ $e \in \{-1 .. 1\}$
⟨*proof*⟩

lemma *eucl-truncate-up-nonneg-eq-zero-iff*:
$x \geq 0 \Longrightarrow$ *eucl-truncate-up* p x = 0 $\longleftrightarrow$ x = 0
⟨*proof*⟩

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
⟨*proof*⟩

lemma
aform-val-consume-errorE:
fixes X::*real aform*
assumes *abs err* $\leq$ *abs* (*pdevs-apply* (*snd* X) n)
obtains *err*′ **where** *aform-val* ($e(n := 0)$) X + *err* = *aform-val* ($e(n :=$ *err*′)) X *err*′ $\in \{-1 .. 1\}$

⟨*proof*⟩

lemma
degree-trunc-pdevs-le:
assumes *degree* $X \leq n$
shows *degree* (*trunc-pdevs p X*) $\leq n$
⟨*proof*⟩

lemma *pdevs-val-sum-less-degree*:
pdevs-val e X = ($\sum$ *i*<*d*. *e i* $*_R$ *pdevs-apply X i*) **if** *degree X* $\leq d$
⟨*proof*⟩

5.1.5 general affine operation

definition *affine-binop* (*X*::*real aform*) *Y a b c d k* =
(*a* ∗ *fst X* + *b* ∗ *fst Y* + *c*,
pdev-upd (*add-pdevs* (*scaleR-pdevs a* (*snd X*)) (*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*)}
⟨*proof*⟩

lemma *pdevs-val-One-pdevs*:
pdevs-val e (*One-pdevs*::*'a*::*executable-euclidean-space pdevs*) = ($\sum$ *i*<*DIM*(*'a*). *e i* $*_R$ *Basis-list* ! *i*)
⟨*proof*⟩

lemma *affine-binop*:
assumes *degree-aforms* [*X*, *Y*] $\leq k$
shows *aform-val e* (*affine-binop X Y a b c d k*) =
a ∗ *aform-val e X* + *b* ∗ *aform-val e Y* + *c* + *e k* ∗ *d*
⟨*proof*⟩

definition *affine-binop'* *p* (*X*::*real aform*) *Y a b c d k* =
(*let*
— TODO: more round-off operations here?
(*r*, *e1*) = *trunc-bound-eucl p* (*a* ∗ *fst X* + *b* ∗ *fst Y* + *c*);
(*Z*, *e2*) = *trunc-bound-pdevs p* (*add-pdevs* (*scaleR-pdevs a* (*snd X*)) (*scaleR-pdevs b* (*snd Y*)))
in
(*r*, *pdev-upd Z k* (*sum-list' p* [*e1*, *e2*, *d*]))
)

lemma *sum-list'-noneg-eq-zero-iff*: *sum-list' p xs* = *0* ⟷ (∀ *x*∈*set xs*. *x* = *0*) **if** $\bigwedge x.\ x \in set\ xs \Longrightarrow x \geq 0$
⟨*proof*⟩

lemma *affine-binop'E*:

assumes *deg*: *degree-aforms* $[X,\ Y] \leq k$
assumes *e*: $e \in UNIV \to \{-\ 1..1\}$
assumes *d*: $abs\ u \leq d$
obtains *ek* **where**
$a * aform\text{-}val\ e\ X + b * aform\text{-}val\ e\ Y + c + u = aform\text{-}val\ (e(k{:=}ek))$
(*affine-binop'* *p X Y a b c d k*)
$ek \in \{-1\ ..\ 1\}$
⟨*proof*⟩

5.1.6 Inf/Sup

definition *Inf-aform'* *p X* = *eucl-truncate-down p* (*fst X* − *tdev'* *p* (*snd X*))

definition *Sup-aform'* *p X* = *eucl-truncate-up p* (*fst X* + *tdev'* *p* (*snd X*))

lemma *Inf-aform'*:
shows *Inf-aform'* *p X* ≤ *Inf-aform X*
⟨*proof*⟩

lemma *Sup-aform'*:
shows *Sup-aform X* ≤ *Sup-aform'* *p X*
⟨*proof*⟩

lemma *Inf-aform-le-Sup-aform*[*intro*]:
Inf-aform X ≤ *Sup-aform X*
⟨*proof*⟩

lemma *Inf-aform'-le-Sup-aform'*[*intro*]:
Inf-aform' *p X* ≤ *Sup-aform'* *p X*
⟨*proof*⟩

definition
ivls-of-aforms prec = *map* (λ*a*. *Interval'* (*float-of* (*Inf-aform'* *prec a*)) (*float-of*(*Sup-aform'* *prec a*)))

lemma
assumes ⋀*i*. $e''\ i \leq 1$
assumes ⋀*i*. $-1 \leq e''\ i$
shows *Inf-aform'-le*: *Inf-aform'* *p r* ≤ *aform-val e'' r*
and *Sup-aform'-le*: *aform-val e'' r* ≤ *Sup-aform'* *p r*
⟨*proof*⟩

lemma *InfSup-aform'-in-float*[*intro*, *simp*]:
Inf-aform' *p X* ∈ *float Sup-aform'* *p X* ∈ *float*
⟨*proof*⟩

theorem *ivls-of-aforms*: *xs* ∈ *Joints XS* ⟹ *bounded-by xs* (*ivls-of-aforms prec XS*)

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

5.2 Min Range approximation

lemma
linear-lower:
fixes *x*::*real*
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ (*f has-field-derivative f*′ *x*) (*at x within* {*a* .. *b*})
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ *f*′ *x* ≤ *u*
assumes *x* ∈ {*a* .. *b*}
shows *f b* + *u* ∗ (*x* − *b*) ≤ *f x*
⟨*proof*⟩

lemma
linear-lower2:
fixes *x*::*real*
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ (*f has-field-derivative f*′ *x*) (*at x within* {*a* .. *b*})
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ *l* ≤ *f*′ *x*
assumes *x* ∈ {*a* .. *b*}
shows *f x* ≥ *f a* + *l* ∗ (*x* − *a*)
⟨*proof*⟩

lemma
linear-upper:
fixes *x*::*real*
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ (*f has-field-derivative f*′ *x*) (*at x within* {*a* .. *b*})
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ *f*′ *x* ≤ *u*
assumes *x* ∈ {*a* .. *b*}
shows *f x* ≤ *f a* + *u* ∗ (*x* − *a*)
⟨*proof*⟩

lemma
linear-upper2:
fixes *x*::*real*
assumes ⋀*x*. *x* ∈ {*a* .. *b*} ⟹ (*f has-field-derivative f*′ *x*) (*at x within* {*a* .. *b*})

assumes $\bigwedge x.\ x \in \{a\ ..\ b\} \Longrightarrow l \leq f'\ x$
assumes $x \in \{a\ ..\ b\}$
shows $f\ x \leq f\ b + l * (x - b)$
⟨*proof*⟩

lemma *linear-enclosure*:
fixes x::*real*
assumes $\bigwedge x.\ x \in \{a\ ..\ b\} \Longrightarrow (f$ *has-field-derivative* $f'\ x)$ (*at* x *within* $\{a\ ..\ b\})$
assumes $\bigwedge x.\ x \in \{a\ ..\ b\} \Longrightarrow f'\ x \leq u$
assumes $x \in \{a\ ..\ b\}$
shows $f\ x \in \{f\ b + u * (x - b)\ ..\ f\ a + u * (x - a)\}$
⟨*proof*⟩

definition *mid-err ivl* = ((*lower ivl* + *upper ivl*::*float*)/*2*, (*upper ivl* − *lower ivl*)/*2*)

lemma *degree-aform-uminus-aform*[*simp*]: *degree-aform* (*uminus-aform X*) = *degree-aform X*
⟨*proof*⟩

5.2.1 Addition

definition *add-aform*::'*a*::*real-vector aform* ⇒ '*a aform* ⇒ '*a aform*
where *add-aform x y* = (*fst x* + *fst y*, *add-pdevs* (*snd x*) (*snd y*))

lemma *aform-val-add-aform*:
shows *aform-val e* (*add-aform X Y*) = *aform-val e X* + *aform-val e Y*
⟨*proof*⟩

type-synonym *aform-err* = *real aform* × *real*

definition *add-aform'*::*nat* ⇒ *aform-err* ⇒ *aform-err* ⇒ *aform-err*
where *add-aform' p x y* =
(*let*
z0 = *trunc-bound-eucl p* (*fst* (*fst x*) + *fst* (*fst y*));
z = *trunc-bound-pdevs p* (*add-pdevs* (*snd* (*fst x*)) (*snd* (*fst y*)))
in ((*fst z0*, *fst z*), (*sum-list' p* [*snd z0*, *snd z*, *abs* (*snd x*), *abs* (*snd y*)])))

abbreviation *degree-aform-err*::*aform-err* ⇒ *nat*
where *degree-aform-err X* ≡ *degree-aform* (*fst X*)

lemma *degree-aform-err-add-aform'*:
assumes *degree-aform-err x* ≤ *n*
assumes *degree-aform-err y* ≤ *n*
shows *degree-aform-err* (*add-aform' p x y*) ≤ *n*
⟨*proof*⟩

definition *aform-err e Xe* = {*aform-val e* (*fst Xe*) − *snd Xe* .. *aform-val e* (*fst Xe*) + *snd Xe*::*real*}

lemma *aform-errI*: $x \in$ *aform-err e Xe*
if *abs* $(x -$ *aform-val e* $(fst\ Xe)) \leq$ *snd Xe*
⟨*proof*⟩

lemma *add-aform'*:
assumes *e*: $e \in UNIV \to \{-1..1\}$
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*⟩

5.2.2 Scaling

definition *aform-scaleR*::*real aform* ⇒ *'a*::*real-vector* ⇒ *'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*
⟨*proof*⟩

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*))) +
(∑ *i*<*d*. *e i* $*_R$ *pdevs-apply* (*snd x*) *i*)∗(∑ *i*<*d*. *e i* $*_R$ *pdevs-apply* (*snd y*) *i*)
if *degree* (*snd x*) ≤ *d degree* (*snd y*) ≤ *d*
⟨*proof*⟩

lemma *sum-times-bound*:— TODO: this gives better bounds for the remainder of multiplication
(∑ *i*<*d*. *e i* ∗ *f i*::*real*) ∗ (∑ *i*<*d*. *e i* ∗ *g i*) =
(∑ *i*<*d*. (*e i*)^2 ∗ (*f i* ∗ *g i*)) +
(∑ (*i*, *j*) | *i* < *j* ∧ *j* < *d*. (*e i* ∗ *e j*) ∗ (*f j* ∗ *g i* + *f i* ∗ *g j*)) **for** *d*::*nat*
⟨*proof*⟩

definition *mult-aform*::*aform-err* ⇒ *aform-err* ⇒ *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*: $e \in UNIV \to \{-\ 1..1\}$
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*⟩

definition *mult-aform'*::*nat* ⇒ *aform-err* ⇒ *aform-err* ⇒ *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*
⟨*proof*⟩

lemma *mult-aform'E*:
fixes *X Y*::*aform-err*
assumes *e*: $e \in UNIV \to \{-\ 1..1\}$
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' p X Y*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*: *inverse x* ≤ *inverse a* + *alpha* ∗ (*x* − *a*) (**is** *?upper*)
⟨*proof*⟩

5.2.4 Inverse

definition *inverse-aform′*::*nat* ⇒ *real aform* ⇒ *real aform* × *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* ∈ *UNIV* → {−*1* .. *1*}
assumes *Inf-pos*: *Inf-aform′ p X* > *0*
assumes *x* = *aform-val e X*
shows *inverse x* ∈ *aform-err e* (*inverse-aform′ p X*)
⟨*proof*⟩

definition *inverse-aform p a* =
do {
let l = *Inf-aform′ p a*;
let u = *Sup-aform′ p a*;
if (*l* ≤ *0* ∧ *0* ≤ *u*) *then None*
else if (*l* ≤ *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*))
⟨*proof*⟩

lemma *inverse-aformE*:
fixes *X*::*real aform*

assumes *e*: $e \in UNIV \to \{-1 \mathinner{..} 1\}$
and *disj*: $Inf\text{-}aform'\ p\ X > 0 \vee Sup\text{-}aform'\ p\ X < 0$
obtains Y **where**
$inverse\text{-}aform\ p\ X = Some\ Y$
$inverse\ (aform\text{-}val\ e\ X) \in aform\text{-}err\ e\ Y$
⟨*proof*⟩

definition *aform-err-to-aform*::*aform-err* ⇒ *nat* ⇒ *real aform*
where $aform\text{-}err\text{-}to\text{-}aform\ X\ n = (fst\ (fst\ X),\ \ pdev\text{-}upd\ (snd\ (fst\ X))\ n\ (snd\ X))$

lemma *aform-err-to-aformE*:
assumes $x \in aform\text{-}err\ e\ X$
assumes *deg*: $degree\text{-}aform\text{-}err\ X \le n$
obtains *err* **where** $x = aform\text{-}val\ (e(n{:=}err))\ (aform\text{-}err\text{-}to\text{-}aform\ X\ n)$
$-1 \le err\ err \le 1$
⟨*proof*⟩

definition *aform-to-aform-err*::*real aform* ⇒ *nat* ⇒ *aform-err*
where $aform\text{-}to\text{-}aform\text{-}err\ X\ n = ((fst\ X,\ \ pdev\text{-}upd\ (snd\ X)\ n\ 0), abs\ (pdevs\text{-}apply\ (snd\ X)\ n))$

lemma *aform-to-aform-err*: $aform\text{-}val\ e\ X \in aform\text{-}err\ e\ (aform\text{-}to\text{-}aform\text{-}err\ X\ n)$
if $e \in UNIV \to \{-1 \mathinner{..} 1\}$
⟨*proof*⟩

definition $acc\text{-}err\ p\ x\ e \equiv (fst\ x,\ truncate\text{-}up\ p\ (snd\ x + e))$

definition *ivl-err* :: *real interval* ⇒ (*real* × *real pdevs*) × *real*
where $ivl\text{-}err\ ivl \equiv (((upper\ ivl + lower\ ivl)/2,\ zero\text{-}pdevs{::}real\ pdevs), (upper\ ivl - lower\ ivl)\ /\ 2)$

lemma *inverse-aform*:
fixes *X*::*real aform*
assumes *e*: $e \in UNIV \to \{-1 \mathinner{..} 1\}$
assumes $inverse\text{-}aform\ p\ X = Some\ Y$
shows $inverse\ (aform\text{-}val\ e\ X) \in aform\text{-}err\ e\ Y$
⟨*proof*⟩

lemma *aform-err-acc-err-leI*:
$fx \in aform\text{-}err\ e\ (acc\text{-}err\ p\ X\ err)$
if $aform\text{-}val\ e\ (fst\ X) - (snd\ X + err) \le fx\ fx \le aform\text{-}val\ e\ (fst\ X) + (snd\ X + err)$
⟨*proof*⟩

lemma *aform-err-acc-errI*:
$fx \in aform\text{-}err\ e\ (acc\text{-}err\ p\ X\ err)$
if $fx \in aform\text{-}err\ e\ (fst\ X,\ snd\ X + err)$

⟨*proof*⟩

lemma *minus-times-le-abs*: $-(err * B) \leq |B|$ **if** $-1 \leq err\ err \leq 1$ **for** *err*::*real*
⟨*proof*⟩

lemma *times-le-abs*: $err * B \leq |B|$ **if** $-1 \leq err\ err \leq 1$ **for** *err*::*real*
⟨*proof*⟩

lemma *aform-err-lemma1*: $-1 \leq err \Longrightarrow err \leq 1 \Longrightarrow$
$X1 + (A - e\ d * B + err * B) - e1 \leq x \Longrightarrow$
$X1 + (A - e\ d * B) - truncate\text{-}up\ p\ (|B| + e1) \leq x$
⟨*proof*⟩

lemma *aform-err-lemma2*: $-1 \leq err \Longrightarrow err \leq 1 \Longrightarrow$
$x \leq X1 + (A - e\ d * B + err * B) + e1 \Longrightarrow$
$x \leq X1 + (A - e\ d * B) + truncate\text{-}up\ p\ (|B| + e1)$
⟨*proof*⟩

lemma *aform-err-acc-err-aform-to-aform-errI*:
$x \in aform\text{-}err\ e\ (acc\text{-}err\ p\ (aform\text{-}to\text{-}aform\text{-}err\ X1\ d)\ e1)$
if $-1 \leq err\ err \leq 1\ x \in aform\text{-}err\ (e(d := err))\ (X1, e1)$
⟨*proof*⟩

definition *map-aform-err I p X* =
(*do* {
let X0 = *aform-err-to-aform X* (*degree-aform-err X*);
(*X1*, *e1*) ← *I X0*;
Some (*acc-err p* (*aform-to-aform-err X1* (*degree-aform-err X*)) *e1*)
})

lemma *map-aform-err*:
$i\ x \in aform\text{-}err\ e\ Y$
if *I*: $\bigwedge e\ X\ Y.\ e \in UNIV \to \{-1\ ..\ 1\} \Longrightarrow I\ X = Some\ Y \Longrightarrow i\ (aform\text{-}val\ e\ X) \in aform\text{-}err\ e\ Y$
and *e*: $e \in UNIV \to \{-1\ ..\ 1\}$
and *Y*: *map-aform-err I p X* = *Some Y*
and *x*: $x \in aform\text{-}err\ e\ X$
⟨*proof*⟩

definition *inverse-aform-err p X* = *map-aform-err* (*inverse-aform p*) *p X*

lemma *inverse-aform-err*:
$inverse\ x \in aform\text{-}err\ e\ Y$
if *e*: $e \in UNIV \to \{-1\ ..\ 1\}$
and *Y*: *inverse-aform-err p X* = *Some Y*
and *x*: $x \in aform\text{-}err\ e\ X$
⟨*proof*⟩

5.3 Reduction (Summarization of Coefficients)

definition *pdevs-of-centered-ivl r* = (*inner-scaleR-pdevs r One-pdevs*)

lemma *pdevs-of-centered-ivl-eq-pdevs-of-ivl*[*simp*]: *pdevs-of-centered-ivl r* = *pdevs-of-ivl* (−*r*) *r*
⟨*proof*⟩

lemma *filter-pdevs-raw-nonzeros*: {*i*. *filter-pdevs-raw s f i* ≠ *0*} = {*i*. *f i* ≠ *0*} ∩ {*x*. *s x* (*f x*)}
⟨*proof*⟩

definition *summarize-pdevs*::
nat ⇒ (*nat* ⇒ *'a* ⇒ *bool*) ⇒ *nat* ⇒ *'a*::*executable-euclidean-space pdevs* ⇒ *'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* ⟷ *infnorm y* ≥ *t* ∗ *infnorm* (*eucl-truncate-up p* (*tdev' p x*))

lemma *error-abs-euclE*:
fixes *err*::*'a*::*ordered-euclidean-space*
assumes *abs err* ≤ *k*
obtains *e*::*'a* ⇒ *real* **where** *err* = (∑ *i*∈*Basis*. (*e i* ∗ (*k* ⋅ *i*)) ∗$_R$ *i*) *e* ∈ *UNIV* → {−*1* .. *1*}
⟨*proof*⟩

lemma *summarize-pdevsE*:
fixes *x*::*'a*::*executable-euclidean-space pdevs*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *d*: *degree x* ≤ *d*
obtains *e'* **where** *pdevs-val e x* = *pdevs-val e'* (*summarize-pdevs p I d x*)
⋀*i*. *i* < *d* ⟹ *e i* = *e' i*
e' ∈ *UNIV* → {−*1* .. *1*}
⟨*proof*⟩

definition *summarize-pdevs-list p I d xs* =
map (λ(*d*, *x*). *summarize-pdevs p* (λ*i* -. *I i* (*pdevs-applys xs i*)) *d x*) (*zip* [*d*..<*d* + *length xs*] *xs*)

lemma *filter-pdevs-cong*[*cong*]:
assumes *x* = *y*
assumes ⋀*i*. *i* ∈ *pdevs-domain y* ⟹ *P i* (*pdevs-apply x i*) = *Q i* (*pdevs-apply y i*)
shows *filter-pdevs P x* = *filter-pdevs Q y*
⟨*proof*⟩

lemma *summarize-pdevs-cong*[*cong*]:
assumes $p = q$ $a = c$ $b = d$
assumes *PQ*: $\bigwedge i.\ i \in$ *pdevs-domain* $d \Longrightarrow P\ i$ (*pdevs-apply* $b\ i$) $= Q\ i$ (*pdevs-apply* $d\ i$)
shows *summarize-pdevs* $p\ P\ a\ b =$ *summarize-pdevs* $q\ Q\ c\ d$
⟨*proof*⟩

lemma *lookup-eq-None-iff*: (*Mapping.lookup* $M\ x =$ *None*) $= (x \notin$ *Mapping.keys* $M)$
⟨*proof*⟩

lemma *lookup-eq-SomeD*:
(*Mapping.lookup* $M\ x =$ *Some* $y) \Longrightarrow (x \in$ *Mapping.keys* $M)$
⟨*proof*⟩

definition *domain-pdevs* $xs = (\bigcup$(*pdevs-domain* ‘ (*set* xs)))

definition *pdevs-mapping* $xs =$
(*let*
$D =$ *sorted-list-of-set* (*domain-pdevs* xs);
$M =$ *Mapping.tabulate* D (*pdevs-applys* xs);
zeroes $=$ *replicate* (*length* xs) *0*
in Mapping.lookup-default zeroes M)

lemma *pdevs-mapping-eq*[*simp*]: *pdevs-mapping* $xs =$ *pdevs-applys* xs
⟨*proof*⟩

lemma *compute-summarize-pdevs-list*[*code*]:
summarize-pdevs-list $p\ I\ d\ xs =$
(*let* $M =$ *pdevs-mapping* xs
in map ($\lambda(x,\ y)$. *summarize-pdevs* p (λi -. $I\ i\ (M\ i)$) $x\ y$) (*zip* [d..<$d +$ *length* xs] xs))
⟨*proof*⟩

lemma
in-centered-ivlE:
fixes r *t*::*real*
assumes $r \in \{-t\ ..\ t\}$
obtains e **where** $e \in \{-1\ ..\ 1\}$ $r = e * t$
⟨*proof*⟩

lift-definition *singleton-pdevs*::$'a \Rightarrow\ 'a$::*real-normed-vector pdevs* **is** $\lambda x\ i.$ *if* $i = 0$ *then* x *else 0*
⟨*proof*⟩
lemmas [*simp*] $=$ *singleton-pdevs.rep-eq*

lemma *singleton-0*[*simp*]: *singleton-pdevs 0* $=$ *zero-pdevs*
⟨*proof*⟩

lemma *degree-singleton-pdevs*[*simp*]: *degree* (*singleton-pdevs x*) = (*if x = 0 then 0 else Suc 0*)
⟨*proof*⟩

lemma *pdevs-val-singleton-pdevs*[*simp*]: *pdevs-val e* (*singleton-pdevs x*) = *e 0* $*_R$ *x*
⟨*proof*⟩

lemma *pdevs-of-ivl-real*:
fixes *a b*::*real*
shows *pdevs-of-ivl a b* = *singleton-pdevs* ((*b* − *a*) / *2*)
⟨*proof*⟩

lemma *summarize-pdevs-listE*:
fixes *X*::*real pdevs list*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *d*: *degrees X* ≤ *d*
obtains *e′* **where** *pdevs-vals e X* = *pdevs-vals e′* (*summarize-pdevs-list p I d X*)
⋀*i*. *i* < *d* ⟹ *e i* = *e′ i*
e′ ∈ *UNIV* → {−*1* .. *1*}
⟨*proof*⟩

fun *list-ex2* **where**
list-ex2 P [] *xs* = *False*
| *list-ex2 P xs* [] = *False*
| *list-ex2 P* (*x*#*xs*) (*y*#*ys*) = (*P x y* ∨ *list-ex2 P xs ys*)

lemma *list-ex2-iff*:
list-ex2 P xs ys ⟷ (¬*list-all2* (−*P*) (*take* (*length ys*) *xs*) (*take* (*length xs*) *ys*))
⟨*proof*⟩

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* (λ(*x*, *y*). *x* + *y*) (*zip* (*map fst X*) (*pdevs-vals e* (*map snd X*)))
⟨*proof*⟩

lemma *summarize-aformsE*:
fixes *X*::*real aform list*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *d*: *degree-aforms X* ≤ *d*
obtains *e′* **where** *aform-vals e X* = *aform-vals e′* (*summarize-aforms p C d X*)
⋀*i*. *i* < *d* ⟹ *e i* = *e′ i*
e′ ∈ *UNIV* → {−*1* .. *1*}
⟨*proof*⟩

Different reduction strategies:

definition *collect-threshold p ta t* (*X*::*real aform list*) =
(*let*
Xs = *map snd X*;
as = *map* (λ*X*. *max ta* (*t* ∗ *tdev*′ *p X*)) *Xs*
in (λ(*i*::*nat*) *xs*. *list-ex2* (≤) *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 ≤ *m then* (λ- -. *True*) *else*
let
Ds = *sorted-list-of-set D*;
ortho-indices = *map fst* (*take* (2 ∗ *N*) (*sort-key* (λ(*i*, *r*). *r*) (*map* (λ*i*. *let xs* = *M i in* (*i*, *sum-list*′ *p xs* − *fold max xs* 0)) *Ds*)));
- = ()
in (λ*i* (*xs*::*real list*). *i* ∈ *set ortho-indices*))

5.4 Splitting with heuristics

definition *abs-pdevs* = *unop-pdevs abs*

definition *abssum-of-pdevs-list X* = *fold* (λ*a b*. (*add-pdevs* (*abs-pdevs a*) *b*)) *X zero-pdevs*

definition *split-aforms xs i* = (*let splits* = *map* (λ*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* ⇒ *aform-err* ⇒ *float interval*
is λ*p Rd*. (*Inf-aform-err p Rd*, *Sup-aform-err p Rd*)
⟨*proof*⟩
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*
⟨*proof*⟩
end

definition *approx-un*::*nat*
⇒ (*float interval* ⇒ *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 ← *a*;
ivl ← *f* (*ivl-of-aform-err p rd*);
Some (*ivl-err* (*real-interval ivl*))
}

definition *interval-extension1*::(*float interval* ⇒ (*float interval*) *option*) ⇒ (*real* ⇒ *real*) ⇒ *bool*
where *interval-extension1 F f* ⟷ (∀ *ivl ivl'*. *F ivl* = *Some ivl'* ⟶ (∀ *x*. *x* ∈$_r$ *ivl* ⟶ *f x* ∈$_r$ *ivl'*))

lemma *interval-extension1D*:
assumes *interval-extension1 F f*
assumes *F ivl* = *Some ivl'*
assumes *x* ∈$_r$ *ivl*
shows *f x* ∈$_r$ *ivl'*
⟨*proof*⟩

lemma *approx-un-argE*:
assumes *au*: *approx-un p F X* = *Some Y*
obtains *X'* **where** *X* = *Some X'*
⟨*proof*⟩

lemma *degree-aform-independent-from*:
degree-aform (*independent-from d1 X*) ≤ *d1* + *degree-aform X*
⟨*proof*⟩

lemma *degree-aform-of-ivl*:
fixes *a b*::'*a*::*executable-euclidean-space*
shows *degree-aform* (*aform-of-ivl a b*) ≤ *length* (*Basis-list*::'*a list*)
⟨*proof*⟩

lemma *aform-err-ivl-err*[*simp*]: *aform-err e* (*ivl-err ivl'*) = *set-of ivl'*
⟨*proof*⟩

lemma *Inf-Sup-aform-err*:
fixes *X*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
defines *X'* ≡ *fst X*
shows *aform-err e X* ⊆ {*Inf-aform-err p X* .. *Sup-aform-err p X*}
⟨*proof*⟩

lemma *ivl-of-aform-err*:
fixes *X*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
shows *x* ∈ *aform-err e X* ⟹ *x* ∈$_r$ *ivl-of-aform-err p X*
⟨*proof*⟩

lemma *approx-unE*:
assumes *ie*: *interval-extension1 F f*
assumes *e*: $e \in UNIV \to \{-1 .. 1\}$
assumes *au*: *approx-un p F X'err = Some Ye*
assumes *x*: *case X'err of None ⇒ True | Some X'err ⇒ x ∈ aform-err e X'err*
shows *f x ∈ aform-err e Ye*
⟨*proof*⟩

definition *approx-bin p f rd sd = do {*
ivl ← f (ivl-of-aform-err p rd)
(ivl-of-aform-err p sd);
Some (ivl-err (real-interval ivl))
}

definition *interval-extension2*::(*float interval ⇒ float interval ⇒ float interval option*) ⇒ (*real ⇒ real ⇒ real*) ⇒ *bool*
where *interval-extension2 F f ⟷ (∀ ivl1 ivl2 ivl. F ivl1 ivl2 = Some ivl ⟶*
(∀ x y. x ∈ᵣ ivl1 ⟶ y ∈ᵣ ivl2 ⟶ f x y ∈ᵣ ivl))

lemma *interval-extension2D*:
assumes *interval-extension2 F f*
assumes *F ivl1 ivl2 = Some ivl*
shows *x ∈ᵣ ivl1 ⟹ y ∈ᵣ ivl2 ⟹ f x y ∈ᵣ ivl*
⟨*proof*⟩

lemma *approx-binE*:
assumes *ie*: *interval-extension2 F f*
assumes *w*: *w ∈ aform-err e (W', errw)*
assumes *x*: *x ∈ aform-err e (X', errx)*
assumes *ab*: *approx-bin p F ((W', errw)) ((X', errx)) = Some Ye*
assumes *e*: $e \in UNIV \to \{-1 .. 1\}$
shows *f w x ∈ aform-err e Ye*
⟨*proof*⟩

definition *min-aform-err p a1 (a2*::*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*::*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* :: *nat* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *aform-err* ⇒ *aform-err* **where**
affine-unop p a b d X = (*let*
((*x*, *xs*), *xe*) = *X*;
(*ax*, *axe*) = *trunc-bound-eucl p* (*a* ∗ *x*);
(*y*, *ye*) = *trunc-bound-eucl p* (*ax* + *b*);
(*ys*, *yse*) = *trunc-bound-pdevs p* (*scaleR-pdevs a xs*)
in ((*y*, *ys*), *sum-list′ p* [*truncate-up p* (|*a*| ∗ *xe*), *axe*, *ye*, *yse*, *d*]))
— TODO: also do binop

lemma *aform-err-leI*:
y ∈ *aform-err e* (*c*, *d*)
if *y* ∈ *aform-err e* (*c*, *d′*) *d′* ≤ *d*
⟨*proof*⟩

lemma *aform-err-eqI*:
y ∈ *aform-err e* (*c*, *d*)
if *y* ∈ *aform-err e* (*c*, *d′*) *d′* = *d*
⟨*proof*⟩

lemma *sum-list′-append*[*simp*]: *sum-list′ p* (*ds*@[*d*]) = *truncate-up p* (*d* + *sum-list′ p ds*)
⟨*proof*⟩

lemma *aform-err-sum-list′*:
y ∈ *aform-err e* (*c*, *sum-list′ p ds*)
if *y* ∈ *aform-err e* (*c*, *sum-list ds*)
⟨*proof*⟩

lemma *aform-err-trunc-bound-eucl*:
y ∈ *aform-err e* ((*fst* (*trunc-bound-eucl p X*), *xs*), *snd* (*trunc-bound-eucl p X*) + *d*)
if *y*: *y* ∈ *aform-err e* ((*X*, *xs*), *d*)
⟨*proof*⟩

lemma *trunc-err-pdevsE*:
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
obtains *err* **where**
|*err*| ≤ *tdev′ p* (*trunc-err-pdevs p xs*)
pdevs-val e (*trunc-pdevs p xs*) = *pdevs-val e xs* + *err*
⟨*proof*⟩

lemma *aform-err-trunc-bound-pdevsI*:
y ∈ *aform-err e* ((*c*, *fst* (*trunc-bound-pdevs p xs*)), *snd* (*trunc-bound-pdevs p xs*)

$+\ d$)
  if *y*: $y \in$ *aform-err e* ((*c*, *xs*), *d*)
  and *e*: $e \in UNIV \to \{-1\ ..\ 1\}$
  ⟨*proof*⟩

lemma *aform-err-addI*:
  $y \in$ *aform-err e* ((*a* + *b*, *xs*), *d*)
  if $y - b \in$ *aform-err e* ((*a*, *xs*), *d*)
  ⟨*proof*⟩

theorem *affine-unop*:
  assumes *x*: $x \in$ *aform-err e X*
  assumes *f*: $|f\ x - (a * x + b)| \le d$
    and *e*: $e \in UNIV \to \{-1\ ..\ 1\}$
  shows $f\ x \in$ *aform-err e* (*affine-unop p a b d X*)
⟨*proof*⟩

lemma *min-range-coeffs-ge*:
  $|f\ x - (a * x + b)| \le d$
  if *l*: $l \le x$ **and** *u*: $x \le u$
    and *f'*: $\bigwedge y.\ y \in \{l\ ..\ u\} \Longrightarrow (f$ *has-real-derivative* $f'\ y)$ (*at y*)
    and *a*: $\bigwedge y.\ y \in \{l..u\} \Longrightarrow a \le f'\ y$
    and *d*: $d \ge (f\ u - f\ l - a * (u - l))\ /\ 2 + |(f\ l + f\ u - a * (l + u))\ /\ 2 - b|$
  for *a b d*::*real*
⟨*proof*⟩

lemma *min-range-coeffs-le*:
  $|f\ x - (a * x + b)| \le d$
  if *l*: $l \le x$ **and** *u*: $x \le u$
    and *f'*: $\bigwedge y.\ y \in \{l\ ..\ u\} \Longrightarrow (f$ *has-real-derivative* $f'\ y)$ (*at y*)
    and *a*: $\bigwedge y.\ y \in \{l\ ..\ u\} \Longrightarrow f'\ y \le a$
    and *d*: $d \ge (f\ l - f\ u + a * (u - l))\ /\ 2 + |(f\ l + f\ u - a * (l + u))\ /\ 2 - b|$
  for *a b d*::*real*
⟨*proof*⟩

context includes *floatarith-notation* **begin**

definition *range-reducer p l* =
  (*if* $l < 0 \vee l > 2 *$ *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** $n * (2 * pi) \in_r ivl$
⟨*proof*⟩

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 ∈ UNIV → {−1 .. 1}*
assumes *x*: *x ∈ aform-err e X*
assumes *range-reduce-aform-err p X = Some Y*
obtains *n*::*int* **where** *x + n ∗ (2 ∗ pi) ∈ aform-err e Y*
⟨*proof*⟩

definition *min-range-mono p F DF l u X = do {*
let L = Num l;
let U = Num u;
aivl ← approx p (Min (DF L) (DF U)) [];
let a = lower aivl;
let A = Num a;
bivl ← approx p (Half (F L + F U − A ∗ (L + U))) [];
let (b, be) = mid-err bivl;
let (B, Be) = (Num (float-of b), Num (float-of be));
divl ← approx p ((Half (F U − F L − A ∗ (U − L))) + Be) [];
Some (affine-unop p a b (real-of-float (upper divl)) X)
}

lemma *min-range-mono*:
assumes *x*: *x ∈ aform-err e X*
assumes *l ≤ x x ≤ u*
assumes *min-range-mono p F DF l u X = Some Y*
assumes *e*: *e ∈ UNIV → {−1 .. 1}*
assumes *F*: *⋀x. x ∈ {real-of-float l .. u} ⟹ interpret-floatarith (F (Num x)) []* *= f x*
assumes *DF*: *⋀x. x ∈ {real-of-float l .. u} ⟹ interpret-floatarith (DF (Num* *x)) [] = f′ x*
assumes *f′*: *⋀x. x ∈ {real-of-float l .. u} ⟹ (f has-real-derivative f′ x) (at x)*
assumes *f′-le*: *⋀x. x ∈ {real-of-float l .. u} ⟹ min (f′ l) (f′ u) ≤ f′ x*
shows *f x ∈ aform-err e Y*
⟨*proof*⟩

definition *min-range-antimono p F DF l u X = do {*
let L = Num l;
let U = Num u;
aivl ← approx p (Max (DF L) (DF U)) [];
let a = upper aivl;
let A = Num a;
bivl ← approx p (Half (F L + F U − A ∗ (L + U))) [];
let (b, be) = mid-err bivl;
let (B, Be) = (Num (float-of b), Num (float-of be));
divl ← approx p (Add (Half (F L − F U + A ∗ (U − L))) Be) [];

```
  Some (affine-unop p a b (real-of-float (upper divl)) X)
}

lemma min-range-antimono:
  assumes x: x ∈ aform-err e X
  assumes l ≤ x x ≤ u
  assumes min-range-antimono p F DF l u X = Some Y
  assumes e: e ∈ UNIV → {-1 .. 1}
  assumes F: ⋀x. x ∈ {real-of-float l .. u} ⟹ interpret-floatarith (F (Num x)) []
= f x
  assumes DF: ⋀x. x ∈ {real-of-float l .. u} ⟹ interpret-floatarith (DF (Num
x)) [] = f' x
  assumes f': ⋀x. x ∈ {real-of-float l .. u} ⟹ (f has-real-derivative f' x) (at x)
  assumes f'-le: ⋀x. x ∈ {real-of-float l .. u} ⟹ f' x ≤ max (f' l) (f' u)
  shows f x ∈ aform-err e Y
⟨proof⟩

definition cos-aform-err p X = do {
  X ← 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 ≥ 0 ∧ u ≤ lb-pi p then
   min-range-antimono p Cos (λx. (Minus (Sin x))) l u X
  else if l ≥ ub-pi p ∧ u ≤ 2 * lb-pi p then
   min-range-mono p Cos (λx. (Minus (Sin x))) l u X
  else do {
    Some (ivl-err (real-interval (cos-float-interval p ivl)))
  }
}

lemma abs-half-enclosure:
  fixes r::real
  assumes bl ≤ r r ≤ bu
  shows |r - (bl + bu) / 2| ≤ (bu - bl) / 2
  ⟨proof⟩

lemma cos-aform-err:
  assumes x: x ∈ aform-err e X0
  assumes cos-aform-err p X0 = Some Y
  assumes e: e ∈ UNIV → {-1 .. 1}
  shows cos x ∈ aform-err e Y
⟨proof⟩

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 (λx. 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 ∈ aform-err e X*
assumes *sqrt-aform-err p X = Some Y*
assumes *e*: *e ∈ UNIV → {−1 .. 1}*
shows *sqrt x ∈ aform-err e Y*
⟨*proof*⟩

definition *ln-aform-err p X = do {*
let ivl = ivl-of-aform-err p X;
let l = lower ivl;
if 0 < l then min-range-mono p Ln inverse l (upper ivl) X
else None
}

lemma *ln-aform-err*:
assumes *x*: *x ∈ aform-err e X*
assumes *ln-aform-err p X = Some Y*
assumes *e*: *e ∈ UNIV → {−1 .. 1}*
shows *ln x ∈ aform-err e Y*
⟨*proof*⟩

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*⟩

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 ⟹ b ≥ 0 ⟹ a + b ≠ 0*
for *a b*::*real*
⟨*proof*⟩

lemma *arctan-aform-err*:

assumes x: $x \in$ *aform-err e X*
assumes *arctan-aform-err p X* = *Some Y*
assumes e: $e \in$ *UNIV* $\to \{-1 \,..\, 1\}$
shows *arctan x* $\in$ *aform-err e Y*
⟨*proof*⟩

5.6 Power, TODO: compare with Min-range approximation?!

definition *power-aform-err p* (*X*::*aform-err*) *n* =
(*if n* = *0 then* ((*1*, *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*) +
(*Sum*$_e$ (λ*k*. *Num* (*of-nat* (*n choose k*)) ∗ *Abs* (*Num x0*) $\hat{}_e$ (*n* − *k*) ∗ (*Num xe* + *Num* (*float-of t*)) $\hat{}_e$ *k*)
[*2*..<*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* [] []
⟨*proof*⟩

lemma *plain-floatarith-approx*:
assumes *plain-floatarith 0 f*
shows *interpret-floatarith f* [] $\in_r$ (*the* (*approx p f* []))
⟨*proof*⟩

lemma *plain-floatarith-Sum*$_e$:
plain-floatarith n (*Sum*$_e$ *f xs*) ⟷ *list-all* (λ*i*. *plain-floatarith n* (*f i*)) *xs*
⟨*proof*⟩

lemma *sum-list′-float*[*simp*]: *sum-list′ p xs* $\in$ *float*
⟨*proof*⟩

lemma *tdev′-float*[*simp*]: *tdev′ p xs* $\in$ *float*
⟨*proof*⟩

lemma
  fixes *x y*::*real*
  assumes *abs* $(x - y) \leq e$
  obtains *err* **where** $x = y + err$ *abs err* $\leq e$
  ⟨*proof*⟩

theorem *power-aform-err*:
  assumes $x \in$ *aform-err e X*
  assumes *floats*[*simp*]: *fst* (*fst X*) ∈ *float snd X* ∈ *float*
  assumes *e*: $e \in$ *UNIV* $\to \{-1 \,..\, 1\}$
  shows $x \,\hat{}\, n \in$ *aform-err e* (*power-aform-err p X n*)
⟨*proof*⟩

definition [*code-abbrev*]: *is-float r* ⟷ *r* ∈ *float*
lemma [*code*]: *is-float* (*real-of-float f*) = *True*
  ⟨*proof*⟩

definition *powr-aform-err p X A* = (
    if Inf-aform-err p X > *0 then do* {
      L ← *ln-aform-err p X*;
      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*)
  ⟨*proof*⟩

theorem *powr-aform-err*:
  assumes *x*: $x \in$ *aform-err e X*
  assumes *a*: $a \in$ *aform-err e A*
  assumes *e*: $e \in$ *UNIV* $\to \{-1 \,..\, 1\}$
  assumes *Y*: *powr-aform-err p X A* = *Some Y*
  shows *x powr a* ∈ *aform-err e Y*
⟨*proof*⟩

fun
  approx-floatarith :: *nat* ⇒ *floatarith* ⇒ *aform-err list* ⇒ (*aform-err*) *option*
where
  approx-floatarith p (*Add a b*) *vs* =
    do {
      a1 ← *approx-floatarith p a vs*;
      a2 ← *approx-floatarith p b vs*;
      Some (*add-aform' p a1 a2*)
    }
| *approx-floatarith p* (*Mult a b*) *vs* =
    do {
      a1 ← *approx-floatarith p a vs*;
      a2 ← *approx-floatarith p b vs*;
      Some (*mult-aform' p a1 a2*)

```
    }
| approx-floatarith p (Inverse a) vs =
    do {
      a ← 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 ← 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-aform (uminus-aform z) = (z::'a::real-vector
aform)
  ⟨proof⟩

lemma degree-aform-inverse-aform':
  degree-aform X ≤ n ⟹ degree-aform (fst (inverse-aform' p X)) ≤ n
  ⟨proof⟩

lemma degree-aform-inverse-aform:
  assumes inverse-aform p X = Some Y
  assumes degree-aform X ≤ n
  shows degree-aform (fst Y) ≤ n
  ⟨proof⟩

lemma degree-aform-ivl-err[simp]: degree-aform (fst (ivl-err a)) = 0
  ⟨proof⟩

lemma degree-aform-approx-bin:
  assumes approx-bin p ivl X Y = Some Z
  assumes degree-aform (fst X) ≤ m
  assumes degree-aform (fst Y) ≤ m
```

shows *degree-aform* (*fst Z*) ≤ *m*
⟨*proof*⟩

lemma *degree-aform-approx-un*:
assumes *approx-un p ivl X* = *Some Y*
assumes *case X of None* ⇒ *True* | *Some X* ⇒ *degree-aform* (*fst X*) ≤ *d1*
shows *degree-aform* (*fst Y*) ≤ *d1*
⟨*proof*⟩

lemma *degree-aform-num-aform*[*simp*]: *degree-aform* (*num-aform x*) = *0*
⟨*proof*⟩

lemma *degree-max-aform*:
assumes *degree-aform-err x* ≤ *d*
assumes *degree-aform-err y* ≤ *d*
shows *degree-aform-err* (*max-aform-err p x y*) ≤ *d*
⟨*proof*⟩

lemma *degree-min-aform*:
assumes *degree-aform-err x* ≤ *d*
assumes *degree-aform-err y* ≤ *d*
shows *degree-aform-err* ((*min-aform-err p x y*)) ≤ *d*
⟨*proof*⟩

lemma *degree-aform-acc-err*:
degree-aform (*fst* (*acc-err p X e*)) ≤ *d*
if *degree-aform* (*fst X*) ≤ *d*
⟨*proof*⟩

lemma *degree-pdev-upd-degree*:
assumes *degree b* ≤ *Suc n*
assumes *degree b* ≤ *Suc* (*degree-aform-err X*)
assumes *degree-aform-err X* ≤ *n*
shows *degree* (*pdev-upd b* (*degree-aform-err X*) *0*) ≤ *n*
⟨*proof*⟩

lemma *degree-aform-err-inverse-aform-err*:
assumes *inverse-aform-err p X* = *Some Y*
assumes *degree-aform-err X* ≤ *n*
shows *degree-aform-err Y* ≤ *n*
⟨*proof*⟩

lemma *degree-aform-err-affine-unop*:
degree-aform-err (*affine-unop p a b d X*) ≤ *n*
if *degree-aform-err X* ≤ *n*
⟨*proof*⟩

lemma *degree-aform-err-min-range-mono*:

assumes *min-range-mono p F D l u X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *degree-aform-err-min-range-antimono*:
assumes *min-range-antimono p F D l u X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *degree-aform-err-cos-aform-err*:
assumes *cos-aform-err p X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *degree-aform-err-sqrt-aform-err*:
assumes *sqrt-aform-err p X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *degree-aform-err-arctan-aform-err*:
assumes *arctan-aform-err p X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *degree-aform-err-exp-aform-err*:
assumes *exp-aform-err p X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *degree-aform-err-ln-aform-err*:
assumes *ln-aform-err p X = Some Y*
assumes *degree-aform-err X* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *degree-aform-err-powr-aform-err*:
assumes *powr-aform-err p X Z = Some Y*
assumes *degree-aform-err X* $\leq$ *n*

assumes *degree-aform-err Z* $\leq$ *n*
shows *degree-aform-err Y* $\leq$ *n*
⟨*proof*⟩

lemma *approx-floatarith-degree*:
assumes *approx-floatarith p ra VS = Some X*
assumes $\bigwedge V.$ *V* $\in$ *set VS* $\Longrightarrow$ *degree-aform-err V* $\leq$ *d*
shows *degree-aform-err X* $\leq$ *d*
⟨*proof*⟩

definition *affine-extension2* **where**
affine-extension2 fnctn-aff fnctn $\longleftrightarrow$ (
$\forall$ *d a1 a2 X e2*.
fnctn-aff d a1 a2 = Some X $\longrightarrow$
e2 $\in$ *UNIV* $\rightarrow$ {$-$ *1..1*} $\longrightarrow$
d $\geq$ *degree-aform a1* $\longrightarrow$
d $\geq$ *degree-aform a2* $\longrightarrow$
($\exists$ *e3* $\in$ *UNIV* $\rightarrow$ {$-$ *1..1*}.
(*fnctn* (*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 fnctn-aff fnctn*
assumes *fnctn-aff d a1 a2 = Some X*
e $\in$ *UNIV* $\rightarrow$ {$-$ *1..1*}
d $\geq$ *degree-aform a1*
d $\geq$ *degree-aform a2*
obtains *e′* **where** *e′* $\in$ *UNIV* $\rightarrow$ {$-$ *1..1*}
fnctn (*aform-val e a1*) (*aform-val e a2*) = *aform-val e′ X*
$\bigwedge$*n*. *n* $<$ *d* $\Longrightarrow$ *e′ n* = *e n*
aform-val e a1 = aform-val e′ a1
aform-val e a2 = aform-val e′ a2
⟨*proof*⟩

lemma *aform-err-uminus-aform*:
$-$ *x* $\in$ *aform-err e* (*uminus-aform X*, *ba*)
if *e* $\in$ *UNIV* $\rightarrow$ {-1 .. *1*} *x* $\in$ *aform-err e* (*X*, *ba*)
⟨*proof*⟩

definition *aforms-err e* (*xs*::*aform-err list*) = *listset* (*map* (*aform-err e*) *xs*)

lemma *aforms-err-Nil*[*simp*]: *aforms-err e* [] = {[]}
and *aforms-err-Cons*: *aforms-err e* (*x*#*xs*) = *set-Cons* (*aform-err e x*) (*aforms-err e xs*)
⟨*proof*⟩

lemma *in-set-ConsI*: *a*#*b* $\in$ *set-Cons A B*
if *a* $\in$ *A* **and** *b* $\in$ *B*

⟨*proof*⟩

lemma *mem-aforms-err-Cons-iff*[*simp*]: $x\#xs \in \textit{aforms-err}\ e\ (X\#XS) \longleftrightarrow x \in \textit{aform-err}\ e\ X \wedge xs \in \textit{aforms-err}\ e\ XS$
⟨*proof*⟩

lemma *mem-aforms-err-Cons-iff-Ex-conv*: $x \in \textit{aforms-err}\ e\ (X\#XS) \longleftrightarrow (\exists\, y\ ys.\ x = y\#ys \wedge y \in \textit{aform-err}\ e\ X \wedge ys \in \textit{aforms-err}\ e\ XS)$
⟨*proof*⟩

lemma *listset-Cons-mem-conv*:
$a\ \#\ vs \in \textit{listset}\ AVS \longleftrightarrow (\exists\, A\ VS.\ AVS = A\ \#\ VS \wedge a \in A \wedge vs \in \textit{listset}\ VS)$
⟨*proof*⟩

lemma *listset-Nil-mem-conv*[*simp*]:
$[] \in \textit{listset}\ AVS \longleftrightarrow AVS = []$
⟨*proof*⟩

lemma *listset-nthD*: $vs \in \textit{listset}\ VS \Longrightarrow i < \textit{length}\ vs \Longrightarrow vs\ !\ i \in VS\ !\ i$
⟨*proof*⟩

lemma *length-listsetD*:
$vs \in \textit{listset}\ VS \Longrightarrow \textit{length}\ vs = \textit{length}\ VS$
⟨*proof*⟩

lemma *length-aforms-errD*:
$vs \in \textit{aforms-err}\ e\ VS \Longrightarrow \textit{length}\ vs = \textit{length}\ VS$
⟨*proof*⟩

lemma *nth-aforms-errI*:
$vs\ !\ i \in \textit{aform-err}\ e\ (VS\ !\ i)$
if $vs \in \textit{aforms-err}\ e\ VS\ i < \textit{length}\ vs$
⟨*proof*⟩

lemma *eucl-truncate-down-float*[*simp*]: $\textit{eucl-truncate-down}\ p\ x \in \textit{float}$
⟨*proof*⟩

lemma *eucl-truncate-up-float*[*simp*]: $\textit{eucl-truncate-up}\ p\ x \in \textit{float}$
⟨*proof*⟩

lemma *trunc-bound-eucl-float*[*simp*]: $\textit{fst}\ (\textit{trunc-bound-eucl}\ p\ x) \in \textit{float}$
$\textit{snd}\ (\textit{trunc-bound-eucl}\ p\ x) \in \textit{float}$
⟨*proof*⟩

lemma *add-aform'-float*:
$\textit{add-aform'}\ p\ x\ y = ((a,\ b),\ ba) \Longrightarrow a \in \textit{float}$
$\textit{add-aform'}\ p\ x\ y = ((a,\ b),\ ba) \Longrightarrow ba \in \textit{float}$
⟨*proof*⟩

lemma *uminus-aform-float*: *uminus-aform* (*aa*, *bb*) = (*a*, *b*) ⟹ *aa* ∈ *float* ⟹ *a* ∈ *float*
⟨*proof*⟩

lemma *mult-aform'-float*: *mult-aform'* *p* *x* *y* = ((*a*, *b*), *ba*) ⟹ *a* ∈ *float*
mult-aform' *p* *x* *y* = ((*a*, *b*), *ba*) ⟹ *ba* ∈ *float*
⟨*proof*⟩

lemma *inverse-aform'-float*: *inverse-aform'* *p* *x* = ((*a*, *bb*), *baa*) ⟹ *a* ∈ *float*
⟨*proof*⟩

lemma *inverse-aform-float*:
inverse-aform *p* *x* = *Some* ((*a*, *bb*), *baa*) ⟹ *a* ∈ *float*
⟨*proof*⟩

lemma *inverse-aform-err-float*: *inverse-aform-err* *p* *x* = *Some* ((*a*, *b*), *ba*) ⟹ *a* ∈ *float*
inverse-aform-err *p* *x* = *Some* ((*a*, *b*), *ba*) ⟹ *ba* ∈ *float*
⟨*proof*⟩

lemma *affine-unop-float*:
affine-unop *p* *asdf* *aaa* *bba* *h* = ((*a*, *b*), *ba*) ⟹ *a* ∈ *float*
affine-unop *p* *asdf* *aaa* *bba* *h* = ((*a*, *b*), *ba*) ⟹ *ba* ∈ *float*
⟨*proof*⟩

lemma *min-range-antimono-float*:
min-range-antimono *p* *f* *f'* *i* *g* *h* = *Some* ((*a*, *b*), *ba*) ⟹ *a* ∈ *float*
min-range-antimono *p* *f* *f'* *i* *g* *h* = *Some* ((*a*, *b*), *ba*) ⟹ *ba* ∈ *float*
⟨*proof*⟩

lemma *min-range-mono-float*:
min-range-mono *p* *f* *f'* *i* *g* *h* = *Some* ((*a*, *b*), *ba*) ⟹ *a* ∈ *float*
min-range-mono *p* *f* *f'* *i* *g* *h* = *Some* ((*a*, *b*), *ba*) ⟹ *ba* ∈ *float*
⟨*proof*⟩

lemma *in-float-timesI*: *a* ∈ *float* **if** *b* = *a* ∗ *2* *b* ∈ *float*
⟨*proof*⟩

lemma *interval-extension-floor*: *interval-extension1* (λ*ivl*. *Some* (*floor-float-interval* *ivl*)) *floor*
⟨*proof*⟩

lemma *approx-floatarith-Elem*:
assumes *approx-floatarith* *p* *ra* *VS* = *Some* *X*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *vs* ∈ *aforms-err* *e* *VS*
shows *interpret-floatarith* *ra* *vs* ∈ *aform-err* *e* *X*
⟨*proof*⟩

primrec *approx-floatariths-aformerr* ::
nat ⇒ *floatarith list* ⇒ *aform-err list* ⇒ *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 ∈ UNIV → {−1 .. 1}*
assumes *approx-floatariths-aformerr p ra VS = Some X*
assumes *vs ∈ aforms-err e VS*
shows *interpret-floatariths ra vs ∈ aforms-err e X*
⟨*proof*⟩

lemma *fold-max-mono*:
fixes *x*::*'a*::*linorder*
shows *x ≤ y ⟹ fold max zs x ≤ fold max zs y*
⟨*proof*⟩

lemma *fold-max-le-self*:
fixes *y*::*'a*::*linorder*
shows *y ≤ fold max ys y*
⟨*proof*⟩

lemma *fold-max-le*:
fixes *x*::*'a*::*linorder*
shows *x ∈ set xs ⟹ x ≤ fold max xs z*
⟨*proof*⟩

abbreviation *degree-aforms-err ≡ degrees o map (snd o fst)*

definition *aforms-err-to-aforms d xs =*
(map (λ(d, x). aform-err-to-aform x d) (zip [d..<d + length xs] xs))

lemma *aform-vals-empty*[*simp*]: *aform-vals e' [] = []*
⟨*proof*⟩
lemma *aforms-err-to-aforms-Nil*[*simp*]: *(aforms-err-to-aforms n []) = []*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *degree-aform-err-to-aform-le*:
degree-aform (*aform-err-to-aform X n*) ≤ *max* (*degree-aform-err X*) (*Suc n*)
⟨*proof*⟩

lemma *less-degree-aform-aform-err-to-aformD*: *i* < *degree-aform* (*aform-err-to-aform X n*) ⟹ *i* < *max* (*Suc n*) (*degree-aform-err X*)
⟨*proof*⟩

lemma *pdevs-domain-aform-err-to-aform*:
pdevs-domain (*snd* (*aform-err-to-aform X n*)) = *pdevs-domain* (*snd* (*fst X*)) ∪ (*if snd X* = *0 then* {} *else* {*n*})
if *n* ≥ *degree-aform-err X*
⟨*proof*⟩

lemma *length-aforms-err-to-aforms*[*simp*]: *length* (*aforms-err-to-aforms i XS*) = *length XS*
⟨*proof*⟩

lemma *aforms-err-to-aforms-ex*:
assumes *X*: *x* ∈ *aforms-err e X*
assumes *deg*: *degree-aforms-err X* ≤ *n*
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
shows ∃ *e'*∈ *UNIV* → {−*1* .. *1*}. *x* = *aform-vals e'* (*aforms-err-to-aforms n X*) ∧
(∀ *i* < *n*. *e' i* = *e i*)
⟨*proof*⟩

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*
⟨*proof*⟩

definition *approx-floatariths p ea as* =
do {
let da = (*degree-aforms as*);
let aes = (*map* (λ*x*. (*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* (λ*x*. {*f x*}) *as*) = {*map f as*}
⟨*proof*⟩

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*⟩

lemma *length-eq-NilI*: *length* [] = *length* []
and *length-eq-ConsI*: *length xs* = *length ys* ⟹ *length* (*x*#*xs*) = *length* (*y*#*ys*)
⟨*proof*⟩

5.7 Generic operations on Affine Forms in Euclidean Space

lemma *pdevs-val-domain-cong*:
assumes *b* = *d*
assumes ⋀*i*. *i* ∈ *pdevs-domain b* ⟹ *a i* = *c i*
shows *pdevs-val a b* = *pdevs-val c d*
⟨*proof*⟩

lemma *fresh-JointsI*:
assumes *xs* ∈ *Joints XS*
assumes *list-all* (λ*Y*. *pdevs-domain* (*snd X*) ∩ *pdevs-domain* (*snd Y*) = {}) *XS*
assumes *x* ∈ *Affine X*
shows *x*#*xs* ∈ *Joints* (*X*#*XS*)
⟨*proof*⟩

primrec *approx-slp*::*nat* ⇒ *slp* ⇒ *aform-err list* ⇒ *aform-err list option*
where
approx-slp p [] *xs* = *Some xs*
| *approx-slp p* (*ea* # *eas*) *xs* =
do {
r ← *approx-floatarith p ea xs*;
approx-slp p eas (*r*#*xs*)
}

lemma *Nil-mem-Joints*[*intro*, *simp*]: [] ∈ *Joints* []
⟨*proof*⟩

lemma *map-nth-Joints*: *xs* ∈ *Joints XS* ⟹ (⋀*i*. *i* ∈ *set is* ⟹ *i* < *length XS*)
⟹ *map* (*nth xs*) *is* @ *xs* ∈ *Joints* (*map* (*nth XS*) *is* @ *XS*)
⟨*proof*⟩

lemma *map-nth-Joints'*: *xs* ∈ *Joints XS* ⟹ (⋀*i*. *i* ∈ *set is* ⟹ *i* < *length XS*)
⟹ *map* (*nth xs*) *is* ∈ *Joints* (*map* (*nth XS*) *is*)
⟨*proof*⟩

lemma *approx-slp-Elem*:
assumes *e*: *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *vs* ∈ *aforms-err e VS*

assumes *approx-slp p ra VS = Some X*
shows *interpret-slp ra vs ∈ aforms-err e X*
⟨*proof*⟩

definition *approx-slp-outer p n slp XS =*
do {
let d = degree-aforms XS;
let XSe = (map (λx. (x, 0)) XS);
rs ← approx-slp p slp XSe;
let rs′ = take n rs;
let d′ = max d (degree-aforms-err rs′);
Some (aforms-err-to-aforms d′ rs′)
}

lemma *take-in-listsetI*: *xs ∈ listset XS ⟹ take n xs ∈ listset (take n XS)*
⟨*proof*⟩

lemma *take-in-aforms-errI*: *take n xs ∈ aforms-err e (take n XS)*
if *xs ∈ aforms-err e XS*
⟨*proof*⟩

theorem *approx-slp-outer*:
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 @ xs ∈ Joints (RS @ XS)*
⟨*proof*⟩

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*⟩

end

end

6 Counterclockwise

theory *Counterclockwise*
imports *HOL−Analysis.Multivariate-Analysis*
begin

6.1 Auxiliary Lemmas

lemma *convex3-alt*:

fixes $x\ y\ z::'a::real\text{-}vector$
assumes $0 \leq a\ 0 \leq b\ 0 \leq c\ a + b + c = 1$
obtains $u\ v$ **where** $a *_R x + b *_R y + c *_R z = x + u *_R (y - x) + v *_R (z - x)$
and $0 \leq u\ 0 \leq v\ u + v \leq 1$
⟨*proof*⟩

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*⟩

lemma *sum-nonpos-eq-0-iff*:
fixes $f :: 'a \Rightarrow 'b::ordered\text{-}ab\text{-}group\text{-}add$
shows $[\![finite\ A; \forall x{\in}A.\ f\ x \leq 0]\!] \Longrightarrow sum\ f\ A = 0 \longleftrightarrow (\forall x{\in}A.\ f\ x = 0)$
⟨*proof*⟩

lemma *fold-if-in-set*:
$fold\ (\lambda x\ m.\ if\ P\ x\ m\ then\ x\ else\ m)\ xs\ x \in set\ (x\#xs)$
⟨*proof*⟩

6.2 Sort Elements of a List

locale *linorder-list0* = **fixes** $le::'a \Rightarrow 'a \Rightarrow bool$
begin

definition *min-for a b* = (*if le a b then a else b*)

lemma *min-for-in*[*simp*]: $x \in S \Longrightarrow y \in S \Longrightarrow min\text{-}for\ x\ y \in S$
⟨*proof*⟩

lemma *fold-min-eqI1*: $fold\ min\text{-}for\ ys\ y \notin set\ ys \Longrightarrow fold\ min\text{-}for\ ys\ y = y$
⟨*proof*⟩

function *selsort* **where**
$selsort\ [] = []$
| $selsort\ (y\#ys) = (let$
$xm = fold\ min\text{-}for\ ys\ y;$
$xs' = List.remove1\ xm\ (y\#ys)$
$in\ (xm\#selsort\ xs'))$
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *in-set-selsort-eq*: $x \in set\ (selsort\ xs) \longleftrightarrow x \in (set\ xs)$
⟨*proof*⟩

lemma *set-selsort*[*simp*]: $set\ (selsort\ xs) = set\ xs$
⟨*proof*⟩

lemma *length-selsort*[*simp*]: *length* (*selsort xs*) = *length xs*
⟨*proof*⟩

lemma *distinct-selsort*[*simp*]: *distinct* (*selsort xs*) = *distinct xs*
⟨*proof*⟩

lemma *selsort-eq-empty-iff*[*simp*]: *selsort xs* = [] ⟷ *xs* = []
⟨*proof*⟩

inductive *sortedP* :: *'a list* ⇒ *bool* **where**
Nil: *sortedP* []
| *Cons*: ∀ *y*∈*set ys*. *le x y* ⟹ *sortedP ys* ⟹ *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* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *le x y*))
⟨*proof*⟩

lemma *sortedP-appendI*:
sortedP xs ⟹ *sortedP ys* ⟹ (⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *le x y*) ⟹ *sortedP* (*xs* @ *ys*)
⟨*proof*⟩

lemma *sorted-nth-less*: *sortedP xs* ⟹ *i* < *j* ⟹ *j* < *length xs* ⟹ *le* (*xs* ! *i*) (*xs* ! *j*)
⟨*proof*⟩

lemma *sorted-butlastI*[*intro*, *simp*]: *sortedP xs* ⟹ *sortedP* (*butlast xs*)
⟨*proof*⟩

lemma *sortedP-right-of-append1*:
assumes *sortedP* (*zs*@[*z*])
assumes *y* ∈ *set zs*
shows *le y z*
⟨*proof*⟩

lemma *sortedP-right-of-last*:
assumes *sortedP zs*
assumes *y* ∈ *set zs y* ≠ *last zs*
shows *le y* (*last zs*)

⟨*proof*⟩

lemma *selsort-singleton-iff*: *selsort xs* = [*x*] ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *hd-last-sorted*:
assumes *sortedP xs length xs* > *1*
shows *le* (*hd xs*) (*last xs*)
⟨*proof*⟩

end

lemma (**in** *comm-monoid-add*) *sum-list-distinct-selsort*:
assumes *distinct xs*
shows *sum-list* (*linorder-list0.selsort le xs*) = *sum-list xs*
⟨*proof*⟩

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* ⇒ - +
fixes *S*
assumes *order-refl*: *a* ∈ *S* ⟹ *le a a*
assumes *trans*′: *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *c* ∈ *S* ⟹ *a* ≠ *b* ⟹ *b* ≠ *c* ⟹ *a* ≠ *c*
⟹
le a b ⟹ *le b c* ⟹ *le a c*
assumes *antisym*: *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *le a b* ⟹ *le b a* ⟹ *a* = *b*
assumes *linear*′: *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *a* ≠ *b* ⟹ *le a b* ∨ *le b a*
begin

lemma *trans*: *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *c* ∈ *S* ⟹ *le a b* ⟹ *le b c* ⟹ *le a c*
⟨*proof*⟩

lemma *linear*: *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *le a b* ∨ *le b a*
⟨*proof*⟩

lemma *min-le1*: *w* ∈ *S* ⟹ *y* ∈ *S* ⟹ *le* (*min-for w y*) *y*
and *min-le2*: *w* ∈ *S* ⟹ *y* ∈ *S* ⟹ *le* (*min-for w y*) *w*
⟨*proof*⟩

lemma *fold-min*:
assumes *set xs* ⊆ *S*
shows *list-all* (λ*y*. *le* (*fold min-for* (*tl xs*) (*hd xs*)) *y*) *xs*
⟨*proof*⟩

lemma
sortedP-selsort:

assumes *set xs* $\subseteq$ *S*
shows *sortedP* (*selsort xs*)
⟨*proof*⟩

end

6.3 Abstract CCW Systems

locale *ccw-system0* =
fixes *ccw*::'*a* ⇒ '*a* ⇒ '*a* ⇒ *bool*
and *S*::'*a set*
begin

abbreviation *indelta t p q r* ≡ *ccw t q r* ∧ *ccw p t r* ∧ *ccw p q t*
abbreviation *insquare p q r s* ≡ *ccw p q r* ∧ *ccw q r s* ∧ *ccw r s p* ∧ *ccw s p q*

end

abbreviation *distinct3 p q r* ≡ ¬(*p* = *q* ∨ *p* = *r* ∨ *q* = *r*)
abbreviation *distinct4 p q r s* ≡ ¬(*p* = *q* ∨ *p* = *r* ∨ *p* = *s* ∨ ¬ *distinct3 q r s*)
abbreviation *distinct5 p q r s t* ≡ ¬(*p* = *q* ∨ *p* = *r* ∨ *p* = *s* ∨ *p* = *t* ∨ ¬ *distinct4 q r s t*)

abbreviation *in3 S p q r* ≡ *p* ∈ *S* ∧ *q* ∈ *S* ∧ *r* ∈ *S*
abbreviation *in4 S p q r s* ≡ *in3 S p q r* ∧ *s* ∈ *S*
abbreviation *in5 S p q r s t* ≡ *in4 S p q r s* ∧ *t* ∈ *S*

locale *ccw-system12* = *ccw-system0* +
assumes *cyclic*: *ccw p q r* ⟹ *ccw q r p*
assumes *ccw-antisym*: *distinct3 p q r* ⟹ *in3 S p q r* ⟹ *ccw p q r* ⟹ ¬ *ccw p r q*

locale *ccw-system123* = *ccw-system12* +
assumes *nondegenerate*: *distinct3 p q r* ⟹ *in3 S p q r* ⟹ *ccw p q r* ∨ *ccw p r q*
begin

lemma *not-ccw-eq*: *distinct3 p q r* ⟹ *in3 S p q r* ⟹ ¬ *ccw p q r* ⟷ *ccw p r q*
⟨*proof*⟩

end

locale *ccw-system4* = *ccw-system123* +
assumes *interior*:
distinct4 p q r t ⟹ *in4 S p q r t* ⟹ *ccw t q r* ⟹ *ccw p t r* ⟹ *ccw p q t* ⟹ *ccw p q r*
begin

lemma *interior'*:

distinct4 p q r t $\Longrightarrow$ *in4 S p q r t* $\Longrightarrow$ *ccw p q t* $\Longrightarrow$ *ccw q r t* $\Longrightarrow$ *ccw r p t* $\Longrightarrow$ *ccw p q r*
⟨*proof*⟩

end

locale *ccw-system1235′* = *ccw-system123* +
assumes *dual-transitive*:
distinct5 p q r s t $\Longrightarrow$ *in5 S p q r s t* $\Longrightarrow$
ccw s t p $\Longrightarrow$ *ccw s t q* $\Longrightarrow$ *ccw s t r* $\Longrightarrow$ *ccw t p q* $\Longrightarrow$ *ccw t q r* $\Longrightarrow$ *ccw t p r*

locale *ccw-system1235* = *ccw-system123* +
assumes *transitive*: *distinct5 p q r s t* $\Longrightarrow$ *in5 S p q r s t* $\Longrightarrow$
ccw t s p $\Longrightarrow$ *ccw t s q* $\Longrightarrow$ *ccw t s r* $\Longrightarrow$ *ccw t p q* $\Longrightarrow$ *ccw t q r* $\Longrightarrow$ *ccw t p r*
begin

lemmas *ccw-axioms* = *cyclic nondegenerate ccw-antisym transitive*

sublocale *ccw-system1235′*
⟨*proof*⟩

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* $\Rightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *ccw 0 c b* $\Longrightarrow$ *ccw 0* (*a* + *c*) *b*
begin

lemma *translate-plus′*[*simp*]:
ccw (*x* + *a*) (*x* + *b*) (*x* + *c*) $\longleftrightarrow$ *ccw a b c*
⟨*proof*⟩

lemma *uminus2*[*simp*]: *ccw 0 a* (− *b*) = *ccw 0 b a*
⟨*proof*⟩

lemma *uminus-all*[*simp*]: *ccw* (−*a*) (−*b*) (−*c*) $\longleftrightarrow$ *ccw a b c*

⟨*proof*⟩

lemma *translate-origin*: *NO-MATCH 0 p* ⟹ *ccw p q r* ⟷ *ccw 0* (*q* − *p*) (*r* − *p*)
⟨*proof*⟩

lemma *translate*[*simp*]: *ccw a* (*a* + *b*) (*a* + *c*) ⟷ *ccw 0 b c*
⟨*proof*⟩

lemma *translate-plus3*: *ccw* (*a* − *x*) (*b* − *x*) *c* ⟷ *ccw a b* (*c* + *x*)
⟨*proof*⟩

lemma *renormalize*:
ccw 0 (*a* − *b*) (*c* − *a*) ⟹ *ccw b a c*
⟨*proof*⟩

lemma *cyclicI*: *ccw p q r* ⟹ *ccw q r p*
⟨*proof*⟩

lemma
scaleR2-eq[*simp*]:
0 < *e* ⟹ *ccw 0 xr* (*e* ∗*R P*) ⟷ *ccw 0 xr P*
⟨*proof*⟩

lemma *scaleR1-nonzero-eq*:
e ≠ *0* ⟹ *ccw 0* (*e* ∗*R a*) *b* = (*if e* > *0 then ccw 0 a b else ccw 0 b a*)
⟨*proof*⟩

lemma *neg-scaleR*[*simp*]: *x* < *0* ⟹ *ccw 0* (*x* ∗*R b*) *c* ⟷ *ccw 0 c b*
⟨*proof*⟩

lemma
scaleR1:
0 < *e* ⟹ *ccw 0 xr P* ⟹ *ccw 0* (*e* ∗*R xr*) *P*
⟨*proof*⟩

lemma
add3: *ccw 0 a b* ∧ *ccw 0 a c* ⟹ *ccw 0 a* (*b* + *c*)
⟨*proof*⟩

lemma *add3-self*[*simp*]: *ccw 0 p* (*p* + *q*) ⟷ *ccw 0 p q*
⟨*proof*⟩

lemma *add2-self*[*simp*]: *ccw 0* (*p* + *q*) *p* ⟷ *ccw 0 q p*
⟨*proof*⟩

lemma *scale-add3*[*simp*]: *ccw 0 a* (*x* ∗*R a* + *b*) ⟷ *ccw 0 a b*
⟨*proof*⟩

lemma *scale-add3'*[*simp*]: *ccw 0 a* (*b* + *x* $*_R$ *a*) ⟷ *ccw 0 a b*
and *scale-minus3*[*simp*]: *ccw 0 a* (*x* $*_R$ *a* − *b*) ⟷ *ccw 0 b a*
and *scale-minus3'*[*simp*]: *ccw 0 a* (*b* − *x* $*_R$ *a*) ⟷ *ccw 0 a b*
⟨*proof*⟩

lemma *sum*:
assumes *fin*: *finite X*
assumes *ne*: *X*≠{}
assumes *ncoll*: (⋀*x*. *x* ∈ *X* ⟹ *ccw 0 a* (*f x*))
shows *ccw 0 a* (*sum f X*)
⟨*proof*⟩

lemma *sum2*:
assumes *fin*: *finite X*
assumes *ne*: *X*≠{}
assumes *ncoll*: (⋀*x*. *x* ∈ *X* ⟹ *ccw 0* (*f x*) *a*)
shows *ccw 0* (*sum f X*) *a*
⟨*proof*⟩

lemma *translate-minus*[*simp*]:
ccw (*x* − *a*) (*x* − *b*) (*x* − *c*) = *ccw* (−*a*) (−*b*) (−*c*)
⟨*proof*⟩

end

locale *ccw-convex* = *ccw-system ccw S* **for** *ccw* **and** *S*::'*a*::*real-vector set* +
fixes *oriented*
assumes *convex2*:
u ≥ *0* ⟹ *v* ≥ *0* ⟹ *u* + *v* = *1* ⟹ *ccw a b c* ⟹ *ccw a b d* ⟹ *oriented a b* ⟹
ccw a b (*u* $*_R$ *c* + *v* $*_R$ *d*)
begin

lemma *convex-hull*:
assumes [*intro*, *simp*]: *finite C*
assumes *ccw*: ⋀*c*. *c* ∈ *C* ⟹ *ccw a b c*
assumes *ch*: *x* ∈ *convex hull C*
assumes *oriented*: *oriented a b*
shows *ccw a b x*
⟨*proof*⟩

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* ⇒ *point* ⇒ *point* ⇒ *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*
⟨*proof*⟩

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^3^3*)
⟨*proof*⟩

declare *det3.simps*[*simp del*]

lemma *det3-self23*[*simp*]: *det3 a b b* = *0*
and *det3-self12*[*simp*]: *det3 b b a* = *0*
⟨*proof*⟩

lemma
coll-ex-scaling:
assumes *b* ≠ *c*
assumes *d*: *det3 a b c* = *0*
shows ∃ *r*. *a* = *b* + *r* ∗*R* (*c* − *b*)
⟨*proof*⟩

lemma *cramer*: ¬*det3 s t q* = *0* ⟹
(*det3 t p r*) = ((*det3 t q r*) ∗ (*det3 s t p*) + (*det3 t p q*) ∗ (*det3 s t r*))/(*det3 s t q*)
⟨*proof*⟩

lemma *convex-comb-dets*:
assumes *det3 p q r* > *0*
shows *s* = (*det3 s q r* / *det3 p q r*) ∗*R p* + (*det3 p s r* / *det3 p q r*) ∗*R q* +
(*det3 p q s* / *det3 p q r*) ∗*R r*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *four-points-aligned*:
assumes *c*: *det3 t p q* = *0 det3 t q r* = *0*
assumes *distinct*: *distinct5 t s p q r*
shows *det3 t r p* = *0 det3 p q r* = *0*

⟨*proof*⟩

lemma *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*
⟨*proof*⟩

lemma *det3-eq-zeroI*:
assumes *p = q + x *$_R$ (t − q)*
shows *det3 q t p = 0*
⟨*proof*⟩

lemma *det3-rotate*: *det3 a b c = det3 c a b*
⟨*proof*⟩

lemma *det3-switch*: *det3 a b c = − det3 a c b*
⟨*proof*⟩

lemma *det3-switch′*: *det3 a b c = − det3 b a c*
⟨*proof*⟩

lemma *det3-pos-transitive-coll*:
det3 t s p > 0 ⟹ det3 t s r ≥ 0 ⟹ det3 t p q ≥ 0 ⟹
det3 t q r > 0 ⟹ det3 t s q = 0 ⟹ det3 t p r > 0
⟨*proof*⟩

lemma *det3-pos-transitive*:
det3 t s p > 0 ⟹ det3 t s q ≥ 0 ⟹ det3 t s r ≥ 0 ⟹ det3 t p q ≥ 0 ⟹
det3 t q r > 0 ⟹ det3 t p r > 0
⟨*proof*⟩

lemma *det3-zero-translate-plus*[*simp*]: *det3 (a + x) (b + x) (c + x) = 0 ⟷ det3 a b c = 0*
⟨*proof*⟩

lemma *det3-zero-translate-plus′*[*simp*]: *det3 (a) (a + b) (a + c) = 0 ⟷ det3 0 b c = 0*
⟨*proof*⟩

lemma
det30-zero-scaleR1:
*0 < e ⟹ det3 0 xr P = 0 ⟹ det3 0 (e *$_R$ xr) P = 0*
⟨*proof*⟩

lemma *det3-same*[*simp*]: *det3 a x x = 0*
⟨*proof*⟩

lemma
det30-zero-scaleR2:
*0 < e ⟹ det3 0 P xr = 0 ⟹ det3 0 P (e *$_R$ xr) = 0*

⟨*proof*⟩

lemma *det3-eq-zero*: $e \neq 0 \Longrightarrow det3\ 0\ xr\ (e *_R Q) = 0 \longleftrightarrow det3\ 0\ xr\ Q = 0$
⟨*proof*⟩

lemma *det30-plus-scaled3*[*simp*]: $det3\ 0\ a\ (b + x *_R a) = 0 \longleftrightarrow det3\ 0\ a\ b = 0$
⟨*proof*⟩

lemma *det30-plus-scaled2*[*simp*]:
shows $det3\ 0\ (a + x *_R a)\ b = 0 \longleftrightarrow (if\ x = -1\ then\ True\ else\ det3\ 0\ a\ b = 0)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *det30-uminus2*[*simp*]: $det3\ 0\ (-a)\ (b) = 0 \longleftrightarrow det3\ 0\ a\ b = 0$
and *det30-uminus3*[*simp*]: $det3\ 0\ a\ (-b) = 0 \longleftrightarrow det3\ 0\ a\ b = 0$
⟨*proof*⟩

lemma *det30-minus-scaled3*[*simp*]: $det3\ 0\ a\ (b - x *_R a) = 0 \longleftrightarrow det3\ 0\ a\ b = 0$
⟨*proof*⟩

lemma *det30-scaled-minus3*[*simp*]: $det3\ 0\ a\ (e *_R a - b) = 0 \longleftrightarrow det3\ 0\ a\ b = 0$
⟨*proof*⟩

lemma *det30-minus-scaled2*[*simp*]:
$det3\ 0\ (a - x *_R a)\ b = 0 \longleftrightarrow (if\ x = 1\ then\ True\ else\ det3\ 0\ a\ b = 0)$
⟨*proof*⟩

lemma *det3-nonneg-scaleR1*:
$0 < e \Longrightarrow det3\ 0\ xr\ P \geq 0 \Longrightarrow det3\ 0\ (e*_R xr)\ P \geq 0$
⟨*proof*⟩

lemma *det3-nonneg-scaleR1-eq*:
$0 < e \Longrightarrow det3\ 0\ (e*_R xr)\ P \geq 0 \longleftrightarrow det3\ 0\ xr\ P \geq 0$
⟨*proof*⟩

lemma *det3-translate-origin*: $NO\text{-}MATCH\ 0\ p \Longrightarrow det3\ p\ q\ r = det3\ 0\ (q - p)\ (r - p)$
⟨*proof*⟩

lemma *det3-nonneg-scaleR-segment2*:
assumes $det3\ x\ y\ z \geq 0$
assumes $a > 0$
shows $det3\ x\ ((1 - a) *_R x + a *_R y)\ z \geq 0$
⟨*proof*⟩

lemma *det3-nonneg-scaleR-segment1*:
assumes $det3\ x\ y\ z \geq 0$
assumes $0 \leq a\ a < 1$

shows *det3* $((1 - a) *_R x + a *_R y)\ y\ z \geq 0$
⟨*proof*⟩

8.2 Strict CCW Predicate

definition $ccw'\ p\ q\ r \longleftrightarrow 0 < det3\ p\ q\ r$

interpretation *ccw'*: *ccw-vector-space ccw'*
⟨*proof*⟩

interpretation *ccw'*: *linorder-list0 ccw' x* **for** *x* ⟨*proof*⟩

lemma *ccw'-contra*: $ccw'\ t\ r\ q \Longrightarrow ccw'\ t\ q\ r = False$
⟨*proof*⟩

lemma *not-ccw'-eq*: $\neg\ ccw'\ t\ p\ s \longleftrightarrow ccw'\ t\ s\ p \vee det3\ t\ s\ p = 0$
⟨*proof*⟩

lemma *neq-left-right-of*: $ccw'\ a\ b\ c \Longrightarrow ccw'\ a\ c\ d \Longrightarrow b \neq d$
⟨*proof*⟩

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*⟩

lemma *ccw'-sorted-scaleR*: *ccw'.sortedP 0 xs* $\Longrightarrow r > 0 \Longrightarrow$ *ccw'.sortedP 0* (*map* $((*_R)\ r)$ *xs*)
⟨*proof*⟩

8.3 Collinearity

abbreviation $coll\ a\ b\ c \equiv det3\ a\ b\ c = 0$

lemma *coll-zero*[*intro*, *simp*]: *coll 0 z 0*
⟨*proof*⟩

lemma *coll-zero1*[*intro*, *simp*]: *coll 0 0 z*
⟨*proof*⟩

lemma *coll-self*[*intro*, *simp*]: *coll 0 z z*
⟨*proof*⟩

lemma *ccw'-not-coll*:
$ccw'\ a\ b\ c \Longrightarrow \neg coll\ a\ b\ c$
$ccw'\ a\ b\ c \Longrightarrow \neg coll\ a\ c\ b$
$ccw'\ a\ b\ c \Longrightarrow \neg coll\ b\ a\ c$
$ccw'\ a\ b\ c \Longrightarrow \neg coll\ b\ c\ a$

$ccw'\ a\ b\ c \Longrightarrow \neg coll\ c\ a\ b$
$ccw'\ a\ b\ c \Longrightarrow \neg coll\ c\ b\ a$
⟨*proof*⟩

lemma *coll-add*: $coll\ 0\ x\ y \Longrightarrow coll\ 0\ x\ z \Longrightarrow coll\ 0\ x\ (y + z)$
⟨*proof*⟩

lemma *coll-scaleR-left-eq*[*simp*]: $coll\ 0\ (r *_R x)\ y \longleftrightarrow r = 0 \vee coll\ 0\ x\ y$
⟨*proof*⟩

lemma *coll-scaleR-right-eq*[*simp*]: $coll\ 0\ y\ (r *_R x) \longleftrightarrow r = 0 \vee coll\ 0\ y\ x$
⟨*proof*⟩

lemma *coll-scaleR*: $coll\ 0\ x\ y \Longrightarrow coll\ 0\ (r *_R x)\ y$
⟨*proof*⟩

lemma *coll-sum-list*: $(\bigwedge y.\ y \in set\ ys \Longrightarrow coll\ 0\ x\ y) \Longrightarrow coll\ 0\ x\ (sum\text{-}list\ ys)$
⟨*proof*⟩

lemma *scaleR-left-normalize*:
fixes a ::*real* **and** $b\ c$::'a::*real-vector*
shows $a *_R b = c \longleftrightarrow (if\ a = 0\ then\ c = 0\ else\ b = c /_R a)$
⟨*proof*⟩

lemma *coll-scale-pair*: $coll\ 0\ (a,\ b)\ (c,\ d) \Longrightarrow (a,\ b) \neq 0 \Longrightarrow (\exists x.\ (c,\ d) = x *_R (a,\ b))$
⟨*proof*⟩

lemma *coll-scale*: $coll\ 0\ r\ q \Longrightarrow r \neq 0 \Longrightarrow (\exists x.\ q = x *_R r)$
⟨*proof*⟩

lemma *coll-add-trans*:
assumes $coll\ 0\ x\ (y + z)$
assumes $coll\ 0\ y\ z$
assumes $x \neq 0$
assumes $y \neq 0$
assumes $z \neq 0$
assumes $y + z \neq 0$
shows $coll\ 0\ x\ z$
⟨*proof*⟩

lemma *coll-commute*: $coll\ 0\ a\ b \longleftrightarrow coll\ 0\ b\ a$
⟨*proof*⟩

lemma *coll-add-cancel*: $coll\ 0\ a\ (a + b) \Longrightarrow coll\ 0\ a\ b$
⟨*proof*⟩

lemma *coll-trans*:
$coll\ 0\ a\ b \Longrightarrow coll\ 0\ a\ c \Longrightarrow a \neq 0 \Longrightarrow coll\ 0\ b\ c$

⟨*proof*⟩

lemma *sum-list-posI*:
fixes *xs*::*'a*::*ordered-comm-monoid-add list*
shows $(\bigwedge x.\ x \in set\ xs \Longrightarrow x > 0) \Longrightarrow xs \neq [] \Longrightarrow sum\text{-}list\ xs > 0$
⟨*proof*⟩

lemma *nonzero-fstI*[*intro*, *simp*]: $fst\ x \neq 0 \Longrightarrow x \neq 0$
and *nonzero-sndI*[*intro*, *simp*]: $snd\ x \neq 0 \Longrightarrow x \neq 0$
⟨*proof*⟩

lemma *coll-sum-list-trans*:
$xs \neq [] \Longrightarrow coll\ 0\ a\ (sum\text{-}list\ xs) \Longrightarrow (\bigwedge x.\ x \in set\ xs \Longrightarrow coll\ 0\ x\ y) \Longrightarrow$
$(\bigwedge x.\ x \in set\ xs \Longrightarrow coll\ 0\ x\ (sum\text{-}list\ xs)) \Longrightarrow$
$(\bigwedge x.\ x \in set\ xs \Longrightarrow snd\ x > 0) \Longrightarrow a \neq 0 \Longrightarrow coll\ 0\ a\ y$
⟨*proof*⟩

lemma *sum-list-coll-ex-scale*:
assumes *coll*: $\bigwedge x.\ x \in set\ xs \Longrightarrow coll\ 0\ z\ x$
assumes *nz*: $z \neq 0$
shows $\exists r.\ sum\text{-}list\ xs = r *_R z$
⟨*proof*⟩

lemma *sum-list-filter-coll-ex-scale*: $z \neq 0 \Longrightarrow \exists r.\ sum\text{-}list\ (filter\ (coll\ 0\ z)\ zs) = r *_R z$
⟨*proof*⟩

end
theory *Polygon*
imports *Counterclockwise-2D-Strict*
begin

8.4 Polygonal chains

definition *polychain xs* = (∀ *i*. *Suc i*<*length xs* ⟶ *snd* (*xs* ! *i*) = (*fst* (*xs* ! *Suc i*)))

lemma *polychainI*:
assumes $\bigwedge i.\ Suc\ i < length\ xs \Longrightarrow snd\ (xs\ !\ i) = fst\ (xs\ !\ Suc\ i)$
shows *polychain xs*
⟨*proof*⟩

lemma *polychain-Nil*[*simp*]: *polychain* [] = *True*
and *polychain-singleton*[*simp*]: *polychain* [*x*] = *True*
⟨*proof*⟩

lemma *polychain-Cons*:
polychain (*y* # *ys*) = (*if ys* = [] *then True else snd y* = *fst* (*ys* ! 0) ∧ *polychain ys*)

⟨*proof*⟩

lemma *polychain-appendI*:
polychain xs ⟹ *polychain ys* ⟹ (*xs* ≠ [] ⟹ *ys* ≠ [] ⟹ *snd* (*last xs*) = *fst* (*hd ys*)) ⟹
polychain (*xs* @ *ys*)
⟨*proof*⟩

fun *pairself* **where** *pairself f* (*x*, *y*) = (*f x*, *f y*)

lemma *pairself-apply*: *pairself f x* = (*f* (*fst x*), *f* (*snd x*))
⟨*proof*⟩

lemma *polychain-map-pairself*: *polychain xs* ⟹ *polychain* (*map* (*pairself f*) *xs*)
⟨*proof*⟩

definition *convex-polychain xs* ⟷
(*polychain xs* ∧
(∀ *i*. *Suc i* < *length xs* ⟶ *det3* (*fst* (*xs* ! *i*)) (*snd* (*xs* ! *i*)) (*snd* (*xs* ! *Suc i*)) > *0*))

lemma *convex-polychain-Cons2*[*simp*]:
convex-polychain (*x*#*y*#*zs*) ⟷
snd x = *fst y* ∧ *det3* (*fst x*) (*fst y*) (*snd y*) > *0* ∧ *convex-polychain* (*y*#*zs*)
⟨*proof*⟩

lemma *convex-polychain-ConsD*:
assumes *convex-polychain* (*x*#*xs*)
shows *convex-polychain xs*
⟨*proof*⟩

definition
convex-polygon xs ⟷ (*convex-polychain xs* ∧ (*xs* ≠ [] ⟶ *fst* (*hd xs*) = *snd* (*last xs*)))

lemma *convex-polychain-Nil*[*simp*]: *convex-polychain* [] = *True*
and *convex-polychain-Cons*[*simp*]: *convex-polychain* [*x*] = *True*
⟨*proof*⟩

lemma *convex-polygon-Cons2*[*simp*]:
convex-polygon (*x*#*y*#*zs*) ⟷ *fst x* = *snd* (*last* (*y*#*zs*)) ∧ *convex-polychain* (*x*#*y*#*zs*)
⟨*proof*⟩

lemma *polychain-append-connected*:
polychain (*xs* @ *ys*) ⟹ *xs* ≠ [] ⟹ *ys* ≠ [] ⟹ *fst* (*hd ys*) = *snd* (*last xs*)
⟨*proof*⟩

lemma *convex-polychain-appendI*:

assumes *cxs*: *convex-polychain xs*
assumes *cys*: *convex-polychain ys*
assumes *pxy*: *polychain* (*xs* @ *ys*)
assumes *xs* ≠ [] ⟹ *ys* ≠ [] ⟹ *det3* (*fst* (*last xs*)) (*snd* (*last xs*)) (*snd* (*hd ys*)) > *0*
shows *convex-polychain* (*xs* @ *ys*)
⟨*proof*⟩

lemma *convex-polychainI*:
assumes *polychain xs*
assumes ⋀*i*. *Suc i* < *length xs* ⟹ *det3* (*fst* (*xs* ! *i*)) (*snd* (*xs* ! *i*)) (*snd* (*xs* ! *Suc i*)) > *0*
shows *convex-polychain xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

primrec *polychain-of*::'*a*::*ab-group-add* ⇒ '*a list* ⇒ ('*a*∗'*a*) *list* **where**
polychain-of xc [] = []
| *polychain-of xc* (*xm*#*xs*) = (*xc*, *xc* + *xm*)#*polychain-of* (*xc* + *xm*) *xs*

lemma *in-set-polychain-ofD*: *ab* ∈ *set* (*polychain-of x xs*) ⟹ (*snd ab* − *fst ab*) ∈ *set xs*
⟨*proof*⟩

lemma *fst-polychain-of-nth-0*[*simp*]: *xs* ≠ [] ⟹ *fst* ((*polychain-of p xs*) ! *0*) = *p*
⟨*proof*⟩

lemma *fst-hd-polychain-of*: *xs* ≠ [] ⟹ *fst* (*hd* (*polychain-of x xs*)) = *x*
⟨*proof*⟩

lemma *length-polychain-of-eq*[*simp*]:
shows *length* (*polychain-of p qs*) = *length qs*
⟨*proof*⟩

lemma
polychain-of-subsequent-eq:
assumes *Suc i* < *length qs*
shows *snd* (*polychain-of p qs* ! *i*) = *fst* (*polychain-of p qs* ! *Suc i*)
⟨*proof*⟩

lemma *polychain-of-eq-empty-iff*[*simp*]: *polychain-of p xs* = [] ⟷ *xs* = []
⟨*proof*⟩

lemma *in-set-polychain-of-imp-sum-list*:
assumes $z \in set\ (polychain\text{-}of\ Pc\ Ps)$
obtains d **where** $z = (Pc + sum\text{-}list\ (take\ d\ Ps),\ Pc + sum\text{-}list\ (take\ (Suc\ d)\ Ps))$
⟨*proof*⟩

lemma *last-polychain-of*: $length\ xs > 0 \Longrightarrow snd\ (last\ (polychain\text{-}of\ p\ xs)) = p + sum\text{-}list\ xs$
⟨*proof*⟩

lemma *polychain-of-singleton-iff*: $polychain\text{-}of\ p\ xs = [a] \longleftrightarrow fst\ a = p \wedge xs = [(snd\ a - p)]$
⟨*proof*⟩

lemma *polychain-of-add*: $polychain\text{-}of\ (x + y)\ xs = map\ (((+)\ (y, y)))\ (polychain\text{-}of\ x\ xs)$
⟨*proof*⟩

8.5 Dirvec: Inverse of Polychain

primrec *dirvec* **where** $dirvec\ (x, y) = (y - x)$

lemma *dirvec-minus*: $dirvec\ x = snd\ x - fst\ x$
⟨*proof*⟩

lemma *dirvec-nth-polychain-of*: $n < length\ xs \Longrightarrow dirvec\ ((polychain\text{-}of\ p\ xs)\ !\ n) = (xs\ !\ n)$
⟨*proof*⟩

lemma *dirvec-hd-polychain-of*: $xs \neq [] \Longrightarrow dirvec\ (hd\ (polychain\text{-}of\ p\ xs)) = (hd\ xs)$
⟨*proof*⟩

lemma *dirvec-last-polychain-of*: $xs \neq [] \Longrightarrow dirvec\ (last\ (polychain\text{-}of\ p\ xs)) = (last\ xs)$
⟨*proof*⟩

lemma *map-dirvec-polychain-of*[*simp*]: $map\ dirvec\ (polychain\text{-}of\ x\ xs) = xs$
⟨*proof*⟩

8.6 Polychain of Sorted (*polychain-of*, *ccw'.sortedP*)

lemma *ccw'-sortedP-translateD*:
$linorder\text{-}list0.sortedP\ (ccw'\ x0)\ (map\ ((+)\ x \circ g)\ xs) \Longrightarrow$
$linorder\text{-}list0.sortedP\ (ccw'\ (x0 - x))\ (map\ g\ xs)$
⟨*proof*⟩

lemma *ccw'-sortedP-translateI*:
$linorder\text{-}list0.sortedP\ (ccw'\ (x0 - x))\ (map\ g\ xs) \Longrightarrow$
$linorder\text{-}list0.sortedP\ (ccw'\ x0)\ (map\ ((+)\ x \circ g)\ xs)$

⟨*proof*⟩

lemma *ccw′-sortedP-translate-comp*[*simp*]:
linorder-list0.sortedP (*ccw′ x0*) (*map* ((+) *x* ∘ *g*) *xs*) ⟷
linorder-list0.sortedP (*ccw′* (*x0* − *x*)) (*map g xs*)
⟨*proof*⟩

lemma *snd-plus-commute*: *snd* ∘ (+) (*x0*, *x0*) = (+) *x0* ∘ *snd*
⟨*proof*⟩

lemma *ccw′-sortedP-renormalize*:
ccw′.sortedP a (*map snd* (*polychain-of* (*x0* + *x*) *xs*)) ⟷
ccw′.sortedP (*a* − *x0*) (*map snd* (*polychain-of x xs*))
⟨*proof*⟩

lemma *ccw′-sortedP-polychain-of01*:
shows *ccw′.sortedP 0* [*u*] ⟹ *ccw′.sortedP x0* (*map snd* (*polychain-of x0* [*u*]))
and *ccw′.sortedP 0* [] ⟹ *ccw′.sortedP x0* (*map snd* (*polychain-of x0* []))
⟨*proof*⟩

lemma *ccw′-sortedP-polychain-of2*:
assumes *ccw′.sortedP 0* [*u*, *v*]
shows *ccw′.sortedP x0* (*map snd* (*polychain-of x0* [*u*, *v*]))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *ccw′-sortedP-polychain-of-snd*:
assumes *ccw′.sortedP 0 xs*
shows *ccw′.sortedP x0* (*map snd* (*polychain-of x0 xs*))
⟨*proof*⟩

lemma *ccw′-sortedP-implies-distinct*:
assumes *ccw′.sortedP x qs*
shows *distinct qs*
⟨*proof*⟩

lemma *ccw′-sortedP-implies-nonaligned*:
assumes *ccw′.sortedP x qs*
assumes *y* ∈ *set qs z* ∈ *set qs y* ≠ *z*
shows ¬ *coll x y z*
⟨*proof*⟩

lemma *list-all-mp*: *list-all P xs* ⟹ (⋀*x*. *x* ∈ *set xs* ⟹ *P x* ⟹ *Q x*) ⟹ *list-all Q xs*
⟨*proof*⟩

lemma
ccw′-scale-origin:
assumes $e \in UNIV \to \{0<..<1\}$
assumes $x \in set\ (polychain\text{-}of\ Pc\ (P\ \#\ QRRs))$
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*⟩

lemma *polychain-of-ccw-convex*:
assumes $e \in UNIV \to \{0 <..< 1\}$
assumes *sorted*: $linorder\text{-}list0.sortedP\ (ccw'\ 0)\ (P\#Q\#Ps)$
shows *list-all*
$(\lambda(xi,\ xj).\ ccw'\ xi\ xj\ (Pc + (\sum P \in set\ (P\#Q\#Ps).\ e\ P *_R P)))$
$(polychain\text{-}of\ Pc\ (P\#Q\#Ps))$
⟨*proof*⟩

lemma *polychain-of-ccw*:
assumes $e \in UNIV \to \{0 <..< 1\}$
assumes *sorted*: $ccw'.sortedP\ 0\ qs$
assumes *qs*: $length\ qs \neq 1$
shows $list\text{-}all\ (\lambda(xi,\ xj).\ ccw'\ xi\ xj\ (Pc + (\sum P \in set\ qs.\ e\ P *_R P)))\ (polychain\text{-}of\ Pc\ qs)$
⟨*proof*⟩

lemma *in-polychain-of-ccw*:
assumes $e \in UNIV \to \{0 <..< 1\}$
assumes $ccw'.sortedP\ 0\ qs$
assumes $length\ qs \neq 1$
assumes $seg \in set\ (polychain\text{-}of\ Pc\ qs)$
shows $ccw'\ (fst\ seg)\ (snd\ seg)\ (Pc + (\sum P \in set\ qs.\ e\ P *_R P))$
⟨*proof*⟩

lemma *distinct-butlast-ne-last*: $distinct\ xs \Longrightarrow x \in set\ (butlast\ xs) \Longrightarrow x \neq last\ xs$
⟨*proof*⟩

lemma
ccw′-sortedP-convex-rotate-aux:
assumes $ccw'.sortedP\ 0\ (zs)\ ccw'.sortedP\ x\ (map\ snd\ (polychain\text{-}of\ x\ (zs)))$
shows $ccw'.sortedP\ (snd\ (last\ (polychain\text{-}of\ x\ (zs))))\ (map\ snd\ (butlast\ (polychain\text{-}of\ x\ (zs))))$
⟨*proof*⟩

lemma *ccw′-polychain-of-sorted-center-last*:
assumes *set-butlast*: $(c,\ d) \in set\ (butlast\ (polychain\text{-}of\ x0\ xs))$
assumes *sorted*: $ccw'.sortedP\ 0\ xs$
assumes *ne*: $xs \neq []$
shows $ccw'\ x0\ d\ (snd\ (last\ (polychain\text{-}of\ x0\ xs)))$

⟨*proof*⟩

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* ⇒ *point* ⇒ *bool* **where**
lex p q ⟷ (*fst p* < *fst q* ∨ *fst p* = *fst q* ∧ *snd p* < *snd q* ∨ *p* = *q*)

definition *psi*::*point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**
psi p q r ⟷ (*lex p q* ∧ *lex q r*)

definition *ccw*::*point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**
ccw p q r ⟷ *ccw′ p q r* ∨ (*det3 p q r* = *0* ∧ (*psi p q r* ∨ *psi q r p* ∨ *psi r p q*))

interpretation *ccw*: *linorder-list0 ccw x* **for** *x* ⟨*proof*⟩

lemma *ccw′-imp-ccw*: *ccw′ a b c* ⟹ *ccw a b c*
⟨*proof*⟩

lemma *ccw-ncoll-imp-ccw*: *ccw a b c* ⟹ ¬*coll a b c* ⟹ *ccw′ a b c*
⟨*proof*⟩

lemma *ccw-translate*: *ccw p* (*p* + *q*) (*p* + *r*) = *ccw 0 q r*
⟨*proof*⟩

lemma *ccw-translate-origin*: *NO-MATCH 0 p* ⟹ *ccw p q r* = *ccw 0* (*q* − *p*) (*r* − *p*)
⟨*proof*⟩

lemma *psi-scale*:
psi (*r* ∗$_R$ *a*) (*r* ∗$_R$ *b*) *0* = (*if r* > *0 then psi a b 0 else if r* < *0 then psi 0 b a else True*)
psi (*r* ∗$_R$ *a*) *0* (*r* ∗$_R$ *b*) = (*if r* > *0 then psi a 0 b else if r* < *0 then psi b 0 a else True*)
psi 0 (*r* ∗$_R$ *a*) (*r* ∗$_R$ *b*) = (*if r* > *0 then psi 0 a b else if r* < *0 then psi b a 0 else True*)
⟨*proof*⟩

lemma *ccw-scale23*: *ccw 0 a b* ⟹ *r* > *0* ⟹ *ccw 0* (*r* ∗$_R$ *a*) (*r* ∗$_R$ *b*)
⟨*proof*⟩

lemma *psi-notI*: *distinct3 p q r* ⟹ *psi p q r* ⟹ ¬ *psi q p r*

⟨*proof*⟩

lemma *not-lex-eq*: ¬ *lex a b* ⟷ *lex b a* ∧ *b* ≠ *a*
⟨*proof*⟩

lemma *lex-trans*: *lex a b* ⟹ *lex b c* ⟹ *lex a c*
⟨*proof*⟩

lemma *lex-sym-eqI*: *lex a b* ⟹ *lex b a* ⟹ *a* = *b*
and *lex-sym-eq-iff*: *lex a b* ⟹ *lex b a* ⟷ *a* = *b*
⟨*proof*⟩

lemma *lex-refl*[*simp*]: *lex p p*
⟨*proof*⟩

lemma *psi-disjuncts*:
distinct3 p q r ⟹ *psi p q r* ∨ *psi p r q* ∨ *psi q r p* ∨ *psi q p r* ∨ *psi r p q* ∨ *psi r q p*
⟨*proof*⟩

lemma *nlex-ccw-left*: *lex x 0* ⟹ *ccw 0* (*0*, *1*) *x*
⟨*proof*⟩

interpretation *ccw-system123 ccw*
⟨*proof*⟩

lemma *lex-scaleR-nonneg*: *lex a b* ⟹ *r* ≥ *0* ⟹ *lex a* (*a* + *r* ∗*R* (*b* − *a*))
⟨*proof*⟩

lemma *lex-scale1-zero*:
lex (*v* ∗*R* *u*) *0* = (*if v* > *0 then lex u 0 else if v* < *0 then lex 0 u else True*)
and *lex-scale2-zero*:
lex 0 (*v* ∗*R* *u*) = (*if v* > *0 then lex 0 u else if v* < *0 then lex u 0 else True*)
⟨*proof*⟩

lemma *nlex-add*:
assumes *lex a 0 lex b 0*
shows *lex* (*a* + *b*) *0*
⟨*proof*⟩

lemma *nlex-sum*:
assumes *finite X*
assumes ⋀*x*. *x* ∈ *X* ⟹ *lex* (*f x*) *0*
shows *lex* (*sum f X*) *0*
⟨*proof*⟩

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*⟩

lemma *lex-sum-list*: *(⋀x. x ∈ set xs ⟹ lex x 0) ⟹ lex (sum-list xs) 0*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *lex-diff1*: *lex (a − b) c = lex a (c + b)*
and *lex-diff2*: *lex c (a − b) = lex (c + b) a*
⟨*proof*⟩

lemma *sum-list-eq-0-iff-nonpos*:
fixes *xs*::*'a*::*ordered-ab-group-add list*
shows *list-all (λx. x ≤ 0) xs ⟹ sum-list xs = 0 ⟷ (∀ n∈set xs. n = 0)*
⟨*proof*⟩

lemma *sum-list-nlex-eq-zeroI*:
assumes *nlex*: *list-all (λx. lex x 0) xs*
assumes *sum-list xs = 0*
assumes *x ∈ set xs*
shows *x = 0*
⟨*proof*⟩

lemma *sum-list-eq0I*: *(∀ x∈set xs. x = 0) ⟹ sum-list xs = 0*
⟨*proof*⟩

lemma *sum-list-nlex-eq-zero-iff*:
assumes *nlex*: *list-all (λx. lex x 0) xs*
shows *sum-list xs = 0 ⟷ list-all ((=) 0) xs*
⟨*proof*⟩

lemma
assumes *lex p q lex q r 0 ≤ a 0 ≤ b 0 ≤ c a + b + c = 1*
assumes *comb-def*: *comb = a *$_R$ p + b *$_R$ q + c *$_R$ r*
shows *lex-convex3*: *lex p comb lex comb r*
⟨*proof*⟩

lemma *lex-convex-self2*:
assumes *lex p q 0 ≤ a a ≤ 1*
defines *r ≡ a *$_R$ p + (1 − a) *$_R$ q*
shows *lex p r* (**is** *?th1*)

and *lex r q* (**is** *?th2*)
⟨*proof*⟩

lemma *lex-uminus0*[*simp*]: *lex* (−*a*) *0* = *lex 0 a*
⟨*proof*⟩

lemma
lex-fst-zero-imp:
fst x = *0* ⟹ *lex x 0* ⟹ *lex y 0* ⟹ ¬*coll 0 x y* ⟹ *ccw′ 0 y x*
⟨*proof*⟩

lemma *lex-ccw-left*: *lex x y* ⟹ *r* > *0* ⟹ *ccw y* (*y* + (*0*, *r*)) *x*
⟨*proof*⟩

lemma *lex-translate-origin*: *NO-MATCH 0 a* ⟹ *lex a b* = *lex 0* (*b* − *a*)
⟨*proof*⟩

9.2 Order prover setup

definition *lexs p q* ⟷ (*lex p q* ∧ *p* ≠ *q*)

lemma *lexs-irrefl*: ¬ *lexs p p*
and *lexs-imp-lex*: *lexs x y* ⟹ *lex x y*
and *not-lexs*: (¬ *lexs x y*) = (*lex y x*)
and *not-lex*: (¬ *lex x y*) = (*lexs y x*)
and *eq-lex-refl*: *x* = *y* ⟹ *lex x y*
⟨*proof*⟩

lemma *lexs-trans*: *lexs x y* ⟹ *lexs y z* ⟹ *lexs x z*
and *lexs-lex-trans*: *lexs x y* ⟹ *lex y z* ⟹ *lexs x z*
and *lex-lexs-trans*: *lex x y* ⟹ *lexs y z* ⟹ *lexs x z*
and *lex-neq-trans*: *lex a b* ⟹ *a* ≠ *b* ⟹ *lexs a b*
and *neq-lex-trans*: *a* ≠ *b* ⟹ *lex a b* ⟹ *lexs a b*
and *lexs-imp-neq*: *lexs a b* ⟹ *a* ≠ *b*
⟨*proof*⟩

⟨*ML*⟩

9.3 Contradictions

lemma
assumes *d*: *distinct4 s p q r*
shows *contra1*: ¬(*lex p q* ∧ *lex q r* ∧ *lex r s* ∧ *indelta s p q r*) (**is** *?th1*)
and *contra2*: ¬(*lex s p* ∧ *lex p q* ∧ *lex q r* ∧ *indelta s p q r*) (**is** *?th2*)
and *contra3*: ¬(*lex p r* ∧ *lex p s* ∧ *lex q r* ∧ *lex q s* ∧ *insquare p r q s*) (**is** *?th3*)
⟨*proof*⟩

lemma *ccw′-subst-psi-disj*:
assumes *det3 t r s* = *0*
assumes *psi t r s* ∨ *psi t s r* ∨ *psi s r t*

assumes $s \neq t$
assumes $ccw'\ t\ r\ p$
shows $ccw'\ t\ s\ p$
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

interpretation *ccw-system4 ccw*
⟨*proof*⟩

lemma *lex-total*: *lex t q* $\wedge\ t \neq q \vee$ *lex q t* $\wedge\ t \neq q \vee t = q$
⟨*proof*⟩

lemma
ccw-two-up-contra:
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*⟩

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*⟩

interpretation *ccw*: *ccw-system ccw*
⟨*proof*⟩

lemma *ccw-scaleR1*:
det3 0 xr P ≠ *0* ⟹ *0* < *e* ⟹ *ccw 0 xr P* ⟹ *ccw 0* (*e*∗$_R$*xr*) *P*
⟨*proof*⟩

lemma *ccw-scaleR2*:
det3 0 xr P ≠ *0* ⟹ *0* < *e* ⟹ *ccw 0 xr P* ⟹ *ccw 0 xr* (*e*∗$_R$*P*)
⟨*proof*⟩

lemma *ccw-translate3-aux*:
assumes ¬*coll 0 a b*
assumes *x* < *1*
assumes *ccw 0* (*a* − *x*∗$_R$*a*) (*b* − *x* ∗$_R$ *a*)
shows *ccw 0 a b*
⟨*proof*⟩

lemma *ccw-translate3-minus*: *det3 0 a b* ≠ *0* ⟹ *x* < *1* ⟹ *ccw 0 a* (*b* − *x* ∗$_R$ *a*) ⟹ *ccw 0 a b*
⟨*proof*⟩

lemma *ccw-translate3*: *det3 0 a b* ≠ *0* ⟹ *x* < *1* ⟹ *ccw 0 a b* ⟹ *ccw 0 a* (*x* ∗$_R$ *a* + *b*)
⟨*proof*⟩

lemma *ccw-switch23*: *det3 0 P Q* ≠ *0* ⟹ (¬ *ccw 0 Q P* ⟷ *ccw 0 P Q*)
⟨*proof*⟩

lemma *ccw0-upward*: *fst a* > *0* ⟹ *snd a* = *0* ⟹ *snd b* > *snd a* ⟹ *ccw 0 a b*
⟨*proof*⟩

lemma *ccw-uminus3*[*simp*]: *det3 a b c* ≠ *0* ⟹ *ccw* (−*a*) (−*b*) (−*c*) = *ccw a b c*
⟨*proof*⟩

lemma *coll-minus-eq*: *coll* (*x* − *a*) (*x* − *b*) (*x* − *c*) = *coll a b c*
⟨*proof*⟩

lemma *ccw-minus3*: ¬ *coll a b c* ⟹ *ccw* (*x* − *a*) (*x* − *b*) (*x* − *c*) ⟷ *ccw a b c*
⟨*proof*⟩

lemma *ccw0-uminus*[*simp*]: ¬ *coll 0 a b* ⟹ *ccw 0* (−*a*) (−*b*) ⟷ *ccw 0 a b*
⟨*proof*⟩

lemma *lex-convex2*:
assumes *lex p q lex p r 0* ≤ *u u* ≤ *1*
shows *lex p* (*u* ∗$_R$ *q* + (*1* − *u*) ∗$_R$ *r*)
⟨*proof*⟩

lemma *lex-convex2'*:
assumes *lex q p lex r p 0* ≤ *u u* ≤ *1*
shows *lex* (*u* ∗$_R$ *q* + (*1* − *u*) ∗$_R$ *r*) *p*
⟨*proof*⟩

lemma *psi-convex1*:
assumes *psi c a b*
assumes *psi d a b*
assumes *0* ≤ *u 0* ≤ *v u* + *v* = *1*
shows *psi* (*u* ∗$_R$ *c* + *v* ∗$_R$ *d*) *a b*
⟨*proof*⟩

lemma *psi-convex2*:
assumes *psi a c b*
assumes *psi a d b*
assumes *0* ≤ *u 0* ≤ *v u* + *v* = *1*
shows *psi a* (*u* ∗$_R$ *c* + *v* ∗$_R$ *d*) *b*
⟨*proof*⟩

lemma *psi-convex3*:
assumes *psi a b c*
assumes *psi a b d*
assumes *0* ≤ *u 0* ≤ *v u* + *v* = *1*
shows *psi a b* (*u* ∗$_R$ *c* + *v* ∗$_R$ *d*)
⟨*proof*⟩

lemma *coll-convex*:
assumes *coll a b c coll a b d*
assumes *0* ≤ *u 0* ≤ *v u* + *v* = *1*
shows *coll a b* (*u* ∗$_R$ *c* + *v* ∗$_R$ *d*)
⟨*proof*⟩

lemma (**in** *ccw-vector-space*) *convex3*:
assumes *u* ≥ *0 v* ≥ *0 u* + *v* = *1 ccw a b d ccw a b c*
shows *ccw a b* (*u* ∗$_R$ *c* + *v* ∗$_R$ *d*)
⟨*proof*⟩

lemma *ccw-self*[*simp*]: *ccw a a b ccw b a a*
⟨*proof*⟩

lemma *ccw-self'*[*simp*]: *ccw a b a*
⟨*proof*⟩

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*⟩

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* $\Longrightarrow$ *coll a b d* $\Longrightarrow$ *lex b a*
shows *ccw a b* $(u *_R c + v *_R d)$
⟨*proof*⟩

interpretation *ccw*: *ccw-convex ccw S* $\lambda a\ b.$ *lex b a* **for** *S*
⟨*proof*⟩

lemma *ccw-sorted-scaleR*: *ccw.sortedP 0 xs* $\Longrightarrow$ $r > 0$ $\Longrightarrow$ *ccw.sortedP 0* (*map* $((*_R)\ r)$ *xs*)
⟨*proof*⟩

lemma *ccw-sorted-implies-ccw'-sortedP*:
assumes *nonaligned*: $\bigwedge y\ z.\ y \in set\ Ps \Longrightarrow z \in set\ Ps \Longrightarrow y \neq z \Longrightarrow \neg$ *coll 0 y z*
assumes *sorted*: *linorder-list0.sortedP* (*ccw 0*) *Ps*
assumes *distinct Ps*
shows *linorder-list0.sortedP* (*ccw' 0*) *Ps*
⟨*proof*⟩

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*: $z \in$ *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*)
⟨*proof*⟩

lemma *lex-polychain-of-center*:
d ∈ *set* (*polychain-of x0 xs*) ⟹ *list-all* (λ*x*. *lex x 0*) *xs* ⟹ *lex* (*snd d*) *x0*
⟨*proof*⟩

lemma *lex-polychain-of-last*:
(*c*, *d*) ∈ *set* (*polychain-of x0 xs*) ⟹ *list-all* (λ*x*. *lex x 0*) *xs* ⟹
lex (*snd* (*last* (*polychain-of x0 xs*))) *d*
⟨*proof*⟩

lemma *distinct-fst-polychain-of*:
assumes *list-all* (λ*x*. *x* ≠ *0*) *xs*
assumes *list-all* (λ*x*. *lex x 0*) *xs*
shows *distinct* (*map fst* (*polychain-of x0 xs*))
⟨*proof*⟩

lemma *distinct-snd-polychain-of*:
assumes *list-all* (λ*x*. *x* ≠ *0*) *xs*
assumes *list-all* (λ*x*. *lex x 0*) *xs*
shows *distinct* (*map snd* (*polychain-of x0 xs*))
⟨*proof*⟩

10.2 Orient all entries

lift-definition *nlex-pdevs*::*point pdevs* ⇒ *point pdevs*
is λ*x n*. *if lex 0* (*x n*) *then* − *x n else x n* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nlex-pdevs-zero-pdevs*[*simp*]: *nlex-pdevs zero-pdevs* = *zero-pdevs*
⟨*proof*⟩

lemma *pdevs-domain-nlex-pdevs*[*simp*]: *pdevs-domain* (*nlex-pdevs x*) = *pdevs-domain x*
⟨*proof*⟩

lemma *degree-nlex-pdevs*[*simp*]: *degree* (*nlex-pdevs x*) = *degree x*
⟨*proof*⟩

lemma
pdevs-val-nlex-pdevs:
assumes *e* ∈ *UNIV* → *I uminus* ‘ *I* = *I*
obtains *e′* **where** *e′* ∈ *UNIV* → *I pdevs-val e x* = *pdevs-val e′* (*nlex-pdevs x*)
⟨*proof*⟩

lemma
pdevs-val-nlex-pdevs2:
assumes $e \in UNIV \to I$ *uminus* ‘ $I = I$
obtains e' **where** $e' \in UNIV \to I$ *pdevs-val e* (*nlex-pdevs x*) = *pdevs-val* e' *x*
⟨*proof*⟩

lemma
pdevs-val-selsort-ccw:
assumes *distinct xs*
assumes $e \in UNIV \to I$
obtains e' **where** $e' \in UNIV \to I$
pdevs-val e (*pdevs-of-list xs*) = *pdevs-val* e' (*pdevs-of-list* (*ccw.selsort 0 xs*))
⟨*proof*⟩

lemma
pdevs-val-selsort-ccw2:
assumes *distinct xs*
assumes $e \in UNIV \to I$
obtains e' **where** $e' \in UNIV \to I$
pdevs-val e (*pdevs-of-list* (*ccw.selsort 0 xs*)) = *pdevs-val* e' (*pdevs-of-list xs*)
⟨*proof*⟩

lemma *lex-nlex-pdevs*: *lex* (*pdevs-apply* (*nlex-pdevs x*) *i*) *0*
⟨*proof*⟩

10.3 Lowest Vertex

definition *lowest-vertex*::′*a*::*ordered-euclidean-space aform* ⇒ ′*a* **where**
lowest-vertex X = *fst X* − *sum-list* (*map snd* (*list-of-pdevs* (*snd X*)))

lemma *snd-abs*: *snd* (*abs x*) = *abs* (*snd x*)
⟨*proof*⟩

lemma *lowest-vertex*:
fixes *X Y*::(*real*∗*real*) *aform*
assumes $e \in UNIV \to \{-1 .. 1\}$
assumes $\bigwedge i$. *snd* (*pdevs-apply* (*snd X*) *i*) ≥ *0*
assumes $\bigwedge i$. *abs* (*snd* (*pdevs-apply* (*snd Y*) *i*)) = *abs* (*snd* (*pdevs-apply* (*snd X*) *i*))
assumes *degree-aform Y* = *degree-aform X*
assumes *fst Y* = *fst X*
shows *snd* (*lowest-vertex X*) ≤ *snd* (*aform-val e Y*)
⟨*proof*⟩

lemma *sum-list-nonposI*:
fixes *xs*::′*a*::*ordered-comm-monoid-add list*
shows *list-all* (λ*x*. *x* ≤ *0*) *xs* ⟹ *sum-list xs* ≤ *0*
⟨*proof*⟩

lemma *center-le-lowest*:
fst (*fst X*) ≤ *fst* (*lowest-vertex* (*fst X*, *nlex-pdevs* (*snd X*)))
⟨*proof*⟩

lemma *lowest-vertex-eq-center-iff*:
lowest-vertex (*x0*, *nlex-pdevs* (*snd X*)) = *x0* ⟷ *snd X* = *zero-pdevs*
⟨*proof*⟩

10.4 Collinear Generators

lemma *scaleR-le-self-cancel*:
fixes *c*::'*a*::*ordered-real-vector*
shows *a* *$_R$ *c* ≤ *c* ⟷ (*1* < *a* ∧ *c* ≤ *0* ∨ *a* < *1* ∧ *0* ≤ *c* ∨ *a* = *1*)
⟨*proof*⟩

lemma *pdevs-val-coll*:
assumes *coll*: *list-all* (*coll 0 x*) *xs*
assumes *nlex*: *list-all* (λ*x*. *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* *$_R$ (*sum-list xs*)
⟨*proof*⟩

lemma *scaleR-eq-self-cancel*:
fixes *x*::'*a*::*real-vector*
shows *a* *$_R$ *x* = *x* ⟷ *a* = *1* ∨ *x* = *0*
⟨*proof*⟩

lemma *abs-pdevs-val-less-tdev*:
assumes *e* ∈ *UNIV* → {−*1* <..< *1*} *degree x* > *0*
shows |*pdevs-val e x*| < *tdev x*
⟨*proof*⟩

lemma *pdevs-val-coll-strict*:
assumes *coll*: *list-all* (*coll 0 x*) *xs*
assumes *nlex*: *list-all* (λ*x*. *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* *$_R$ (*sum-list xs*)
⟨*proof*⟩

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* ∈ *set* (*independent-pdevs xs*)
obtains *x* **where** *x*∈*set xs coll 0 x y*
⟨*proof*⟩

lemma *in-set-independent-pdevs-nonzero*: *x* ∈ *set* (*independent-pdevs xs*) ⟹ *x* ≠ *0*
⟨*proof*⟩

lemma *independent-pdevs-pairwise-non-coll*:
assumes *x* ∈ *set* (*independent-pdevs xs*)
assumes *y* ∈ *set* (*independent-pdevs xs*)
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*
assumes *x* ≠ *y*
shows ¬ *coll 0 x y*
⟨*proof*⟩

lemma *distinct-independent-pdevs*[*simp*]:
shows *distinct* (*independent-pdevs xs*)
⟨*proof*⟩

lemma *in-set-independent-pdevs-invariant-nlex*:
x ∈ *set* (*independent-pdevs xs*) ⟹ (⋀*x*. *x* ∈ *set xs* ⟹ *lex x 0*) ⟹
(⋀*x*. *x* ∈ *set xs* ⟹ *x* ≠ *0*) ⟹ *Counterclockwise-2D-Arbitrary.lex x 0*
⟨*proof*⟩

lemma
pdevs-val-independent-pdevs2:
assumes *e* ∈ *UNIV* → *I*
shows ∃ *e'*. *e'* ∈ *UNIV* → *I* ∧
pdevs-val e (*pdevs-of-list* (*independent-pdevs xs*)) = *pdevs-val e'* (*pdevs-of-list xs*)
⟨*proof*⟩

lemma *list-all-filter*: *list-all p* (*filter p xs*)
⟨*proof*⟩

lemma *pdevs-val-independent-pdevs*:
assumes *list-all* (λ*x*. *lex x 0*) *xs*
assumes *list-all* (λ*x*. *x* ≠ *0*) *xs*
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
shows ∃ *e'*. *e'* ∈ *UNIV* → {−*1* .. *1*} ∧ *pdevs-val e* (*pdevs-of-list xs*) =
pdevs-val e' (*pdevs-of-list* (*independent-pdevs xs*))
⟨*proof*⟩

lemma
pdevs-val-independent-pdevs-strict:
assumes *list-all* ($\lambda x.$ *lex x 0*) *xs*
assumes *list-all* ($\lambda x.$ $x \neq 0$) *xs*
assumes $e \in UNIV \to \{-1 <..< 1\}$
shows $\exists e'.\ e' \in UNIV \to \{-1 <..< 1\} \wedge$ *pdevs-val e* (*pdevs-of-list xs*) =
pdevs-val e' (*pdevs-of-list* (*independent-pdevs xs*))
⟨*proof*⟩

lemma *sum-list-independent-pdevs*: *sum-list* (*independent-pdevs xs*) = *sum-list xs*
⟨*proof*⟩

lemma *independent-pdevs-eq-Nil-iff*:
list-all ($\lambda x.$ *lex x 0*) *xs* $\Longrightarrow$ *list-all* ($\lambda x.$ $x \neq 0$) *xs* $\Longrightarrow$ *independent-pdevs xs* = [] $\longleftrightarrow$ *xs* = []
⟨*proof*⟩

10.6 Independent Oriented Generators

definition *inl p* = *independent-pdevs* (*map snd* (*list-of-pdevs* (*nlex-pdevs p*)))

lemma *distinct-inl*[*simp*]: *distinct* (*inl* (*snd X*))
⟨*proof*⟩

lemma *in-set-inl-nonzero*: $x \in$ *set* (*inl xs*) $\Longrightarrow$ $x \neq 0$
⟨*proof*⟩

lemma
inl-ncoll: $y \in$ *set* (*inl* (*snd X*)) $\Longrightarrow$ $z \in$ *set* (*inl* (*snd X*)) $\Longrightarrow$ $y \neq z \Longrightarrow \neg$*coll 0 y z*
⟨*proof*⟩

lemma *in-set-inl-lex*: $x \in$ *set* (*inl xs*) $\Longrightarrow$ *lex x 0*
⟨*proof*⟩

interpretation *ccw0*: *linorder-list ccw 0 set* (*inl* (*snd X*))
⟨*proof*⟩

lemma *sorted-inl*: *ccw.sortedP 0* (*ccw.selsort 0* (*inl* (*snd X*)))
⟨*proof*⟩

lemma *sorted-scaled-inl*: *ccw.sortedP 0* (*map* ($*_R$) *2*) (*ccw.selsort 0* (*inl* (*snd X*))))
⟨*proof*⟩

lemma *distinct-selsort-inl*: *distinct* (*ccw.selsort 0* (*inl* (*snd X*)))
⟨*proof*⟩

lemma *distinct-map-scaleRI*:
fixes *xs*::*'a*::*real-vector list*
shows *distinct xs* $\Longrightarrow$ *c* $\neq$ *0* $\Longrightarrow$ *distinct* (*map* ((∗$_R$) *c*) *xs*)
⟨*proof*⟩

lemma *distinct-scaled-inl*: *distinct* (*map* ((∗$_R$) *2*) (*ccw.selsort 0* (*inl* (*snd X*))))
⟨*proof*⟩

lemma *ccw'-sortedP-scaled-inl*:
ccw'.sortedP 0 (*map* ((∗$_R$) *2*) (*ccw.selsort 0* (*inl* (*snd X*))))
⟨*proof*⟩

lemma *pdevs-val-pdevs-of-list-inl2E*:
assumes *e* $\in$ *UNIV* $\rightarrow$ {*−1* .. *1*}
obtains *e'* **where** *pdevs-val e X* = *pdevs-val e'* (*pdevs-of-list* (*inl X*)) *e'* $\in$ *UNIV* $\rightarrow$ {*−1* .. *1*}
⟨*proof*⟩

lemma *pdevs-val-pdevs-of-list-inlE*:
assumes *e* $\in$ *UNIV* $\rightarrow$ *I uminus* ‘ *I* = *I 0* $\in$ *I*
obtains *e'* **where** *pdevs-val e* (*pdevs-of-list* (*inl X*)) = *pdevs-val e' X e'* $\in$ *UNIV* $\rightarrow$ *I*
⟨*proof*⟩

lemma *sum-list-nlex-eq-sum-list-inl*:
sum-list (*map snd* (*list-of-pdevs* (*nlex-pdevs X*))) = *sum-list* (*inl X*)
⟨*proof*⟩

lemma *Affine-inl*: *Affine* (*fst X*, *pdevs-of-list* (*inl* (*snd X*))) = *Affine X*
⟨*proof*⟩

10.7 Half Segments

definition *half-segments-of-aform*::*point aform* ⇒ (*point*∗*point*) *list*
where *half-segments-of-aform X* =
(*let*
x0 = *lowest-vertex* (*fst X*, *nlex-pdevs* (*snd X*))
in
polychain-of x0 (*map* ((∗$_R$) *2*) (*ccw.selsort 0* (*inl* (*snd X*)))))

lemma *subsequent-half-segments*:
fixes *X*
assumes *Suc i* < *length* (*half-segments-of-aform X*)
shows *snd* (*half-segments-of-aform X* ! *i*) = *fst* (*half-segments-of-aform X* ! *Suc i*)
⟨*proof*⟩

lemma *polychain-half-segments-of-aform*: *polychain* (*half-segments-of-aform X*)
⟨*proof*⟩

lemma *fst-half-segments*:
half-segments-of-aform X ≠ [] ⟹
fst (*half-segments-of-aform X* ! *0*) = *lowest-vertex* (*fst X*, *nlex-pdevs* (*snd X*))
⟨*proof*⟩

lemma *nlex-half-segments-of-aform*: (*a*, *b*) ∈ *set* (*half-segments-of-aform X*) ⟹ *lex b a*
⟨*proof*⟩

lemma *ccw-half-segments-of-aform-all*:
assumes *cd*: (*c*, *d*) ∈ *set* (*half-segments-of-aform X*)
shows *list-all* (λ(*xi*, *xj*). *ccw xi xj c* ∧ *ccw xi xj d*) (*half-segments-of-aform X*)
⟨*proof*⟩

lemma *ccw-half-segments-of-aform*:
assumes *ij*: (*xi*, *xj*) ∈ *set* (*half-segments-of-aform X*)
assumes *c*: (*c*, *d*) ∈ *set* (*half-segments-of-aform X*)
shows *ccw xi xj c ccw xi xj d*
⟨*proof*⟩

lemma *half-segments-of-aform1*:
assumes *ch*: *x* ∈ *convex hull set* (*map fst* (*half-segments-of-aform X*))
assumes *ab*: (*a*, *b*) ∈ *set* (*half-segments-of-aform X*)
shows *ccw a b x*
⟨*proof*⟩

lemma *half-segments-of-aform2*:
assumes *ch*: *x* ∈ *convex hull set* (*map snd* (*half-segments-of-aform X*))
assumes *ab*: (*a*, *b*) ∈ *set* (*half-segments-of-aform X*)
shows *ccw a b x*
⟨*proof*⟩

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*⟩

lemma *fst-hd-half-segments-of-aform*:
assumes *half-segments-of-aform X* ≠ []
shows *fst* (*hd* (*half-segments-of-aform X*)) = *lowest-vertex* (*fst X*, (*nlex-pdevs* (*snd X*)))
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *rev-zip*: *length xs* = *length ys* ⟹ *rev* (*zip xs ys*) = *zip* (*rev xs*) (*rev ys*)
⟨*proof*⟩

lemma *zip-upt-self-aux*: *zip* [*0*..<*length xs*] *xs* = *map* (λ*i*. (*i*, *xs* ! *i*)) [*0*..<*length xs*]
⟨*proof*⟩

lemma *half-segments-of-aform-strict*:
assumes *e* ∈ *UNIV* → {−*1* <..< *1*}
assumes *seg* ∈ *set* (*half-segments-of-aform X*)
assumes *length* (*half-segments-of-aform X*) ≠ *1*
shows *ccw'* (*fst seg*) (*snd seg*) (*aform-val e X*)
⟨*proof*⟩

lemma *half-segments-of-aform-strict-all*:
assumes *e* ∈ *UNIV* → {−*1* <..< *1*}
assumes *length* (*half-segments-of-aform X*) ≠ *1*
shows *list-all* (λ*seg*. *ccw'* (*fst seg*) (*snd seg*) (*aform-val e X*)) (*half-segments-of-aform X*)
⟨*proof*⟩

lemma *list-allD*: *list-all P xs* ⟹ *x* ∈ *set xs* ⟹ *P x*
⟨*proof*⟩

lemma *minus-one-less-multI*:
fixes *e x*::*real*
shows − *1* ≤ *e* ⟹ *0* < *x* ⟹ *x* < *1* ⟹ − *1* < *e* ∗ *x*
⟨*proof*⟩

lemma *less-one-multI*:
fixes *e x*::*real*
shows *e* ≤ *1* ⟹ *0* < *x* ⟹ *x* < *1* ⟹ *e* ∗ *x* < *1*
⟨*proof*⟩

lemma *ccw-half-segments-of-aform-strictI*:
assumes *e* ∈ *UNIV* → {−*1* <..< *1*}
assumes (*s1*, *s2*) ∈ *set* (*half-segments-of-aform X*)
assumes *length* (*half-segments-of-aform X*) ≠ *1*
assumes *x* = (*aform-val e X*)
shows *ccw' s1 s2 x*
⟨*proof*⟩

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*))

⟨*proof*⟩

lemma *ccw'-sortedP-uminus*: *ccw'.sortedP 0 xs* ⟹ *ccw'.sortedP 0* (*map uminus xs*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *convex-polychain-half-segments-of-aform*: *convex-polychain* (*half-segments-of-aform X*)
⟨*proof*⟩

lemma *hd-distinct-neq-last*: *distinct xs* ⟹ *length xs* > *1* ⟹ *hd xs* ≠ *last xs*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *le-less-Suc-eq*: *x* − *Suc 0* ≤ (*i*::*nat*) ⟹ *i* < *x* ⟹ *x* − *Suc 0* = *i*
⟨*proof*⟩

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*))]
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma
lex-half-segments-lowest-vertex:

assumes $(c,\ d) \in$ *set* (*half-segments-of-aform* X)
shows *lex* d (*lowest-vertex* (*fst* X, *nlex-pdevs* (*snd* X)))
⟨*proof*⟩

lemma
lex-half-segments-lowest-vertex':
assumes $d \in$ *set* (*map snd* (*half-segments-of-aform* X))
shows *lex* d (*lowest-vertex* (*fst* X, *nlex-pdevs* (*snd* X)))
⟨*proof*⟩

lemma
lex-half-segments-last:
assumes $(c,\ d) \in$ *set* (*half-segments-of-aform* X)
shows *lex* (*snd* (*last* (*half-segments-of-aform* X))) d
⟨*proof*⟩

lemma
lex-half-segments-last':
assumes $d \in$ *set* (*map snd* (*half-segments-of-aform* X))
shows *lex* (*snd* (*last* (*half-segments-of-aform* X))) d
⟨*proof*⟩

lemma
ccw'-half-segments-lowest-last:
assumes *set-butlast*: $(c,\ d) \in$ *set* (*butlast* (*half-segments-of-aform* X))
assumes *ne*: *inl* (*snd* X) $\neq$ []
shows *ccw'* (*lowest-vertex* (*fst* X, *nlex-pdevs* (*snd* X))) d (*snd* (*last* (*half-segments-of-aform* X)))
⟨*proof*⟩

lemma *distinct-fst-half-segments*:
distinct (*map fst* (*half-segments-of-aform* X))
⟨*proof*⟩

lemma *distinct-snd-half-segments*:
distinct (*map snd* (*half-segments-of-aform* X))
⟨*proof*⟩

10.8 Mirror

definition *mirror-point* $x\ y = 2 *_R x - y$

lemma *ccw'-mirror-point3*[*simp*]:
ccw' (*mirror-point* $x\ y$) (*mirror-point* $x\ z$) (*mirror-point* $x\ w$) $\longleftrightarrow$ *ccw'* $y\ z\ w$
⟨*proof*⟩

lemma *mirror-point-self-inverse*[*simp*]:
fixes x::'a::*real-vector*
shows *mirror-point* p (*mirror-point* $p\ x$) $= x$

⟨*proof*⟩

lemma *mirror-half-segments-of-aform*:
assumes $e \in UNIV \to \{-1 <..< 1\}$
assumes *length* (*half-segments-of-aform X*) ≠ *1*
shows *list-all* (λ*seg*. *ccw′* (*fst seg*) (*snd seg*) (*aform-val e X*))
(*map* (*pairself* (*mirror-point* (*fst X*))) (*half-segments-of-aform X*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *convex-polychain-map-mirror*:
assumes *convex-polychain hs*
shows *convex-polychain* (*map* (*pairself* (*mirror-point x*)) *hs*)
⟨*proof*⟩

lemma *ccw′-mirror-point0*:
ccw′ (*mirror-point x y*) *z w* ⟷ *ccw′ y* (*mirror-point x z*) (*mirror-point x w*)
⟨*proof*⟩

lemma *ccw′-sortedP-mirror*:
ccw′.sortedP x0 (*map* (*mirror-point p0*) *xs*) ⟷ *ccw′.sortedP* (*mirror-point p0 x0*) *xs*
⟨*proof*⟩

lemma *ccw′-sortedP-mirror2*:
ccw′.sortedP (*mirror-point p0 x0*) (*map* (*mirror-point p0*) *xs*) ⟷ *ccw′.sortedP x0 xs*
⟨*proof*⟩

lemma *map-mirror-o-snd-polychain-of-eq*: *map* (*mirror-point x0* ∘ *snd*) (*polychain-of y xs*) =
map snd (*polychain-of* (*mirror-point x0 y*) (*map uminus xs*))
⟨*proof*⟩

lemma *lowest-vertex-eq-mirror-last*:
half-segments-of-aform X ≠ [] ⟹
(*lowest-vertex* (*fst X*, *nlex-pdevs* (*snd X*))) =
mirror-point (*fst X*) (*snd* (*last* (*half-segments-of-aform X*)))
⟨*proof*⟩

lemma *snd-last*: *xs* ≠ [] ⟹ *snd* (*last xs*) = *last* (*map snd xs*)
⟨*proof*⟩

lemma *mirror-point-snd-last-eq-lowest*:

half-segments-of-aform X $\neq$ [] $\Longrightarrow$
mirror-point (*fst X*) (*last* (*map snd* (*half-segments-of-aform X*))) =
lowest-vertex (*fst X*, *nlex-pdevs* (*snd X*))
⟨*proof*⟩

lemma *lex-mirror-point*: *lex* (*mirror-point x0 a*) (*mirror-point x0 b*) $\Longrightarrow$ *lex b a*
⟨*proof*⟩

lemma *ccw'-mirror-point*:
ccw' (*mirror-point x0 a*) (*mirror-point x0 b*) (*mirror-point x0 c*) $\Longrightarrow$ *ccw' a b c*
⟨*proof*⟩

lemma *inj-mirror-point*: *inj* (*mirror-point* (*x*::'*a*::*real-vector*))
⟨*proof*⟩

lemma
distinct-map-mirror-point-eq:
distinct (*map* (*mirror-point* (*x*::'*a*::*real-vector*)) *xs*) = *distinct xs*
⟨*proof*⟩

lemma *eq-self-mirror-iff*: **fixes** *x*::'*a*::*real-vector* **shows** *x* = *mirror-point y x* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

10.9 Full Segments

definition *segments-of-aform*::*point aform* $\Rightarrow$ (*point* $*$ *point*) *list*
where *segments-of-aform X* =
(*let hs* = *half-segments-of-aform X in hs* @ *map* (*pairself* (*mirror-point* (*fst X*))) *hs*)

lemma *segments-of-aform-strict*:
assumes *e* $\in$ *UNIV* $\rightarrow$ {−*1* <..< *1*}
assumes *length* (*half-segments-of-aform X*) $\neq$ *1*
shows *list-all* (λ*seg*. *ccw'* (*fst seg*) (*snd seg*) (*aform-val e X*)) (*segments-of-aform X*)
⟨*proof*⟩

lemma *mirror-point-aform-val*[*simp*]: *mirror-point* (*fst X*) (*aform-val e X*) = *aform-val* (− *e*) *X*
⟨*proof*⟩

lemma
in-set-segments-of-aform-aform-valE:
assumes (*x2*, *y2*) $\in$ *set* (*segments-of-aform X*)
obtains *e* **where** *y2* = *aform-val e X e* $\in$ *UNIV* $\rightarrow$ {−*1* .. *1*}
⟨*proof*⟩

lemma
last-half-segments-eq-mirror-hd:
assumes *half-segments-of-aform X* ≠ []
shows *snd* (*last* (*half-segments-of-aform X*)) = *mirror-point* (*fst X*) (*fst* (*hd* (*half-segments-of-aform X*)))
⟨*proof*⟩

lemma *polychain-segments-of-aform*: *polychain* (*segments-of-aform X*)
⟨*proof*⟩

lemma *segments-of-aform-closed*:
assumes *segments-of-aform X* ≠ []
shows *fst* (*hd* (*segments-of-aform X*)) = *snd* (*last* (*segments-of-aform X*))
⟨*proof*⟩

lemma *convex-polychain-segments-of-aform*:
assumes *1* < *length* (*half-segments-of-aform X*)
shows *convex-polychain* (*segments-of-aform X*)
⟨*proof*⟩

lemma *convex-polygon-segments-of-aform*:
assumes *1* < *length* (*half-segments-of-aform X*)
shows *convex-polygon* (*segments-of-aform X*)
⟨*proof*⟩

lemma
previous-segments-of-aformE:
assumes (*y*, *z*) ∈ *set* (*segments-of-aform X*)
obtains *x* **where** (*x*, *y*) ∈ *set* (*segments-of-aform X*)
⟨*proof*⟩

lemma *fst-compose-pairself*: *fst o pairself f* = *f o fst*
and *snd-compose-pairself*: *snd o pairself f* = *f o snd*
⟨*proof*⟩

lemma *in-set-butlastI*: *xs* ≠ [] ⟹ *x* ∈ *set xs* ⟹ *x* ≠ *last xs* ⟹ *x* ∈ *set* (*butlast xs*)
⟨*proof*⟩

lemma *distinct-in-set-butlastD*:
x ∈ *set* (*butlast xs*) ⟹ *distinct xs* ⟹ *x* ≠ *last xs*
⟨*proof*⟩

lemma *distinct-in-set-tlD*:
x ∈ *set* (*tl xs*) ⟹ *distinct xs* ⟹ *x* ≠ *hd xs*
⟨*proof*⟩

lemma *ccw'-sortedP-snd-segments-of-aform*:
assumes *length* (*inl* (*snd X*)) > *1*

shows
ccw'.sortedP (*lowest-vertex* (*fst X*, *nlex-pdevs* (*snd X*)))
(*butlast* (*map snd* (*segments-of-aform X*)))
⟨*proof*⟩

lemma *polychain-of-segments-of-aform1*:
assumes *length* (*segments-of-aform X*) = *1*
shows *False*
⟨*proof*⟩

lemma *polychain-of-segments-of-aform2*:
assumes *segments-of-aform X* = [*x*, *y*]
assumes *between* (*fst x*, *snd x*) *p*
shows *p* ∈ *convex hull set* (*map fst* (*segments-of-aform X*))
⟨*proof*⟩

lemma *append-eq-2*:
assumes *length xs* = *length ys*
shows *xs* @ *ys* = [*x*, *y*] ⟷ *xs* = [*x*] ∧ *ys* = [*y*]
⟨*proof*⟩

lemma *segments-of-aform-line-segment*:
assumes *segments-of-aform X* = [*x*, *y*]
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
shows *aform-val e X* ∈ *closed-segment* (*fst x*) (*snd x*)
⟨*proof*⟩

10.10 Continuous Generalization

lemma *LIMSEQ-minus-fract-mult*:
(λ*n*. *r* ∗ (*1* − *1* / *real* (*Suc* (*Suc n*)))) ⟶ *r*
⟨*proof*⟩

lemma *det3-nonneg-segments-of-aform*:
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *length* (*half-segments-of-aform X*) ≠ *1*
shows *list-all* (λ*seg*. *det3* (*fst seg*) (*snd seg*) (*aform-val e X*) ≥ *0*) (*segments-of-aform X*)
⟨*proof*⟩

lemma *det3-nonneg-segments-of-aformI*:
assumes *e* ∈ *UNIV* → {−*1* .. *1*}
assumes *length* (*half-segments-of-aform X*) ≠ *1*
assumes *seg* ∈ *set* (*segments-of-aform X*)
shows *det3* (*fst seg*) (*snd seg*) (*aform-val e X*) ≥ *0*
⟨*proof*⟩

10.11 Intersection of Vertical Line with Segment

fun *intersect-segment-xline'*::*nat* ⇒ *point* ∗ *point* ⇒ *real* ⇒ *point option*

where *intersect-segment-xline′ p* ((*x0*, *y0*), (*x1*, *y1*)) *xl* =
(*if x0* ≤ *xl* ∧ *xl* ≤ *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*
⟨*proof*⟩

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*⟩

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*)}
⟨*proof*⟩

lemma
in-segment-fst-le:
fixes *x0 x1 b*::*real*
assumes *x0* ≤ *x1* (*x*, *b*) ∈ *closed-segment* (*x0*, *y0*) (*x1*, *y1*)
shows *x* ≤ *x1*
⟨*proof*⟩

lemma
in-segment-fst-ge:
fixes *x0 x1 b*::*real*
assumes *x0* ≤ *x1* (*x*, *b*) ∈ *closed-segment* (*x0*, *y0*) (*x1*, *y1*)
shows *x0* ≤ *x*
⟨*proof*⟩

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* $= x\} = \{\}$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *intersect-segment-xline*:
assumes *intersect-segment-xline prec* $(p0, p1)$ *x* = *Some* (m, M)
shows *closed-segment p0 p1* $\cap$ $\{p.$ *fst p* $= x\} \subseteq \{(x, m)$ *..* $(x, M)\}$
⟨*proof*⟩

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* $= x\} \subseteq \{(x, m)$ *..* $(x, M)\}$
⟨*proof*⟩

lemma *intersect-segment-xline-eq-None*:
assumes *intersect-segment-xline prec* $(p0, p1)$ *x* = *None*
shows *closed-segment p0 p1* $\cap$ $\{p.$ *fst p* $= x\} = \{\}$
⟨*proof*⟩

lemma *inter-image-empty-iff*: $(X \cap \{p.\ f\ p = x\} = \{\}) \longleftrightarrow (x \notin f\ `\ X)$
⟨*proof*⟩

lemma *fst-closed-segment*[*simp*]: *fst* ‘ *closed-segment a b* = *closed-segment* (*fst a*) (*fst b*)
⟨*proof*⟩

lemma *intersect-segment-xline-eq-empty*:
fixes *p0 p1*::*real* $*$ *real*
assumes *closed-segment p0 p1* $\cap$ $\{p.$ *fst p* $= x\} = \{\}$
shows *intersect-segment-xline prec* $(p0, p1)$ *x* = *None*
⟨*proof*⟩

lemma *intersect-segment-xline-le*:
assumes *intersect-segment-xline prec y xl* = *Some* $(m0, M0)$
shows *m0* $\leq$ *M0*
⟨*proof*⟩

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* $= x\} = \{\}$

⟨*proof*⟩

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 ⇒ *intersect-segment-xline prec X xl*
| *Some* (*m*, *M*) ⇒
(*case intersect-segment-xline prec X xl of*
None ⇒ *Some* (*m*, *M*)
| *Some* (*m'*, *M'*) ⇒ *Some* (*min m' m*, *max M' M*)))

lemma
bound-intersect-2d-eq-None:
assumes *bound-intersect-2d prec Xs x* = *None*
assumes *X* ∈ *set Xs*
shows *intersect-segment-xline prec X x* = *None*
⟨*proof*⟩

lemma *bound-intersect-2d-upper*:
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*⟩

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*⟩

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*⟩

lemma *bound-intersect-2d-eq-None-iff*:
bound-intersect-2d prec Xs x = *None* ⟷ (∀ *X*∈*set Xs*. *intersect-segment-xline prec X x* = *None*)
⟨*proof*⟩

lemma *bound-intersect-2d-nonempty*:
assumes *bound-intersect-2d prec Xs x = Some (m, M)*
shows $(\bigcup (p1, p2) \in set\ Xs.\ closed\text{-}segment\ p1\ p2) \cap \{p.\ fst\ p = x\} \neq \{\}$
⟨*proof*⟩

lemma *bound-intersect-2d-le*:
assumes *bound-intersect-2d prec Xs x = Some (m, M)* **shows** $m \leq M$
⟨*proof*⟩

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*
[] ⇒ *if fst (fst X) = xl then let m = snd (fst X) in Some (m, m) else None*
| [*x*, *y*] ⇒ *intersect-segment-xline prec x xl*
| *xs* ⇒
(*case bound-intersect-2d prec (filter* (λ((*x1*, *y1*), *x2*, *y2*). *x1* < *x2*) *xs*) *xl of*
Some (*m*, *M'*) ⇒
(*case bound-intersect-2d prec (filter* (λ((*x1*, *y1*), *x2*, *y2*). *x1* > *x2*) *xs*) *xl of*
Some (*m'*, *M*) ⇒ *Some* (*min m m'*, *max M M'*)
| *None* ⇒ *None*)
| *None* ⇒ *None*))

lemma *list-cases4*:
$\bigwedge x\ P.\ (x = [] \Longrightarrow P) \Longrightarrow (\bigwedge y.\ x = [y] \Longrightarrow P) \Longrightarrow$
$(\bigwedge y\ z.\ x = [y, z] \Longrightarrow P) \Longrightarrow$
$(\bigwedge w\ y\ z\ zs.\ x = w \# y \# z \# zs \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *bound-intersect-2d-ud-segments-of-aform-le*:
bound-intersect-2d-ud prec X xl = Some (m0, M0) ⟹ $m0 \leq M0$
⟨*proof*⟩

lemma *pdevs-domain-eq-empty-iff*[*simp*]: *pdevs-domain (snd X) = {}* ⟷ *snd X = zero-pdevs*
⟨*proof*⟩

lemma *ccw-contr-on-line-left*:
assumes *det3 (a, b) (x, c) (x, d)* ≥ *0 a* > *x*
shows $d \leq c$
⟨*proof*⟩

lemma *ccw-contr-on-line-right*:
assumes *det3 (a, b) (x, c) (x, d)* ≥ *0 a* < *x*
shows $d \geq c$
⟨*proof*⟩

lemma *singleton-contrE*:

assumes $\bigwedge x\ y.\ x \neq y \Longrightarrow x \in X \Longrightarrow y \in X \Longrightarrow \mathit{False}$
assumes $X \neq \{\}$
obtains x **where** $X = \{x\}$
⟨*proof*⟩

lemma *segment-intersection-singleton*:
fixes *xl* **and** *a b*::*real* ∗ *real*
defines $i \equiv \mathit{closed\text{-}segment}\ a\ b \cap \{p.\ \mathit{fst}\ p = xl\}$
assumes *ne1*: $\mathit{fst}\ a \neq \mathit{fst}\ b$
assumes *upper*: $i \neq \{\}$
obtains *p1* **where** $i = \{p1\}$
⟨*proof*⟩

lemma *bound-intersect-2d-ud-segments-of-aform*:
assumes *bound-intersect-2d-ud prec X xl* = *Some* (*m0*, *M0*)
assumes $e \in \mathit{UNIV} \to \{-1\ ..\ 1\}$
shows $\{\mathit{aform\text{-}val}\ e\ X\} \cap \{x.\ \mathit{fst}\ x = xl\} \subseteq \{(xl,\ m0)\ ..\ (xl,\ M0)\}$
⟨*proof*⟩

10.14 Approximation from Orthogonal Directions

definition *inter-aform-plane-ortho*::
nat ⇒ *'a*::*executable-euclidean-space aform* ⇒ *'a* ⇒ *real* ⇒ *'a aform option*
where
inter-aform-plane-ortho p Z n g = *do* {
mMs ← *those* (*map* (λ*b*. *bound-intersect-2d-ud p* (*inner2-aform Z n b*) *g*) *Basis-list*);
let l = (∑ (*b*, *m*)←*zip Basis-list* (*map fst mMs*). *m* ∗$_R$ *b*);
let u = (∑ (*b*, *M*)←*zip Basis-list* (*map snd mMs*). *M* ∗$_R$ *b*);
Some (*aform-of-ivl l u*)
}

lemma
those-eq-SomeD:
assumes *those* (*map f xs*) = *Some ys*
shows *ys* = *map* (*the* ∘ *f*) *xs* ∧ (∀ *i*.∃ *y*. *i* < *length xs* ⟶ *f* (*xs* ! *i*) = *Some y*)
⟨*proof*⟩

lemma
sum-list-zip-map:
assumes *distinct xs*
shows (∑ (*x*, *y*)←*zip xs* (*map g xs*). *f x y*) = (∑ *x*∈*set xs*. *f x* (*g x*))
⟨*proof*⟩

lemma
inter-aform-plane-ortho-overappr:
assumes *inter-aform-plane-ortho p Z n g* = *Some X*
shows {*x*. ∀ *i* ∈ *Basis*. *x* ∙ *i* ∈ {*y*. (*g*, *y*) ∈ (λ*x*. (*x* ∙ *n*, *x* ∙ *i*)) ‘ *Affine Z*}} ⊆ *Affine X*

⟨*proof*⟩

lemma *inter-proj-eq*:
fixes n g l
defines $G \equiv \{x.\ x \cdot n = g\}$
shows $(\lambda x.\ x \cdot l)\ `\ (Z \cap G) =$
$\{y.\ (g,\ y) \in (\lambda x.\ (x \cdot n,\ x \cdot l))\ `\ Z\}$
⟨*proof*⟩

lemma
inter-overappr:
fixes n γ l
shows $Z \cap \{x.\ x \cdot n = g\} \subseteq \{x.\ \forall\, i \in Basis.\ x \cdot i \in \{y.\ (g,\ y) \in (\lambda x.\ (x \cdot n,\ x \cdot i))\ `\ Z\}\}$
⟨*proof*⟩

lemma *inter-inter-aform-plane-ortho*:
assumes *inter-aform-plane-ortho p Z n g = Some X*
shows *Affine Z* $\cap$ $\{x.\ x \cdot n = g\} \subseteq$ *Affine X*
⟨*proof*⟩

10.15 "Completeness" of Intersection

abbreviation *encompasses x seg* $\equiv$ *det3* (*fst seg*) (*snd seg*) $x \geq 0$

lemma *encompasses-cases*:
encompasses x seg $\vee$ *encompasses x* (*snd seg*, *fst seg*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *triangle-encompassing-polychain-of*:
assumes *det3 p a b* ≥ 0 *det3 p b c* ≥ 0 *det3 p c a* ≥ 0
assumes *ccw' a b c*
shows $p \in$ *convex hull* $\{a,\ b,\ c\}$
⟨*proof*⟩

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 $p \in$ *convex hull* (*set* (*map fst* (*x#y#z#zs*)))
⟨*proof*⟩

lemma *segments-of-aform-empty-Affine-eq*:

assumes *segments-of-aform X* = []
shows *Affine X* = {*fst X*}
⟨*proof*⟩

lemma *not-segments-of-aform-singleton*: *segments-of-aform X* ≠ [*x*]
⟨*proof*⟩

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*⟩

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*::*linorder* × '*b*) *list*. *distinct* (*map fst xs*) ∧ *sorted* (*rev* (*map fst xs*))}
⟨*proof*⟩

setup-lifting *type-definition-slist*

lift-definition *map-of-slist*::(*nat*, '*a*::*zero*) *slist* ⇒ *nat* ⇒ '*a option* **is** *map-of*
⟨*proof*⟩

lemma *finite-dom-map-of-slist*[*intro*, *simp*]: *finite* (*dom* (*map-of-slist xs*))
⟨*proof*⟩

abbreviation *the-default a x* ≡ (*case x of None* ⇒ *a* | *Some b* ⇒ *b*)

definition *Pdevs-raw xs i* = *the-default 0* (*map-of xs i*)

lemma *nonzeros-Pdevs-raw-subset*: {*i*. *Pdevs-raw xs i* ≠ *0*} ⊆ *dom* (*map-of xs*)
⟨*proof*⟩

lift-definition *Pdevs*::(*nat*, '*a*::*zero*) *slist* ⇒ '*a pdevs*
is *Pdevs-raw*
⟨*proof*⟩

code-datatype *Pdevs*

11.2 Degree

primrec *degree-list*::(*nat* × *'a*::*zero*) *list* ⇒ *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* ⇒ *nat* **is** *degree-list* ⟨*proof*⟩

lemma *degree-list-eq-zeroD*:
  assumes *degree-list xs* = *0*
  shows *the-default 0* (*map-of xs i*) = *0*
  ⟨*proof*⟩

lemma *degree-slist-eq-zeroD*: *degree-slist xs* = *0* ⟹ *degree* (*Pdevs xs*) = *0*
  ⟨*proof*⟩

lemma *degree-slist-eq-SucD*: *degree-slist xs* = *Suc n* ⟹ *pdevs-apply* (*Pdevs xs*) *n* ≠ *0*
⟨*proof*⟩

lemma *degree-slist-zero*:
  degree-slist xs = *n* ⟹ *n* ≤ *j* ⟹ *pdevs-apply* (*Pdevs xs*) *j* = *0*
⟨*proof*⟩

lemma *compute-degree*[*code*]: *degree* (*Pdevs xs*) = *degree-slist xs*
  ⟨*proof*⟩

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*) =
    (*if* (*i* = *j*) *then* (*i*, *f x y*) # *binop f z1 z2 xs ys*
    else if (*i* > *j*) *then* (*i*, *f x z2*) # *binop f z1 z2 xs* ((*j*, *y*)#*ys*)
    else (*j*, *f z1 y*) # *binop f z1 z2* ((*i*, *x*)#*xs*) *ys*)

lemma *set-binop-elemD1*:
  (*a*, *b*) ∈ *set* (*binop f z1 z2 xs ys*) ⟹ (*a* ∈ *set* (*map fst xs*) ∨ *a* ∈ *set* (*map fst ys*))
  ⟨*proof*⟩

lemma *set-binop-elemD2*:
  (*a*, *b*) ∈ *set* (*binop f z1 z2 xs ys*) ⟹
    (∃ *x*∈*set* (*map snd xs*). *b* = *f x z2*) ∨
    (∃ *y*∈*set* (*map snd ys*). *b* = *f z1 y*) ∨

$(\exists\, x{\in}set\ (map\ snd\ xs).\ \exists\, y{\in}set\ (map\ snd\ ys).\ b = f\ x\ y)$
⟨*proof*⟩

abbreviation *rsorted*≡$\lambda x.\ sorted\ (rev\ x)$

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*)))
⟨*proof*⟩

lemma *distinct-binop*:
fixes *xs*::('*a*::*linorder* ∗ '*b*) *list* **and** *ys*::('*a*::*linorder* ∗ '*c*) *list*
assumes *distinct* (*map fst xs*)
assumes *distinct* (*map fst ys*)
assumes *rsorted* ((*map fst xs*))
assumes *rsorted* ((*map fst ys*))
shows *distinct* (*map fst* (*binop f z1 z2 xs ys*))
⟨*proof*⟩

lemma *binop-plus*:
fixes *b*::(*nat* ∗ '*a*::*euclidean-space*) *list*
shows
$(\sum (i,\ y){\leftarrow}binop\ (+)\ 0\ 0\ b\ ba.\ e\ i\ *_R\ y) = (\sum (i,\ y){\leftarrow}b.\ e\ i\ *_R\ y) + (\sum (i,\ y){\leftarrow}ba.\ e\ i\ *_R\ y)$
⟨*proof*⟩

lemma *binop-compose*:
binop ($\lambda x\ y.\ f\ (g\ x\ y)$) *z1 z2 xs ys* = *map* (*apsnd f*) (*binop g z1 z2 xs ys*)
⟨*proof*⟩

lemma *linear-cmul-left*[*intro*, *simp*]: *linear* ((∗) *x*::*real* ⇒ -)
⟨*proof*⟩

lemma *length-merge-sorted-eq*:
length (*binop f z1 z2 xs ys*) = *length* (*binop g y1 y2 xs ys*)
⟨*proof*⟩

11.4 Pointswise Addition

lift-definition *add-slist*::(*nat*, '*a*::{*plus*, *zero*}) *slist* ⇒ (*nat*, '*a*) *slist* ⇒ (*nat*, '*a*) *slist* **is**
$\lambda xs\ ys.\ binop\ (+)\ 0\ 0\ xs\ ys$
⟨*proof*⟩

lemma *map-of-binop*[*simp*]: *rsorted* (*map fst xs*) ⟹ *rsorted* (*map fst ys*) ⟹
distinct (*map fst xs*) ⟹ *distinct* (*map fst ys*) ⟹
map-of (*binop f z1 z2 xs ys*) *i* =

(*case map-of xs i of*
Some x ⇒ Some (f x (case map-of ys i of Some x ⇒ x | None ⇒ z2))
| None ⇒ (case map-of ys i of Some y ⇒ Some (f z1 y) | None ⇒ None))
⟨*proof*⟩

lemma *pdevs-apply-Pdevs-add-slist*[*simp*]:
fixes *xs ys*::(*nat*, *'a*::*monoid-add*) *slist*
shows *pdevs-apply* (*Pdevs* (*add-slist xs ys*)) *i* =
pdevs-apply (*Pdevs xs*) *i* + *pdevs-apply* (*Pdevs ys*) *i*
⟨*proof*⟩

lemma *compute-add-pdevs*[*code*]: *add-pdevs* (*Pdevs xs*) (*Pdevs ys*) = *Pdevs* (*add-slist xs ys*)
⟨*proof*⟩

11.5 prod of pdevs

lift-definition *prod-slist*::(*nat*, *'a*::*zero*) *slist* ⇒ (*nat*, *'b*::*zero*) *slist* ⇒ (*nat*, (*'a* × *'b*)) *slist* **is**
λxs ys. binop Pair 0 0 xs ys
⟨*proof*⟩

lemma *pdevs-apply-Pdevs-prod-slist*[*simp*]:
pdevs-apply (*Pdevs* (*prod-slist xs ys*)) *i* = (*pdevs-apply* (*Pdevs xs*) *i*, *pdevs-apply* (*Pdevs ys*) *i*)
⟨*proof*⟩

lemma *compute-prod-of-pdevs*[*code*]: *prod-of-pdevs* (*Pdevs xs*) (*Pdevs ys*) = *Pdevs* (*prod-slist xs ys*)
⟨*proof*⟩

11.6 Set of Coefficients

lift-definition *set-slist*::(*nat*, *'a*::*real-vector*) *slist* ⇒ (*nat* ∗ *'a*) *set* **is** *set* ⟨*proof*⟩

lemma *finite-set-slist*[*intro*, *simp*]: *finite* (*set-slist xs*)
⟨*proof*⟩

11.7 Domain

lift-definition *list-of-slist*::(*'a*::*linorder*, *'b*::*zero*) *slist* ⇒ (*'a* ∗ *'b*) *list*
is *λxs. filter* (*λx. snd x* ≠ *0*) *xs* ⟨*proof*⟩

lemma *compute-pdevs-domain*[*code*]: *pdevs-domain* (*Pdevs xs*) = *set* (*map fst* (*list-of-slist xs*))
⟨*proof*⟩

lemma *sort-rev-eq-sort*: *distinct xs* ⟹ *sort* (*rev xs*) = *sort xs*
⟨*proof*⟩

lemma *compute-list-of-pdevs*[*code*]: *list-of-pdevs* (*Pdevs xs*) = *list-of-slist xs*
⟨*proof*⟩

lift-definition *slist-of-pdevs*::*'a pdevs* ⇒ (*nat*, *'a*::*real-vector*) *slist* **is** *list-of-pdevs*
⟨*proof*⟩

11.8 Application

lift-definition *slist-apply*::(*'a*::*linorder*, *'b*::*zero*) *slist* ⇒ *'a* ⇒ *'b* **is**
λ*xs i*. *the-default 0* (*map-of xs i*) ⟨*proof*⟩

lemma *compute-pdevs-apply*[*code*]: *pdevs-apply* (*Pdevs x*) *i* = *slist-apply x i*
⟨*proof*⟩

11.9 Total Deviation

lift-definition *tdev-slist*::(*nat*, *'a*::*ordered-euclidean-space*) *slist* ⇒ *'a* **is**
sum-list o map (*abs o snd*) ⟨*proof*⟩

lemma *tdev-slist-sum*: *tdev-slist xs* = *sum* (*abs* ∘ *snd*) (*set-slist xs*)
⟨*proof*⟩

lemma *pdevs-apply-set-slist*: *x* ∈ *set-slist xs* ⟹ *snd x* = *pdevs-apply* (*Pdevs xs*)
(*fst x*)
⟨*proof*⟩

lemma
tdev-list-eq-zeroI:
shows (⋀*i*. *pdevs-apply* (*Pdevs xs*) *i* = *0*) ⟹ *tdev-slist xs* = *0*
⟨*proof*⟩

lemma *inj-on-fst-set-slist*: *inj-on fst* (*set-slist xs*)
⟨*proof*⟩

lemma *pdevs-apply-Pdevs-eq-0*:
pdevs-apply (*Pdevs xs*) *i* = *0* ⟷ ((∀ *x*. (*i*, *x*) ∈ *set-slist xs* ⟶ *x* = *0*))
⟨*proof*⟩

lemma *compute-tdev*[*code*]: *tdev* (*Pdevs xs*) = *tdev-slist xs*
⟨*proof*⟩

11.10 Minkowski Sum

lemma *dropWhile-rsorted-eq-filter*:
rsorted (*map fst xs*) ⟹ *dropWhile* (λ(*i*, *x*). *i* ≥ (*m*::*nat*)) *xs* = *filter* (λ(*i*, *x*). *i*
< *m*) *xs*
(**is** - ⟹ *?lhs xs* = *?rhs xs*)
⟨*proof*⟩

lift-definition *msum-slist*::*nat* ⇒ (*nat*, *'a*) *slist* ⇒ (*nat*, *'a*) *slist* ⇒ (*nat*, *'a*) *slist*

is $\lambda m\ xs\ ys.\ map\ (apfst\ (\lambda n.\ n + m))\ ys\ @\ dropWhile\ (\lambda(i,\ x).\ i \geq m)\ xs$
⟨*proof*⟩

lemma *slist-apply-msum-slist*:
$slist\text{-}apply\ (msum\text{-}slist\ m\ xs\ ys)\ i =$
$(if\ i < m\ then\ slist\text{-}apply\ xs\ i\ else\ slist\text{-}apply\ ys\ (i - m))$
⟨*proof*⟩

lemma *pdevs-apply-msum-slist*:
$pdevs\text{-}apply\ (Pdevs\ (msum\text{-}slist\ m\ xs\ ys))\ i =$
$(if\ i < m\ then\ pdevs\text{-}apply\ (Pdevs\ xs)\ i\ else\ pdevs\text{-}apply\ (Pdevs\ ys)\ (i - m))$
⟨*proof*⟩

lemma *compute-msum-pdevs*[*code*]: $msum\text{-}pdevs\ m\ (Pdevs\ xs)\ (Pdevs\ ys) = Pdevs\ (msum\text{-}slist\ m\ xs\ ys)$
⟨*proof*⟩

11.11 Unary Operations

lift-definition $map\text{-}slist::('a \Rightarrow 'b) \Rightarrow (nat,\ 'a)\ slist \Rightarrow (nat,\ 'b)\ slist$ **is** $\lambda f.\ map\ (apsnd\ f)$
⟨*proof*⟩

lemma *pdevs-apply-map-slist*:
$f\ 0 = 0 \Longrightarrow pdevs\text{-}apply\ (Pdevs\ (map\text{-}slist\ f\ xs))\ i = f\ (pdevs\text{-}apply\ (Pdevs\ xs)\ i)$
⟨*proof*⟩

lemma *compute-scaleR-pdves*[*code*]: $scaleR\text{-}pdevs\ r\ (Pdevs\ xs) = Pdevs\ (map\text{-}slist\ (\lambda x.\ r *_R x)\ xs)$
and *compute-pdevs-scaleR*[*code*]: $pdevs\text{-}scaleR\ (Pdevs\ rs)\ x = Pdevs\ (map\text{-}slist\ (\lambda r.\ r *_R x)\ rs)$
and *compute-uminus-pdevs*[*code*]: $uminus\text{-}pdevs\ (Pdevs\ xs) = Pdevs\ (map\text{-}slist\ (\lambda x.\ -x)\ xs)$
and *compute-abs-pdevs*[*code*]: $abs\text{-}pdevs\ (Pdevs\ xs) = Pdevs\ (map\text{-}slist\ abs\ xs)$
and *compute-pdevs-inner*[*code*]: $pdevs\text{-}inner\ (Pdevs\ xs)\ b = Pdevs\ (map\text{-}slist\ (\lambda x.\ x \cdot b)\ xs)$
and *compute-pdevs-inner2*[*code*]:
$pdevs\text{-}inner2\ (Pdevs\ xs)\ b\ c = Pdevs\ (map\text{-}slist\ (\lambda x.\ (x \cdot b,\ x \cdot c))\ xs)$
and *compute-inner-scaleR-pdevs*[*code*]:
$inner\text{-}scaleR\text{-}pdevs\ x\ (Pdevs\ ys) = Pdevs\ (map\text{-}slist\ (\lambda y.\ (x \cdot y) *_R y)\ ys)$
and *compute-trunc-pdevs*[*code*]:
$trunc\text{-}pdevs\ p\ (Pdevs\ xs) = Pdevs\ (map\text{-}slist\ (\lambda x.\ eucl\text{-}truncate\text{-}down\ p\ x)\ xs)$
and *compute-trunc-err-pdevs*[*code*]:
$trunc\text{-}err\text{-}pdevs\ p\ (Pdevs\ xs) = Pdevs\ (map\text{-}slist\ (\lambda x.\ eucl\text{-}truncate\text{-}down\ p\ x - x)\ xs)$
⟨*proof*⟩

11.12 Filter

lift-definition *filter-slist*::(*nat* ⇒ *'a* ⇒ *bool*) ⇒ (*nat*, *'a*) *slist* ⇒ (*nat*, *'a*) *slist*
is λ*P xs*. *filter* (λ(*i*, *x*). (*P i x*)) *xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compute-filter-pdevs*[*code*]: *filter-pdevs P* (*Pdevs xs*) = *Pdevs* (*filter-slist P xs*)
⟨*proof*⟩

11.13 Constant

lift-definition *zero-slist*::(*nat*, *'a*) *slist* **is** [] ⟨*proof*⟩

lemma *compute-zero-pdevs*[*code*]: *zero-pdevs* = *Pdevs* (*zero-slist*)
⟨*proof*⟩

lift-definition *One-slist*::(*nat*, *'a*::*executable-euclidean-space*) *slist*
is *rev* (*zip* [*0*..<*length* (*Basis-list*::*'a list*)] (*Basis-list*::*'a list*))
⟨*proof*⟩

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*⟩

lemma
map-of-rev-zip-upto-length-eq-None:
assumes ¬*i* < *length B*
assumes *d* = *length B*
shows (*map-of* (*rev* (*zip* [*0*..<*d*] *B*)) *i*) = *None*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *compute-One-pdevs*[*code*]: *One-pdevs* = *Pdevs One-slist*

⟨*proof*⟩

lift-definition *coord-slist*::*nat* ⇒ (*nat*, *real*) *slist* **is** λ*i*. [(*i*, *1*)] ⟨*proof*⟩

lemma *compute-coord-pdevs*[*code*]: *coord-pdevs i* = *Pdevs* (*coord-slist i*)
⟨*proof*⟩

11.14 Update

primrec *update-list*::*nat* ⇒ *'a* ⇒ (*nat* ∗ *'a*) *list* ⇒ (*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*)
⟨*proof*⟩

lemma *in-set-update-listD*:
assumes *y* ∈ *set* (*update-list n x ys*)
shows *y* = (*n*, *x*) ∨ (*y* ∈ *set ys*)
⟨*proof*⟩

lemma *in-set-update-listI*:
y = (*n*, *x*) ∨ (*fst y* ≠ *n* ∧ *y* ∈ *set ys*) ⟹ *y* ∈ *set* (*update-list n x ys*)
⟨*proof*⟩

lemma *in-set-update-list*: (*n*, *x*) ∈ *set* (*update-list n x xs*)
⟨*proof*⟩

lemma *overwrite-update-list*: (*a*, *b*) ∈ *set xs* ⟹ (*a*, *b*) ∉ *set* (*update-list n x xs*) ⟹ *a* = *n*
⟨*proof*⟩

lemma *insert-update-list*:
distinct (*map fst xs*) ⟹ *rsorted* (*map fst xs*) ⟹ (*a*, *b*) ∈ *set* (*update-list a x xs*) ⟹ *b* = *x*
⟨*proof*⟩

lemma *set-update-list-eq*: *distinct* (*map fst xs*) ⟹ *rsorted* (*map fst xs*) ⟹
set (*update-list n x xs*) = *insert* (*n*, *x*) (*set xs* − {*x*. *fst x* = *n*})
⟨*proof*⟩

lift-definition *update-slist*::*nat* ⇒ *'a* ⇒ (*nat*, *'a*) *slist* ⇒ (*nat*, *'a*) *slist* **is** *update-list*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compute-pdev-upd*[*code*]: *pdev-upd* (*Pdevs xs*) *n x* = *Pdevs* (*update-slist n x xs*)
⟨*proof*⟩

11.15 Approximate Total Deviation

lift-definition *fold-slist*::('*a* ⇒ '*b* ⇒ '*b*) ⇒ (*nat*, '*a*::*zero*) *slist* ⇒ '*b* ⇒ '*b*
is λ*f xs z*. *fold* (*f o snd*) (*filter* (λ*x*. *snd x* ≠ *0*) *xs*) *z* ⟨*proof*⟩

lemma *Pdevs-raw-Cons*:
Pdevs-raw ((*a*, *b*) # *xs*) = (λ*i*. *if i* = *a then b else Pdevs-raw xs i*)
⟨*proof*⟩

lemma *zeros-aux*: − (λ*i*. *if i* = *a then b else Pdevs-raw xs i*) −‘ {*0*} ⊆
− *Pdevs-raw xs* −‘ {*0*} ∪ {*a*}
⟨*proof*⟩

lemma *compute-tdev'*[*code*]:
tdev' p (*Pdevs xs*) = *fold-slist* (λ*a b*. *eucl-truncate-up p* (|*a*| + *b*)) *xs 0*
⟨*proof*⟩

11.16 Equality

lemma *slist-apply-list-of-slist-eq*: *slist-apply a i* = *the-default 0* (*map-of* (*list-of-slist a*) *i*)
⟨*proof*⟩

lemma *compute-equal-pdevs*[*code*]:
equal-class.equal (*Pdevs a*) (*Pdevs b*) ⟷ (*list-of-slist a*) = (*list-of-slist b*)
⟨*proof*⟩

11.17 From List of Generators

lift-definition *slist-of-list*::'*a*::*zero list* ⇒ (*nat*, '*a*) *slist*
is λ*xs*. *rev* (*zip* [*0*..<*length xs*] *xs*)
⟨*proof*⟩

lemma *slist-apply-slist-of-list*:
slist-apply (*slist-of-list xs*) *i* = (*if i* < *length xs then xs* ! *i else 0*)
⟨*proof*⟩

lemma *compute-pdevs-of-list*[*code*]: *pdevs-of-list xs* = *Pdevs* (*slist-of-list xs*)
⟨*proof*⟩

abstraction function which can be used in code equations

lift-definition *abs-slist-if*::('*a*::*linorder*×'*b*) *list* ⇒ ('*a*, '*b*) *slist*
is λ*list*. *if distinct* (*map fst list*) ∧ *rsorted* (*map fst list*) *then list else* []
⟨*proof*⟩

definition *slist* = *Abs-slist*

lemma [*code-post*]: *Abs-slist* = *slist*
⟨*proof*⟩

lemma [*code*]: *slist* = (λ*xs*.
(*if distinct* (*map fst xs*) ∧ *rsorted* (*map fst xs*) *then abs-slist-if xs else Code.abort* (*STR* ''') (λ-. *slist xs*)))
⟨*proof*⟩

abbreviation *pdevs* ≡ (λ*x*. *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*))
⟨*proof*⟩

lemma *Pdevs-raw-map*: *f 0* = *0* ⟹ *Pdevs-raw* (*map* (λ(*i*, *x*). (*i*, *f x*)) *xs*) *i* = *f* (*Pdevs-raw xs i*)
⟨*proof*⟩

lemma *compute-nlex-pdevs*[*code*]: *nlex-pdevs* (*Pdevs x*) = *Pdevs* (*nlex-slist x*)
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma [*code*]: *log2* (*int-of-integer a*) = *int-of-integer* (*log2-integer a*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compute-float-plus*[*code*]: *Float m1 e1* + *Float m2 e2* =
(*if m1* = *0 then Float m2 e2 else if m2* = *0 then Float m1 e1 else*
if e1 ≤ *e2 then Float* (*m1* + *m2* ∗ *power-int 2* (*e2* − *e1*)) *e1*
else Float (*m2* + *m1* ∗ *power-int 2* (*e1* − *e2*)) *e2*)
⟨*proof*⟩

lemma *compute-real-of-float*[*code*]:
real-of-float (*Float m e*) = (*if e* ≥ *0 then m* ∗ *2* ^ *nat e else m* / *power-int 2* (−*e*))
⟨*proof*⟩

lemma *compute-float-down*[*code*]:
float-down p (*Float m e*) =
(*if p* + *e* < *0 then Float* (*m div power-int 2* (−(*p* + *e*))) (−*p*) *else Float m e*)
⟨*proof*⟩

lemma *compute-lapprox-posrat*[*code*]:
fixes *prec*::*nat* **and** *x y*::*nat*
shows *lapprox-posrat prec x y* =
(*let*
l = *rat-precision prec x y*;
d = *if 0* ≤ *l then int x* ∗ *power-int 2 l div y else int x div power-int 2* (− *l*) *div y*
in normfloat (*Float d* (− *l*)))
⟨*proof*⟩

lemma *compute-rapprox-posrat*[*code*]:
fixes *prec x y*
defines *l* ≡ *rat-precision prec x y*
shows *rapprox-posrat prec x y* = (*let*
l = *l* ;
(*r*, *s*) = *if 0* ≤ *l then* (*int x* ∗ *power-int 2 l*, *int y*) *else* (*int x*, *int y* ∗ *power-int 2* (−*l*)) ;
d = *r div s* ;
m = *r mod s*
in normfloat (*Float* (*d* + (*if m* = *0* ∨ *y* = *0 then 0 else 1*)) (− *l*)))
⟨*proof*⟩

lemma *compute-float-truncate-down*[*code*]:
float-round-down prec (*Float m e*) = (*let d* = *bitlen* (*abs m*) − *int prec* − *1 in*
if 0 < *d then let P* = *power-int 2 d* ; *n* = *m div P in Float n* (*e* + *d*)
else Float m e)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

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.open'-output* (*fn os => f* (*File.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
⟨*proof*⟩

lemma *showsp-float-append* [*show-law-simps*]:
showsp-float p r (*x* @ *y*) = *showsp-float p r x* @ *y*
⟨*proof*⟩

⟨*ML*⟩

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** e *999*)

partial-function (*tailrec*) *normalize-float10*
  where [*code*]: *normalize-float10 f* =
    (*if mantissa10 f mod 10* ≠ *0* ∨ *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-notation* **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 ← *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 ← *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* ⟨*proof*⟩

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 (λ*f*. (*let* (*d*, *e*) = *a in* (λ*y*. *f* (*c d e y*))) *b*)
  ⟨*proof*⟩

definition *divmod-int*::*int* ⇒ *int* ⇒ *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 ∨ d ≠ 0) (if d ≠ 0 ∨ c then p − 1 else p) r
      (digits * 10 + d) (cnt + 1)) e)
declare f2f10-frac.simps[code]

definition float2-float10::int ⇒ bool ⇒ int ⇒ int ⇒ (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 ∧ rd) ∨ (sgn = −1 ∧ ¬ 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 ≠ 0) p r 0 0 e else (0,0,0);

    digits = if round-down ∨ 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* ($\lambda x.\ x$)) *o*
show-digits mp o
(*if ms* = [] *then* ($\lambda x.\ x$) *else shows-string* ''.'' *o show-digits ms*) *o*
(*if e* = 0 *then* ($\lambda x.\ x$) *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*)
⟨*proof*⟩

lemma *show-law-float10* [*show-law-intros*]: *show-law showsp-float10 r*
⟨*proof*⟩

lemma *showsp-float10-append* [*show-law-simps*]:
showsp-float10 p r (*x* @ *y*) = *showsp-float10 p r x* @ *y*
⟨*proof*⟩

⟨*ML*⟩

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*
⟨*proof*⟩

⟨*ML*⟩
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 '''' '''' ''⏎'' ''⏎'' (*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 ≡ shows-box-of-aforms-hr x ''''

definition shows-aforms-hr— human readable
where shows-aforms-hr XS = shows (generators XS)

abbreviation show-aform-hr x ≡ 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 :: nat ⇒ float
where
```

float-of-nat ≡ *of-nat*

abbreviation
float-of-int :: *int* ⇒ *float*
where
float-of-int ≡ *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*)
⟨*proof*⟩

lemma *zero-real-code* [*code*]:
0 = *real-of-float 0*
⟨*proof*⟩

lemma *one-real-code* [*code*]:
1 = *real-of-float 1*
⟨*proof*⟩

lemma [*code-abbrev*]:
(*real-of-float* (*of-int a*) :: *real*) = (*Ratreal* (*Rat.of-int a*) :: *real*)
⟨*proof*⟩

lemma [*code-abbrev*]:
real-of-float 0 ≡ *Ratreal 0*
⟨*proof*⟩

lemma [*code-abbrev*]:
real-of-float 1 = *Ratreal 1*
⟨*proof*⟩

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*) ⟷ *HOL.equal x y*
⟨*proof*⟩

abbreviation *FloatR*::*int*⇒*int*⇒*real* **where**
FloatR a b ≡ *real-of-float* (*Float a b*)

lemma *real-less-eq-code'* [*code*]: *real-of-float x* ≤ *real-of-float y* ⟷ *x* ≤ *y*
and *real-less-code'* [*code*]: *real-of-float x* < *real-of-float y* ⟷ *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''*) (λ-. *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''*) (λ-. *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*]: *abs* (*real-of-float x*) = *real-of-float* (*abs x*)
⟨*proof*⟩

lemma *compute-round-down*[*code*]: *round-down prec* (*real-of-float f*) = *real-of-float* (*float-down prec f*)
⟨*proof*⟩

lemma *compute-round-up*[*code*]: *round-up prec* (*real-of-float f*) = *real-of-float* (*float-up prec f*)
⟨*proof*⟩

lemma *compute-truncate-down*[*code*]:
truncate-down prec (*real-of-float f*) = *real-of-float* (*float-round-down prec f*)
⟨*proof*⟩

lemma *compute-truncate-up*[*code*]:
truncate-up prec (*real-of-float f*) = *real-of-float* (*float-round-up prec f*)
⟨*proof*⟩

lemma [*code*]: *real-divl p* (*real-of-float x*) (*real-of-float y*) = *real-of-float* (*float-divl p x y*)
⟨*proof*⟩

lemma [*code*]: *real-divr p* (*real-of-float x*) (*real-of-float y*) = *real-of-float* (*float-divr p x y*)
⟨*proof*⟩

lemmas [*code*] = *real-of-float-inverse*

end

16 Examples

theory *Ex-Affine-Approximation*
imports
Affine-Code
Print
Float-Real
begin

context includes *floatarith-notation* **begin**

definition *rotate-fas* =
[*Cos* (*Rad-of* (*Var 2*)) ∗ *Var 0* − *Sin* (*Rad-of* (*Var 2*)) ∗ *Var 1*,
Sin (*Rad-of* (*Var 2*)) ∗ *Var 0* + *Cos* (*Rad-of* (*Var 2*)) ∗ *Var 1*]

definition *rotate-slp* = *slp-of-fas rotate-fas*
definition *approx-rotate p t X* = *approx-slp-outer p 3 rotate-slp X*

fun *rotate-aform* **where**
rotate-aform x i = (*let r* = (((*the o* (λ*x*. *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* (λ*x*. *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* ''# '' ∘ *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 $R \neq None \wedge 0 < Inf\text{-}aform'\ p\ (the\ R)$
then let - = *if prnt* $\neq$ [] *then print* (*STR* ''# *Success*↩'') *else* () *in prove-pos prnt i p F XS*
else let - = *if prnt* $\neq$ [] *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* (λ*p xs*.
case approx-slp-outer p 1 slp xs of None ⇒ *None* | *Some* [*x*] ⇒ *Some x* | *Some* - ⇒ *None*) *xs*)

experiment begin

unbundle *floatarith-notation*

The examples below are taken from 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* ^ *10*) + (*Var 0* − (*Var 1*)^$_e$*2*)^$_e$*2* + (*Var 1* − *1*)^$_e$*2* + (*Var 0* − (*Var 2*)^$_e$*2*)^$_e$*2* + (*Var 2* − *1*)^$_e$*2*

lemma *schwefel*:
$5.8806 / 10 \hat{}\ 10 + (x0 - (x1)^2)^2 + (x1 - 1)^2 + (x0 - (x2)^2)^2 + (x2 - 1)^2$
=
interpret-floatarith schwefel [*x0*, *x1*, *x2*]
⟨*proof*⟩

lemma *prove-pos-slp* [] *30 schwefel 100000* [*aforms-of-ivls* [−*10*,−*10*,−*10*] [*10*,*10*,*10*]]
⟨*proof*⟩

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*

$- x3 * x4 * x5) = interpret\text{-}floatarith\ delta6\ [x0, x1, x2, x3, x4, x5]$
⟨*proof*⟩

lemma *prove-pos-slp [] 20 delta6 10000 [aforms-of-ivls (replicate 6 4) (replicate 6 (FloatR 104045 (−14)))]*
⟨*proof*⟩

definition $caprasse = (3.1801 + - Var\ 0 * (Var\ 2)\ \hat{}_e\ 3 + 4 * Var\ 1 * (Var\ 2)\hat{}_e 2 * Var\ 3 +$
$4 * Var\ 0 * Var\ 2 * (Var\ 3)\hat{}_e 2 + 2 * Var\ 1 * (Var\ 3)\hat{}_e 3 + 4 * Var\ 0 * Var\ 2 + 4 * (Var\ 2)\hat{}_e 2 - 10 * Var\ 1 * Var\ 3 +$
$-10 * (Var\ 3)\hat{}_e 2 + 2)$

schematic-goal *caprasse*:
$(3.1801 + - xs!0 * (xs!2)\ \hat{}\ 3 + 4 * xs!1 * (xs!2)^2 * xs!3 +$
$4 * xs!0 * xs!2 * (xs!3)^2 + 2 * xs!1 * (xs!3)\hat{}3 + 4 * xs!0 * xs!2 + 4 * (xs!2)^2 - 10 * xs!1 * xs!3 +$
$-10 * (xs!3)^2 + 2) = interpret\text{-}floatarith\ caprasse\ xs$
⟨*proof*⟩

lemma *prove-pos-slp [] 20 caprasse 10000 [aforms-of-ivls (replicate 4 (1/2)) (replicate 4 (1/2))]*
⟨*proof*⟩

definition *magnetism* =
$0.25001 + (Var\ 0)\hat{}_e 2 + 2 * (Var\ 1)\hat{}_e 2 + 2 * (Var\ 2)\hat{}_e 2 + 2 * (Var\ 3)\hat{}_e 2 + 2 * (Var\ 4)\hat{}_e 2 + 2 * (Var\ 5)\hat{}_e 2 +$
$2 * (Var\ 6)\hat{}_e 2 - Var\ 0$
schematic-goal *magnetism*:
$0.25001 + (xs!0)^2 + 2 * (xs!1)^2 + 2 * (xs!2)^2 + 2 * (xs!3)^2 + 2 * (xs!4)^2 + 2 * (xs!5)^2 +$
$2 * (xs!6)^2 - xs!0 = interpret\text{-}floatarith\ magnetism\ xs$
⟨*proof*⟩

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*∗*real*∗*real*∗*real*∗*real*∗*real*∗*real*∗*real*∗*real*∗
real∗*real*∗*real*∗*real*∗*real*∗*real*∗*real*∗*real*∗*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*⟩

Timings

⟨*ML*⟩

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.