Taylor Models

Christoph Traut and Fabian Immler

November 9, 2026

Abstract

We present a formally verified implementation of multivariate Taylor models. Taylor models are a form of rigorous polynomial approximation, consisting of an approximation polynomial based on Taylor expansions, combined with a rigorous bound on the approximation error. Taylor models were introduced as a tool to mitigate the dependency problem of interval arithmetic. Our implementation automatically computes Taylor models for the class of elementary functions, expressed by composition of arithmetic operations and basic functions like exp, sin, or square root.

Contents

1 Topology for Floating Point Numbers **1**

2 Horner Evaluation **2**

3 Splitting polynomials to reduce floating point precision **6**

4 Splitting polynomials by degree **7**

5 Multivariate Taylor Models **11**
- 5.1 Computing interval bounds on arithmetic expressions 11
- 5.2 Definition of Taylor models and notion of rangeity 12
- 5.3 Interval bounds for Taylor models 13
- 5.4 Computing taylor models for basic, univariate functions . . . 16
  - 5.4.1 Derivations of floatarith expressions 16
  - 5.4.2 Computing Taylor models for arbitrary univariate expressions 17
- 5.5 Operations on Taylor models 20
- 5.6 Computing Taylor models for multivariate expressions 29
- 5.7 Computing bounds for floatarith expressions 32

1 Topology for Floating Point Numbers

theory *Float-Topology*
  imports
    HOL−Analysis.Multivariate-Analysis
    HOL−Library.Float
begin

This topology is totally disconnected and not complete, in which sense is it useful? Perhaps for convergence of intervals?

unbundle *float.lifting*

instantiation *float* :: *dist*
begin

lift-definition *dist-float* :: *float* ⇒ *float* ⇒ *real* **is** *dist* ⟨*proof*⟩

lemma *dist-float-eq-0-iff*: (*dist x y* = *0*) = (*x* = *y*) **for** *x y*::*float*
  ⟨*proof*⟩

lemma *dist-float-triangle2*: *dist x y* ≤ *dist x z* + *dist y z* **for** *x y z*::*float*
  ⟨*proof*⟩

instance ⟨*proof*⟩
end

instantiation *float* :: *uniformity*
begin

definition *uniformity-float* :: (*float* × *float*) *filter*
  where *uniformity-float* = (*INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist x y* < *e*})

instance ⟨*proof*⟩
end

lemma *float-dense-in-real*:
  fixes *x* :: *real*
  assumes *x* < *y*
  shows ∃ *r*∈*float*. *x* < *r* ∧ *r* < *y*
⟨*proof*⟩

lemma *real-of-float-dense*:
  fixes *x y* :: *real*
  assumes *x* < *y*
  shows ∃ *q* :: *float*. *x* < *real-of-float q* ∧ *real-of-float q* < *y*
  ⟨*proof*⟩

instantiation *float* :: *linorder-topology*
begin

definition *open-float*::*float set* ⇒ *bool* **where**
  open-float S = (∀ *x*∈*S*. ∃ *e*>*0*. ∀ *y*. *dist y x* < *e* ⟶ *y* ∈ *S*)

instance
⟨*proof*⟩

end

instance *float* :: *metric-space*
⟨*proof*⟩

instance *float*::*topological-ab-group-add*
⟨*proof*⟩

lifting-update *float.lifting*
lifting-forget *float.lifting*

end

2 Horner Evaluation

theory *Horner-Eval*
  imports *HOL−Library.Interval*
begin

Function and lemmas for evaluating polynomials via the horner scheme. Because interval multiplication is not distributive, interval polynomials expressed as a sum of monomials are not equivalent to their respective horner form. The functions and lemmas in this theory can be used to express interval polynomials in horner form and prove facts about them.

fun *horner-eval′* **where**
  horner-eval′ f x v 0 = *v*
| *horner-eval′ f x v* (*Suc i*) = *horner-eval′ f x* (*f i* + *x* ∗ *v*) *i*

definition *horner-eval*
  where *horner-eval f x n* = *horner-eval′ f x 0 n*

lemma *horner-eval-cong*:
  assumes ⋀*i*. *i* < *n* ⟹ *f i* = *g i*
  assumes *x* = *y*
  assumes *n* = *m*
  shows *horner-eval f x n* = *horner-eval g y m*
⟨*proof*⟩

lemma *horner-eval-eq-setsum*:
  fixes *x*::′*a*::*linordered-idom*
  shows *horner-eval f x n* = (∑ *i*<*n*. *f i* ∗ *x*^*i*)

⟨*proof*⟩

lemma *horner-eval-Suc*[*simp*]:
fixes *x*::*'a*::*linordered-idom*
shows *horner-eval f x* (*Suc n*) = *horner-eval f x n* + (*f n*) ∗ *x*^*n*
⟨*proof*⟩

lemma *horner-eval-Suc'*[*simp*]:
fixes *x*::*'a*::{*comm-monoid-add*, *times*}
shows *horner-eval f x* (*Suc n*) = *f 0* + *x* ∗ (*horner-eval* (λ*i*. *f* (*Suc i*)) *x n*)
⟨*proof*⟩

lemma *horner-eval-0*[*simp*]:
shows *horner-eval f x 0* = *0*
⟨*proof*⟩

lemma *horner-eval'-interval*:
fixes *x*::*'a*::*linordered-ring*
assumes ⋀*i*. *i* < *n* ⟹ *f i* ∈ *set-of* (*g i*)
assumes *x* ∈*i* *I v* ∈*i* *V*
shows *horner-eval' f x v n* ∈*i* *horner-eval' g I V n*
⟨*proof*⟩

lemma *horner-eval-interval*:
fixes *x*::*'a*::*linordered-idom*
assumes ⋀*i*. *i* < *n* ⟹ *f i* ∈ *set-of* (*g i*)
assumes *x* ∈ *set-of I*
shows *horner-eval f x n* ∈*i* *horner-eval g I n*
⟨*proof*⟩

end
theory *Polynomial-Expression-Additional*
imports
Polynomial-Expression
HOL−Decision-Procs.Approximation
begin

lemma *real-of-float-eq-zero-iff*[*simp*]: *real-of-float x* = *0* ⟷ *x* = *0*
⟨*proof*⟩

Theory *Taylor-Models.Polynomial-Expression* contains a, more or less, 1:1 generalization of theory *Multivariate-Polynomial*. Any additions belong here.

declare [[*coercion-map map-poly*]]
declare [[*coercion interval-of*::*float*⇒*float interval*]]

Apply float interval arguments to a float poly.

value *Ipoly* [*Ivl* (*Float 4* (−*6*)) (*Float 10 6*)] (*poly.Add* (*poly.C* (*Float 3 5*)) (*poly.Bound 0*))

map-poly for homomorphisms

lemma *map-poly-homo-polyadd-eq-zero-iff*:
map-poly f (*p* +$_p$ *q*) = 0$_p$ ⟷ *p* +$_p$ *q* = 0$_p$
if [*symmetric*, *simp*]: ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y* ⋀*x*. *f x* = 0 ⟷ *x* = 0
⟨*proof*⟩

lemma *zero-iffD*: (⋀*x*. *f x* = 0 ⟷ *x* = 0) ⟹ *f* 0 = 0
⟨*proof*⟩

lemma *map-poly-homo-polyadd*:
map-poly f (*p1* +$_p$ *p2*) = *map-poly f p1* +$_p$ *map-poly f p2*
if [*simp*]: ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y* ⋀*x*. *f x* = 0 ⟷ *x* = 0
⟨*proof*⟩

lemma *map-poly-homo-polyneg*:
map-poly f (~$_p$ *p1*) = ~$_p$ (*map-poly f p1*)
if [*simp*]: ⋀*x y*. *f* (− *x*) = − *f x*
⟨*proof*⟩

lemma *map-poly-homo-polysub*:
map-poly f (*p1* −$_p$ *p2*) = *map-poly f p1* −$_p$ *map-poly f p2*
if [*simp*]: ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y* ⋀*x*. *f x* = 0 ⟷ *x* = 0 ⋀*x y*. *f* (− *x*) =
− *f x*
⟨*proof*⟩

lemma *map-poly-homo-polymul*:
map-poly f (*p1* ∗$_p$ *p2*) = *map-poly f p1* ∗$_p$ *map-poly f p2*
if [*simp*]: ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y* ⋀*x*. *f x* = 0 ⟷ *x* = 0 ⋀*x y*. *f* (*x* ∗ *y*)
= *f x* ∗ *f y*
⟨*proof*⟩

lemma *map-poly-homo-polypow*:
map-poly f (*p1* ^$_p$ *n*) = *map-poly f p1* ^$_p$ *n*
if [*simp*]: ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y* ⋀*x*. *f x* = 0 ⟷ *x* = 0 ⋀*x y*. *f* (*x* ∗ *y*)
= *f x* ∗ *f y*
f 1 = 1
⟨*proof*⟩

lemmas *map-poly-homo-polyarith* = *map-poly-homo-polyadd map-poly-homo-polyneg*
map-poly-homo-polysub map-poly-homo-polymul map-poly-homo-polypow

Count the number of parameters of a polynomial.

fun *num-params* :: *'a poly* ⇒ *nat*
where *num-params* (*poly.C c*) = 0
| *num-params* (*poly.Bound n*) = *Suc n*
| *num-params* (*poly.Add a b*) = *max* (*num-params a*) (*num-params b*)
| *num-params* (*poly.Sub a b*) = *max* (*num-params a*) (*num-params b*)
| *num-params* (*poly.Mul a b*) = *max* (*num-params a*) (*num-params b*)
| *num-params* (*poly.Neg a*) = *num-params a*

| *num-params* (*poly.Pw a n*) = *num-params a*
| *num-params* (*poly.CN a n b*) = *max* (*max* (*num-params a*) (*num-params b*)) (*Suc n*)

lemma *num-params-map-poly*[*simp*]:
shows *num-params* (*map-poly f p*) = *num-params p*
⟨*proof*⟩

lemma *num-params-polyadd*:
shows *num-params* (*p1* $+_p$ *p2*) ≤ *max* (*num-params p1*) (*num-params p2*)
⟨*proof*⟩

lemma *num-params-polyneg*:
shows *num-params* ($\sim_p$ *p*) = *num-params p*
⟨*proof*⟩

lemma *num-params-polymul*:
shows *num-params* (*p1* $*_p$ *p2*) ≤ *max* (*num-params p1*) (*num-params p2*)
⟨*proof*⟩

lemma *num-params-polypow*:
shows *num-params* (*p* $\hat{}_p$ *n*) ≤ *num-params p*
⟨*proof*⟩

lemma *num-params-polynate*:
shows *num-params* (*polynate p*) ≤ *num-params p*
⟨*proof*⟩

lemma *polynate-map-poly-real*[*simp*]:
fixes *p* :: *float poly*
shows *map-poly real-of-float* (*polynate p*) = *polynate* (*map-poly real-of-float p*)
⟨*proof*⟩

Evaluating a float poly is equivalent to evaluating the corresponding real poly with the float parameters converted to reals.

lemma *Ipoly-real-float-eqiv*:
fixes *p*::*float poly* **and** *xs*::*float list*
assumes *num-params p* ≤ *length xs*
shows *Ipoly xs* (*p*::*real poly*) = *Ipoly xs p*
⟨*proof*⟩

Evaluating an *'a poly* with *'a interval* arguments is monotone.

lemma *Ipoly-interval-args-mono*:
fixes *p*::*'a*::*linordered-idom poly*
and *x*::*'a list*
and *xs*::*'a interval list*
assumes *x all-in*$_i$ *xs*
assumes *num-params p* ≤ *length xs*
shows *Ipoly x p* ∈ *set-of* (*Ipoly xs* (*map-poly interval-of p*))

⟨*proof*⟩

lemma *Ipoly-interval-args-inc-mono*:
fixes *p*::*'a*::{*real-normed-algebra*, *linear-continuum-topology*, *linordered-idom*} *poly*
and *I*::*'a interval list* **and** *J*::*'a interval list*
assumes *num-params p* ≤ *length I*
assumes *I all-subset J*
shows *set-of* (*Ipoly I* (*map-poly interval-of p*)) ⊆ *set-of* (*Ipoly J* (*map-poly interval-of p*))
⟨*proof*⟩

3 Splitting polynomials to reduce floating point precision

TODO: Move this! Definitions regarding floating point numbers should not be in a theory about polynomials.

fun *float-prec* :: *float* ⇒ *int*
where *float-prec f* = (*let p*=*exponent f in if p* ≥ *0 then 0 else* −*p*)

fun *float-round* :: *nat* ⇒ *float* ⇒ *float*
where *float-round prec f* = (
let d = *float-down prec f*; *u* = *float-up prec f*
in if f − *d* < *u* − *f then d else u*)

Splits any polynomial p into two polynomials l, r, such that $\forall x{::}real.\ p(x) = l(x) + r(x)$ and all floating point coefficients in p are rounded to precision *prec*. Not all cases need to give good results. Polynomials normalized with polynate only contain *poly.C* and *poly.CN* constructors.

fun *split-by-prec* :: *nat* ⇒ *float poly* ⇒ *float poly* ∗ *float poly*
where *split-by-prec prec* (*poly.C f*) = (*let r* = *float-round prec f in* (*poly.C r*, *poly.C* (*f* − *r*)))
| *split-by-prec prec* (*poly.Bound n*) = (*poly.Bound n*, *poly.C 0*)
| *split-by-prec prec* (*poly.Add l r*) = (*let* (*ll*, *lr*) = *split-by-prec prec l*;
(*rl*, *rr*) = *split-by-prec prec r*
in (*poly.Add ll rl*, *poly.Add lr rr*))
| *split-by-prec prec* (*poly.Sub l r*) = (*let* (*ll*, *lr*) = *split-by-prec prec l*;
(*rl*, *rr*) = *split-by-prec prec r*
in (*poly.Sub ll rl*, *poly.Sub lr rr*))
| *split-by-prec prec* (*poly.Mul l r*) = (*let* (*ll*, *lr*) = *split-by-prec prec l*;
(*rl*, *rr*) = *split-by-prec prec r*
in (*poly.Mul ll rl*, *poly.Add* (*poly.Add* (*poly.Mul lr rl*) (*poly.Mul ll rr*)) (*poly.Mul lr rr*)))
| *split-by-prec prec* (*poly.Neg p*) = (*let* (*l*, *r*) = *split-by-prec prec p in* (*poly.Neg l*, *poly.Neg r*))
| *split-by-prec prec* (*poly.Pw p 0*) = (*poly.C 1*, *poly.C 0*)
| *split-by-prec prec* (*poly.Pw p* (*Suc n*)) = (*let* (*l*, *r*) = *split-by-prec prec p in* (*poly.Pw l n*, *poly.Sub* (*poly.Pw p* (*Suc n*)) (*poly.Pw l n*)))

| *split-by-prec prec* (*poly.CN c n p*) = (*let* (*cl*, *cr*) = *split-by-prec prec c*;
(*pl*, *pr*) = *split-by-prec prec p*
in (*poly.CN cl n pl*, *poly.CN cr n pr*))

TODO: Prove precision constraint on *l*.

lemma *split-by-prec-correct*:
fixes *args* :: *real list*
assumes (*l*, *r*) = *split-by-prec prec p*
shows *Ipoly args p* = *Ipoly args l* + *Ipoly args r* (**is** *?P1*)
and *num-params l* ≤ *num-params p* (**is** *?P2*)
and *num-params r* ≤ *num-params p* (**is** *?P3*)
⟨*proof*⟩

4 Splitting polynomials by degree

fun *maxdegree* :: ('*a*::*zero*) *poly* ⇒ *nat*
where *maxdegree* (*poly.C c*) = *0*
| *maxdegree* (*poly.Bound n*) = *1*
| *maxdegree* (*poly.Add l r*) = *max* (*maxdegree l*) (*maxdegree r*)
| *maxdegree* (*poly.Sub l r*) = *max* (*maxdegree l*) (*maxdegree r*)
| *maxdegree* (*poly.Mul l r*) = *maxdegree l* + *maxdegree r*
| *maxdegree* (*poly.Neg p*) = *maxdegree p*
| *maxdegree* (*poly.Pw p n*) = *n* ∗ *maxdegree p*
| *maxdegree* (*poly.CN c n p*) = *max* (*maxdegree c*) (*1* + *maxdegree p*)

fun *split-by-degree* :: *nat* ⇒ '*a*::*zero poly* ⇒ '*a poly* ∗ '*a poly*
where *split-by-degree n* (*poly.C c*) = (*poly.C c*, *poly.C 0*)
| *split-by-degree 0 p* = (*poly.C 0*, *p*)
| *split-by-degree* (*Suc n*) (*poly.CN c v p*) = (
let (*cl*, *cr*) = *split-by-degree* (*Suc n*) *c*;
(*pl*, *pr*) = *split-by-degree n p*
in (*poly.CN cl v pl*, *poly.CN cr v pr*))
— This function is only intended for use on polynomials in normal form. Hence most cases never get executed.
| *split-by-degree n p* = (*poly.C 0*, *p*)

lemma *split-by-degree-correct*:
fixes *x* :: *real list* **and** *p* :: *float poly*
assumes (*l*, *r*) = *split-by-degree ord p*
shows *maxdegree l* ≤ *ord* (**is** *?P1*)
and *Ipoly x p* = *Ipoly x l* + *Ipoly x r* (**is** *?P2*)
and *num-params l* ≤ *num-params p* (**is** *?P3*)
and *num-params r* ≤ *num-params p* (**is** *?P4*)
⟨*proof*⟩

Operations on lists.

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

lemma *map2-nth*[*simp*]:
assumes $n < length\ a$
assumes $n < length\ b$
shows $(map2\ f\ a\ b)!n = f\ (a!n)\ (b!n)$
⟨*proof*⟩

Translating a polynomial by a vector.

fun *poly-translate* :: *'a list* ⇒ *'a poly* ⇒ *'a poly*
where *poly-translate vs* (*poly.C c*) = *poly.C c*
| *poly-translate vs* (*poly.Bound n*) = *poly.Add* (*poly.Bound n*) (*poly.C* (*vs* ! *n*))
| *poly-translate vs* (*poly.Add l r*) = *poly.Add* (*poly-translate vs l*) (*poly-translate vs r*)
| *poly-translate vs* (*poly.Sub l r*) = *poly.Sub* (*poly-translate vs l*) (*poly-translate vs r*)
| *poly-translate vs* (*poly.Mul l r*) = *poly.Mul* (*poly-translate vs l*) (*poly-translate vs r*)
| *poly-translate vs* (*poly.Neg p*) = *poly.Neg* (*poly-translate vs p*)
| *poly-translate vs* (*poly.Pw p n*) = *poly.Pw* (*poly-translate vs p*) *n*
| *poly-translate vs* (*poly.CN c n p*) = *poly.Add* (*poly-translate vs c*) (*poly.Mul* (*poly.Add* (*poly.Bound n*) (*poly.C* (*vs* ! *n*))) (*poly-translate vs p*))

Translating a polynomial is equivalent to translating its argument.

lemma *poly-translate-correct*:
assumes *num-params p* ≤ *length x*
assumes *length x* = *length v*
shows *Ipoly x* (*poly-translate v p*) = *Ipoly* (*map2* (+) *x v*) *p*
⟨*proof*⟩

lemma *real-poly-translate*:
assumes *num-params p* ≤ *length v*
shows *Ipoly x* (*map-poly real-of-float* (*poly-translate v p*)) = *Ipoly x* (*poly-translate v* (*map-poly real-of-float p*))
⟨*proof*⟩

lemma *num-params-poly-translate*[*simp*]:
shows *num-params* (*poly-translate v p*) = *num-params p*
⟨*proof*⟩

end
theory *Taylor-Models-Misc*
imports
HOL−Library.Float
HOL−Library.Function-Algebras
HOL−Decision-Procs.Approximation
Affine-Arithmetic.Floatarith-Expression
begin

This theory contains anything that doesn't belong anywhere else.

lemma *of-nat-real-float-equiv*: $(of\text{-}nat\ n :: real) = (of\text{-}nat\ n :: float)$
⟨*proof*⟩

lemma *fact-real-float-equiv*: $(fact\ n :: float) = (fact\ n :: real)$
⟨*proof*⟩

lemma *Some-those-length*:
$those\ ys = Some\ xs \Longrightarrow length\ xs = length\ ys$
⟨*proof*⟩

lemma *those-eq-None-iff*: $those\ ys = None \longleftrightarrow None \in set\ ys$
⟨*proof*⟩

lemma *those-eq-Some-iff*: $those\ ys = (Some\ xs) \longleftrightarrow (ys = map\ Some\ xs)$
⟨*proof*⟩

lemma *Some-those-nth*:
assumes $those\ ys = Some\ xs$
assumes $i < length\ xs$
shows $Some\ (xs!i) = ys!i$
⟨*proof*⟩

lemma *fun-pow*: $f\hat{}n = (\lambda x.\ (f\ x)\hat{}n)$
⟨*proof*⟩

context includes *floatarith-syntax* **begin**

Translate floatarith expressions by a vector of floats.

fun *fa-translate* :: *float list* ⇒ *floatarith* ⇒ *floatarith*
where $fa\text{-}translate\ v\ (Add\ a\ b) = Add\ (fa\text{-}translate\ v\ a)\ (fa\text{-}translate\ v\ b)$
| $fa\text{-}translate\ v\ (Minus\ a) = Minus\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Mult\ a\ b) = Mult\ (fa\text{-}translate\ v\ a)\ (fa\text{-}translate\ v\ b)$
| $fa\text{-}translate\ v\ (Inverse\ a) = Inverse\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Cos\ a) = Cos\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Arctan\ a) = Arctan\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Min\ a\ b) = Min\ (fa\text{-}translate\ v\ a)\ (fa\text{-}translate\ v\ b)$
| $fa\text{-}translate\ v\ (Max\ a\ b) = Max\ (fa\text{-}translate\ v\ a)\ (fa\text{-}translate\ v\ b)$
| $fa\text{-}translate\ v\ (Abs\ a) = Abs\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Sqrt\ a) = Sqrt\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Exp\ a) = Exp\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Ln\ a) = Ln\ (fa\text{-}translate\ v\ a)$
| $fa\text{-}translate\ v\ (Var\ n) = Add\ (Var\ n)\ (Num\ (v!n))$
| $fa\text{-}translate\ v\ (Power\ a\ n) = Power\ (fa\text{-}translate\ v\ a)\ n$
| $fa\text{-}translate\ v\ (Powr\ a\ b) = Powr\ (fa\text{-}translate\ v\ a)\ (fa\text{-}translate\ v\ b)$
| $fa\text{-}translate\ v\ (Floor\ x) = Floor\ (fa\text{-}translate\ v\ x)$
| $fa\text{-}translate\ v\ (Num\ c) = Num\ c$
| $fa\text{-}translate\ v\ Pi = Pi$

lemma *fa-translate-correct*:

assumes *max-Var-floatarith f* ≤ *length I*
assumes *length v* = *length I*
shows *interpret-floatarith* (*fa-translate v f*) *I* = *interpret-floatarith f* (*map2* (+) *I v*)
⟨*proof*⟩

primrec *vars-floatarith* **where**
vars-floatarith (*Add a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Mult a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Inverse a*) = *vars-floatarith a*
| *vars-floatarith* (*Minus a*) = *vars-floatarith a*
| *vars-floatarith* (*Num a*) = {}
| *vars-floatarith* (*Var i*) = {*i*}
| *vars-floatarith* (*Cos a*) = *vars-floatarith a*
| *vars-floatarith* (*Arctan a*) = *vars-floatarith a*
| *vars-floatarith* (*Abs a*) = *vars-floatarith a*
| *vars-floatarith* (*Max a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Min a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Pi*) = {}
| *vars-floatarith* (*Sqrt a*) = *vars-floatarith a*
| *vars-floatarith* (*Exp a*) = *vars-floatarith a*
| *vars-floatarith* (*Powr a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Ln a*) = *vars-floatarith a*
| *vars-floatarith* (*Power a n*) = *vars-floatarith a*
| *vars-floatarith* (*Floor a*) = *vars-floatarith a*

lemma *finite-vars-floatarith*[*simp*]: *finite* (*vars-floatarith x*)
⟨*proof*⟩

end

lemma *max-Var-floatarith-eq-Max-vars-floatarith*:
max-Var-floatarith fa = (*if vars-floatarith fa* = {} *then 0 else Suc* (*Max* (*vars-floatarith fa*)))
⟨*proof*⟩

end
theory *Taylor-Models*
imports
Horner-Eval
Polynomial-Expression-Additional
Taylor-Models-Misc
HOL−Decision-Procs.Approximation
HOL−Library.Function-Algebras
HOL−Library.Set-Algebras
Affine-Arithmetic.Straight-Line-Program
Affine-Arithmetic.Affine-Approximation
begin

TODO: get rid of float poly/float inteval and use real poly/real interval and

data refinement?

5 Multivariate Taylor Models

5.1 Computing interval bounds on arithmetic expressions

This is a wrapper around the "approx" function. It computes range bounds on floatarith expressions.

fun *compute-bound-fa* :: *nat* ⇒ *floatarith* ⇒ *float interval list* ⇒ *float interval option*
  where *compute-bound-fa prec f I* = *approx prec f* (*map Some I*)

lemma *compute-bound-fa-correct*:
  interpret-floatarith f i ∈$_r$ *ivl*
  if *compute-bound-fa prec f I* = *Some ivl*
    i all-in I
  for *i*::*real list*
⟨*proof*⟩

5.2 Definition of Taylor models and notion of rangeity

Taylor models are a pair of a polynomial and an absolute error bound.

datatype *taylor-model* = *TaylorModel* (*tm-poly*: *float poly*) (*tm-bound*: *float interval*)

Taylor model for a real valuation of variables

primrec *insertion* :: (*nat* ⇒ *'a*) ⇒ *'a poly* ⇒ *'a*::{*plus*,*zero*,*minus*,*uminus*,*times*,*one*,*power*}
where
  insertion bs (*C c*) = *c*
| *insertion bs* (*poly.Bound n*) = *bs n*
| *insertion bs* (*Neg a*) = − *insertion bs a*
| *insertion bs* (*poly.Add a b*) = *insertion bs a* + *insertion bs b*
| *insertion bs* (*Sub a b*) = *insertion bs a* − *insertion bs b*
| *insertion bs* (*Mul a b*) = *insertion bs a* ∗ *insertion bs b*
| *insertion bs* (*Pw t n*) = *insertion bs t* ^ *n*
| *insertion bs* (*CN c n p*) = *insertion bs c* + (*bs n*) ∗ *insertion bs p*

definition *range-tm* :: (*nat* ⇒ *real*) ⇒ *taylor-model* ⇒ *real interval* **where**
range-tm e tm = *interval-of* (*insertion e* (*tm-poly tm*)) + *real-interval* (*tm-bound tm*)

lemma *Ipoly-num-params-cong*: *Ipoly xs p* = *Ipoly ys p*
  if ⋀*i*. *i* < *num-params p* ⟹ *xs* ! *i* = *ys* ! *i*
  ⟨*proof*⟩

lemma *insertion-num-params-cong*: *insertion e p* = *insertion f p*
  if ⋀*i*. *i* < *num-params p* ⟹ *e i* = *f i*

⟨*proof*⟩

lemma *insertion-eq-IpolyI*: *insertion xs p* = *Ipoly ys p*
if ⋀*i*. *i* < *num-params p* ⟹ *xs i* = *ys* ! *i*
⟨*proof*⟩

lemma *Ipoly-eq-insertionI*: *Ipoly ys p* = *insertion xs p*
if ⋀*i*. *i* < *num-params p* ⟹ *xs i* = *ys* ! *i*
⟨*proof*⟩

lemma *range-tmI*:
x ∈$_i$ *range-tm e tm*
if *x*: *x* ∈$_i$ *interval-of* (*insertion e* ((*tm-poly tm*))) + *real-interval* (*tm-bound tm*)
for *e*::*nat*⇒*real*
⟨*proof*⟩

lemma *range-tmD*:
x ∈$_i$ *interval-of* (*insertion e* (*tm-poly tm*)) + *real-interval* (*tm-bound tm*)
if *x* ∈$_i$ *range-tm e tm*
for *e*::*nat*⇒*real*
⟨*proof*⟩

5.3 Interval bounds for Taylor models

Bound a polynomial by simply approximating it with interval arguments.

fun *compute-bound-poly* :: *nat* ⇒ *float interval poly* ⇒ (*float interval list*) ⇒ (*float interval list*) ⇒ *float interval* **where**
compute-bound-poly prec (*poly.C f*) *I a* = *f*
| *compute-bound-poly prec* (*poly.Bound n*) *I a* = *round-interval prec* (*I* ! *n* − (*a* ! *n*))
| *compute-bound-poly prec* (*poly.Add p q*) *I a* =
round-interval prec (*compute-bound-poly prec p I a* + *compute-bound-poly prec q I a*)
| *compute-bound-poly prec* (*poly.Sub p q*) *I a* =
round-interval prec (*compute-bound-poly prec p I a* − *compute-bound-poly prec q I a*)
| *compute-bound-poly prec* (*poly.Mul p q*) *I a* =
mult-float-interval prec (*compute-bound-poly prec p I a*) (*compute-bound-poly prec q I a*)
| *compute-bound-poly prec* (*poly.Neg p*) *I a* = −*compute-bound-poly prec p I a*
| *compute-bound-poly prec* (*poly.Pw p n*) *I a* = *power-float-interval prec n* (*compute-bound-poly prec p I a*)
| *compute-bound-poly prec* (*poly.CN p n q*) *I a* =
round-interval prec (*compute-bound-poly prec p I a* +
mult-float-interval prec (*round-interval prec* (*I* ! *n* − (*a* ! *n*))) (*compute-bound-poly prec q I a*))

Bounds on Taylor models are simply a bound on its polynomial, widened by the approximation error.

fun *compute-bound-tm* :: *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *float interval*
where *compute-bound-tm prec I a* (*TaylorModel p e*) = *compute-bound-poly prec p I a* + *e*

lemma *compute-bound-tm-def*:
compute-bound-tm prec I a tm = *compute-bound-poly prec* (*tm-poly tm*) *I a* + (*tm-bound tm*)
⟨*proof*⟩

lemma *real-of-float-in-real-interval-of*[*intro*, *simp*]: *real-of-float x* ∈$_r$ *X* **if** *x* ∈$_i$ *X*
⟨*proof*⟩

lemma *in-set-of-round-interval*[*intro*, *simp*]:
x ∈$_r$ *round-interval prec X* **if** *x* ∈$_r$ *X*
⟨*proof*⟩

lemma *in-set-real-minus-interval*[*intro*, *simp*]:
x − *y* ∈$_r$ *X* − *Y* **if** *x* ∈$_r$ *X y* ∈$_r$ *Y*
⟨*proof*⟩

lemma *real-interval-plus*: *real-interval* (*a* + *b*) = *real-interval a* + *real-interval b*
⟨*proof*⟩

lemma *real-interval-uminus*: *real-interval* (− *b*) = − *real-interval b*
⟨*proof*⟩

lemma *real-interval-of*: *real-interval* (*interval-of b*) = *interval-of b*
⟨*proof*⟩

lemma *real-interval-minus*: *real-interval* (*a* − *b*) = *real-interval a* − *real-interval b*
⟨*proof*⟩

lemma *in-set-real-plus-interval*[*intro*, *simp*]:
x + *y* ∈$_r$ *X* + *Y* **if** *x* ∈$_r$ *X y* ∈$_r$ *Y*
⟨*proof*⟩

lemma *in-set-neg-plus-interval*[*intro*, *simp*]:
− *y* ∈$_r$ − *Y* **if** *y* ∈$_r$ *Y*
⟨*proof*⟩

lemma *in-set-real-times-interval*[*intro*, *simp*]:
x ∗ *y* ∈$_r$ *X* ∗ *Y* **if** *x* ∈$_r$ *X y* ∈$_r$ *Y*
⟨*proof*⟩

lemma *real-interval-one*: *real-interval 1* = *1*
⟨*proof*⟩

lemma *real-interval-zero*: *real-interval 0 = 0*
⟨*proof*⟩

lemma *real-interval-power*: *real-interval (a ^ b) = real-interval a ^ b*
⟨*proof*⟩

lemma *in-set-real-power-interval*[*intro*, *simp*]:
x ^ n ∈r X ^ n **if** *x ∈r X*
⟨*proof*⟩

lemma *power-float-interval-real-interval*[*intro*, *simp*]:
x ^ n ∈r power-float-interval prec n X **if** *x ∈r X*
⟨*proof*⟩

lemma *in-set-mult-float-interval*[*intro*, *simp*]:
*x * y ∈r mult-float-interval prec X Y* **if** *x ∈r X y ∈r Y*
⟨*proof*⟩

lemma *in-set-real-minus-swapI*: *e i ∈r I ! i − a ! i*
if *x − e i ∈r a ! i x ∈r I ! i*
⟨*proof*⟩

definition *develops-at-within*::(*nat ⇒ real*) *⇒ float interval list ⇒ float interval list ⇒ bool*
where *develops-at-within e a I ⟷ (a all-subset I) ∧ (∀ i < length I. e i ∈r I ! i − a ! i)*

lemma *develops-at-withinI*:
assumes *all-in*: *a all-subset I*
assumes *e*: *⋀i. i < length I ⟹ e i ∈r I ! i − a ! i*
shows *develops-at-within e a I*
⟨*proof*⟩

lemma *develops-at-withinD*:
assumes *develops-at-within e a I*
shows *a all-subset I*
⋀i. i < length I ⟹ e i ∈r I ! i − a ! i
⟨*proof*⟩

lemma *compute-bound-poly-correct*:
fixes *p*::*float poly*
assumes *num-params p ≤ length I*
assumes *dev*: *develops-at-within e a I*
shows *insertion e* (*p*::*real poly*) *∈r compute-bound-poly prec* (*map-poly interval-of p*) *I a*
⟨*proof*⟩

lemma *compute-bound-tm-correct*:
fixes *I* :: *float interval list* **and** *f* :: *real list ⇒ real*

assumes *n*: *num-params* (*tm-poly t*) ≤ *length I*
assumes *dev*: *develops-at-within e a I*
assumes *x0*: *x0* ∈$_i$ *range-tm e t*
shows *x0* ∈$_r$ *compute-bound-tm prec I a t*
⟨*proof*⟩

lemma *compute-bound-tm-correct-subset*:
fixes *I* :: *float interval list* **and** *f* :: *real list* ⇒ *real*
assumes *n*: *num-params* (*tm-poly t*) ≤ *length I*
assumes *dev*: *develops-at-within e a I*
shows *set-of* (*range-tm e t*) ⊆ *set-of* (*real-interval* (*compute-bound-tm prec I a t*))
⟨*proof*⟩

lemma *compute-bound-poly-mono*:
assumes *num-params p* ≤ *length I*
assumes *mem*: *I all-subset J a all-subset I*
shows *set-of* (*compute-bound-poly prec p I a*) ⊆ *set-of* (*compute-bound-poly prec p J a*)
⟨*proof*⟩

lemma *compute-bound-tm-mono*:
fixes *I* :: *float interval list* **and** *f* :: *real list* ⇒ *real*
assumes *num-params* (*tm-poly t*) ≤ *length I*
assumes *I all-subset J*
assumes *a all-subset I*
shows *set-of* (*compute-bound-tm prec I a t*) ⊆ *set-of* (*compute-bound-tm prec J a t*)
⟨*proof*⟩

5.4 Computing taylor models for basic, univariate functions

definition *tm-const* :: *float* ⇒ *taylor-model*
where *tm-const c* = *TaylorModel* (*poly.C c*) *0*

context includes *floatarith-syntax* **begin**

definition *tm-pi* :: *nat* ⇒ *taylor-model*
where *tm-pi prec* = (
let pi-ivl = *the* (*compute-bound-fa prec Pi* [])
in TaylorModel (*poly.C* (*mid pi-ivl*)) (*centered pi-ivl*)
)

lemma *zero-real-interval*[*intro*,*simp*]: *0* ∈$_r$ *0*
⟨*proof*⟩

lemma *range-TM-tm-const*[*simp*]: *range-tm e* (*tm-const c*) = *interval-of c*
⟨*proof*⟩

lemma *num-params-tm-const*[*simp*]: *num-params* (*tm-poly* (*tm-const c*)) = *0*
⟨*proof*⟩

lemma *num-params-tm-pi*[*simp*]: *num-params* (*tm-poly* (*tm-pi prec*)) = *0*
⟨*proof*⟩

lemma *range-tm-tm-pi*: *pi* ∈$_i$ *range-tm e* (*tm-pi prec*)
⟨*proof*⟩

5.4.1 Derivations of floatarith expressions

Compute the nth derivative of a floatarith expression

fun *deriv* :: *nat* ⇒ *floatarith* ⇒ *nat* ⇒ *floatarith*
where *deriv v f 0* = *f*
| *deriv v f* (*Suc n*) = *DERIV-floatarith v* (*deriv v f n*)

lemma *isDERIV-DERIV-floatarith*:
assumes *isDERIV v f vs*
shows *isDERIV v* (*DERIV-floatarith v f*) *vs*
⟨*proof*⟩

lemma *isDERIV-is-analytic*:
isDERIV i (*Taylor-Models.deriv i f n*) *xs*
if *isDERIV i f xs*
⟨*proof*⟩

lemma *deriv-correct*:
assumes *isDERIV i f* (*xs*[*i*:=*t*]) *i* < *length xs*
shows ((λ*x*. *interpret-floatarith* (*deriv i f n*) (*xs*[*i*:=*x*])) *has-real-derivative interpret-floatarith* (*deriv i f* (*Suc n*)) (*xs*[*i*:=*t*]))
(*at t within S*)
⟨*proof*⟩

Faster derivation for univariate functions, producing smaller terms and thus less over-approximation.

TODO: Extend to Arctan, Log!

fun *deriv-rec* :: *floatarith* ⇒ *nat* ⇒ *floatarith*
where *deriv-rec* (*Exp* (*Var 0*)) - = *Exp* (*Var 0*)
| *deriv-rec* (*Cos* (*Var 0*)) *n* = (*case n mod 4*
of 0 ⇒ *Cos* (*Var 0*)
| *Suc 0* ⇒ *Minus* (*Sin* (*Var 0*))
| *Suc* (*Suc 0*) ⇒ *Minus* (*Cos* (*Var 0*))
| *Suc* (*Suc* (*Suc 0*)) ⇒ *Sin* (*Var 0*))
| *deriv-rec* (*Inverse* (*Var 0*)) *n* = (*if n* = *0 then Inverse* (*Var 0*) *else Mult* (*Num* (*fact n* ∗ (*if n mod 2* = *0 then 1 else* −*1*))) (*Inverse* (*Power* (*Var 0*) (*Suc n*))))
| *deriv-rec f n* = *deriv 0 f n*

lemma *deriv-rec-correct*:

assumes *isDERIV 0 f* (*xs*[*0*:=*t*]) *0* < *length xs*
shows ((λ*x*. *interpret-floatarith* (*deriv-rec f n*) (*xs*[*0*:=*x*])) *has-real-derivative interpret-floatarith* (*deriv-rec f* (*Suc n*)) (*xs*[*0*:=*t*])) (*at t within S*)
⟨*proof*⟩

lemma *deriv-rec-0-idem*[*simp*]:
shows *deriv-rec f 0* = *f*
⟨*proof*⟩

5.4.2 Computing Taylor models for arbitrary univariate expressions

fun *tmf-c* :: *nat* ⇒ *float interval list* ⇒ *floatarith* ⇒ *nat* ⇒ *float interval option*
where *tmf-c prec I f i* = *compute-bound-fa prec* (*Mult* (*deriv-rec f i*) (*Inverse* (*Num* (*fact i*)))) *I*
— The interval coefficients of the Taylor polynomial, i.e. the real coefficients approximated by a float interval.

fun *tmf-ivl-cs* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float list* ⇒ *floatarith* ⇒ *float interval list option*
where *tmf-ivl-cs prec ord I a f* = *those* (*map* (*tmf-c prec a f*) [*0*..<*ord*] @ [*tmf-c prec I f ord*])
— Make a list of bounds on the n+1 coefficients, with the n+1-th coefficient bounding the remainder term of the Taylor-Lagrange formula.

fun *tmf-polys* :: *float interval list* ⇒ *float poly* × *float interval poly*
where *tmf-polys* [] = (*poly.C 0*, *poly.C 0*)
| *tmf-polys* (*c* # *cs*) = (
let (*pf*, *pi*) = *tmf-polys cs*
in (*poly.CN* (*poly.C* (*mid c*)) *0 pf*, *poly.CN* (*poly.C* (*centered c*)) *0 pi*)
)

fun *tm-floatarith* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float list* ⇒ *floatarith* ⇒ *taylor-model option*
where *tm-floatarith prec ord I a f* = (
map-option (λ*cs*.
let (*pf*, *pi*) = *tmf-polys cs*;
- = *compute-bound-tm prec* (*List.map2* (−) *I a*);
e = *round-interval prec* (*Ipoly* (*List.map2* (−) *I a*) *pi*) — TODO: use *compute-bound-tm* here?!
in TaylorModel pf e
) (*tmf-ivl-cs prec ord I a f*)
) — Compute a Taylor model from an arbitrary, univariate floatarith expression, if possible. This is used to compute Taylor models for elemental functions like sin, cos, exp, etc.

term *compute-bound-poly*
lemma *tmf-c-correct*:
fixes *A*::*float interval list* **and** *I*::*float interval* **and** *f*::*floatarith* **and** *a*::*real list*

assumes *a all-in A*
assumes *tmf-c prec A f i = Some I*
shows *interpret-floatarith (deriv-rec f i) a / fact i* $\in_r$ *I*
⟨*proof*⟩

lemma *tmf-ivl-cs-length*:
assumes *tmf-ivl-cs prec n A a f = Some cs*
shows *length cs = n + 1*
⟨*proof*⟩

lemma *tmf-ivl-cs-correct*:
fixes *A*::*float interval list* **and** *f*::*floatarith*
assumes *a all-in I*
assumes *tmf-ivl-cs prec ord I a f = Some cs*
shows $\bigwedge i.$ *i < ord* $\Longrightarrow$ *tmf-c prec (map interval-of a) f i = Some (cs!i)*
and *tmf-c prec I f ord = Some (cs!ord)*
and *length cs = Suc ord*
⟨*proof*⟩

lemma *Ipoly-fst-tmf-polys*:
Ipoly xs (fst (tmf-polys z)) = ($\sum$ *i<length z. xs ! 0* ^ *i * (mid (z ! i)))*
for *xs*::*real list*
⟨*proof*⟩

lemma *insertion-fst-tmf-polys*:
insertion e (fst (tmf-polys z)) = ($\sum$ *i<length z. e 0* ^ *i * (mid (z ! i)))*
for *e*::*nat* $\Rightarrow$ *real*
⟨*proof*⟩

lemma *Ipoly-snd-tmf-polys*:
set-of (horner-eval (real-interval o centered o nth z) x (length z)) $\subseteq$ *set-of (Ipoly [x] (map-poly real-interval (snd (tmf-polys z))))*
⟨*proof*⟩

lemma *zero-interval*[*intro*,*simp*]: *0* $\in_i$ *0*
⟨*proof*⟩

lemma *sum-in-intervalI*: *sum f X* $\in_i$ *sum g X* **if** $\bigwedge x.$ *x* $\in$ *X* $\Longrightarrow$ *f x* $\in_i$ *g x*
for *f* :: - $\Rightarrow$ *'a* :: *ordered-comm-monoid-add*
⟨*proof*⟩

lemma *set-of-sum-subset*: *set-of (sum f X)* $\subseteq$ *set-of (sum g X)*
if $\bigwedge x.$ *x* $\in$ *X* $\Longrightarrow$ *set-of (f x)* $\subseteq$ *set-of (g x)*
for *f* :: -$\Rightarrow$*'a*::*linordered-ab-group-add interval*
⟨*proof*⟩

lemma *interval-of-plus*: *interval-of (a + b) = interval-of a + interval-of b*
⟨*proof*⟩

lemma *interval-of-uminus*: *interval-of* $(-\ a) = -$ *interval-of a*
⟨*proof*⟩

lemma *interval-of-zero*: *interval-of 0 = 0*
⟨*proof*⟩

lemma *interval-of-sum*: *interval-of* $(sum\ f\ X) = sum\ (\lambda x.\ interval\text{-}of\ (f\ x))\ X$
⟨*proof*⟩

lemma *interval-of-prod*: *interval-of* $(a * b) =$ *interval-of a* $*$ *interval-of b*
⟨*proof*⟩

lemma *in-set-of-interval-of*[*simp*]: $x \in_i$ (*interval-of y*) $\longleftrightarrow x = y$ **for** *x y*::'*a*::*order*
⟨*proof*⟩

lemma *real-interval-Ipoly*: *real-interval* (*Ipoly xs p*) = *Ipoly* (*map real-interval xs*) (*map-poly real-interval p*)
if *num-params p* ≤ *length xs*
⟨*proof*⟩

lemma *num-params-tmf-polys1*: *num-params* (*fst* (*tmf-polys z*)) ≤ *Suc 0*
⟨*proof*⟩

lemma *num-params-tmf-polys2*: *num-params* (*snd* (*tmf-polys z*)) ≤ *Suc 0*
⟨*proof*⟩

lemma *set-of-real-interval-subset*: *set-of* (*real-interval x*) ⊆ *set-of* (*real-interval y*)
if *set-of x* ⊆ *set-of y*
⟨*proof*⟩

theorem *tm-floatarith*:
assumes *t*: *tm-floatarith prec ord I xs f = Some t*
assumes *a*: *xs all-in I* **and** *x*: *x* ∈$_r$ *I* ! *0*
assumes *xs-ne*: *xs* ≠ []
assumes *deriv*: ⋀*x*. *x* ∈$_r$ *I* ! *0* ⟹ *isDERIV 0 f* (*xs*[*0* := *x*])
assumes ⋀*i*. *0* < *i* ⟹ *i* < *length xs* ⟹ *e i* = *real-of-float* (*xs* ! *i*)
assumes *diff-e*: (*x* − *real-of-float* (*xs* ! *0*)) = *e 0*
shows *interpret-floatarith f* (*xs*[*0*:=*x*]) ∈$_i$ *range-tm e t*
⟨*proof*⟩

5.5 Operations on Taylor models

fun *tm-norm-poly* :: *taylor-model* ⇒ *taylor-model*
where *tm-norm-poly* (*TaylorModel p e*) = *TaylorModel* (*polynate p*) *e*
— Normalizes the Taylor model by transforming its polynomial into horner form.

fun *tm-lower-order tm-lower-order-of-normed* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model*
where *tm-lower-order prec ord I a t* = *tm-lower-order-of-normed prec ord I a*

(*tm-norm-poly t*)
  | *tm-lower-order-of-normed prec ord I a* (*TaylorModel p e*) = (
        let (*l*, *r*) = *split-by-degree ord p*
        in TaylorModel l (*round-interval prec* (*e* + *compute-bound-poly prec r I a*))
)
— Reduces the degree of a Taylor model's polynomial to n and keeps it range by increasing the error bound.

fun *tm-round-floats tm-round-floats-of-normed* :: *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model*
 where *tm-round-floats prec I a t* = *tm-round-floats-of-normed prec I a* (*tm-norm-poly t*)
  | *tm-round-floats-of-normed prec I a* (*TaylorModel p e*) = (
        let (*l*, *r*) = *split-by-prec prec p*
        in TaylorModel l (*round-interval prec* (*e* + *compute-bound-poly prec r I a*))
)
— Rounding of Taylor models. Rounds both the coefficients of the polynomial and the floats in the error bound.

fun *tm-norm tm-norm′* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model*
  where *tm-norm prec ord I a t* = *tm-norm′ prec ord I a* (*tm-norm-poly t*)
  | *tm-norm′ prec ord I a t* = *tm-round-floats-of-normed prec I a* (*tm-lower-order-of-normed prec ord I a t*)
— Normalization of taylor models. Performs order lowering and rounding on tayor models, also converts the polynomial into horner form.

fun *tm-neg* :: *taylor-model* ⇒ *taylor-model*
  where *tm-neg* (*TaylorModel p e*) = *TaylorModel* ($\sim_p$ *p*) (−*e*)

fun *tm-add* :: *taylor-model* ⇒ *taylor-model* ⇒ *taylor-model*
  where *tm-add* (*TaylorModel p1 e1*) (*TaylorModel p2 e2*) = *TaylorModel* (*p1* $+_p$ *p2*) (*e1* + *e2*)

fun *tm-sub* :: *taylor-model* ⇒ *taylor-model* ⇒ *taylor-model*
  where *tm-sub t1 t2* = *tm-add t1* (*tm-neg t2*)

fun *tm-mul* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model* ⇒ *taylor-model*
  where *tm-mul prec ord I a* (*TaylorModel p1 e1*) (*TaylorModel p2 e2*) = (
        let d1 = *compute-bound-poly prec p1 I a*;
            d2 = *compute-bound-poly prec p2 I a*;
            p = *p1* $*_p$ *p2*;
            e = *e1*∗*d2* + *d1*∗*e2* + *e1*∗*e2*
        in tm-norm′ prec ord I a (*TaylorModel p e*)
)
lemmas [*simp del*] = *tm-norm′.simps*

fun *tm-pow* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model*

```
⇒ nat ⇒ taylor-model
  where tm-pow prec ord I a t 0 = tm-const 1
  | tm-pow prec ord I a t (Suc n) = (
        if odd (Suc n)
        then tm-mul prec ord I a t (tm-pow prec ord I a t n)
        else let t' = tm-pow prec ord I a t ((Suc n) div 2)
             in tm-mul prec ord I a t' t'
      )
```

Evaluates a float polynomial, using a Taylor model as the parameter. This is used to compose Taylor models.

```
fun eval-poly-at-tm :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ float
poly ⇒ taylor-model ⇒ taylor-model
  where eval-poly-at-tm prec ord I a (poly.C c) t = tm-const c
  | eval-poly-at-tm prec ord I a (poly.Bound n) t = t
  | eval-poly-at-tm prec ord I a (poly.Add p1 p2) t
        = tm-add (eval-poly-at-tm prec ord I a p1 t)
                 (eval-poly-at-tm prec ord I a p2 t)
  | eval-poly-at-tm prec ord I a (poly.Sub p1 p2) t
        = tm-sub (eval-poly-at-tm prec ord I a p1 t)
                 (eval-poly-at-tm prec ord I a p2 t)
  | eval-poly-at-tm prec ord I a (poly.Mul p1 p2) t
        = tm-mul prec ord I a (eval-poly-at-tm prec ord I a  p1 t)
                              (eval-poly-at-tm prec ord I a p2 t)
  | eval-poly-at-tm prec ord I a (poly.Neg p) t
        = tm-neg (eval-poly-at-tm prec ord I a p t)
  | eval-poly-at-tm prec ord I a (poly.Pw p n) t
        = tm-pow prec ord I a (eval-poly-at-tm prec ord I a p t) n
  | eval-poly-at-tm prec ord I a (poly.CN c n p) t = (
        let pt = eval-poly-at-tm prec ord I a p t;
            t-mul-pt = tm-mul prec ord I a t pt
        in tm-add (eval-poly-at-tm prec ord I a c t) t-mul-pt
      )

fun tm-inc-err :: float interval ⇒ taylor-model ⇒ taylor-model
  where tm-inc-err i (TaylorModel p e) = TaylorModel p (e + i)

fun tm-comp :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ float ⇒
taylor-model ⇒ taylor-model ⇒ taylor-model
  where tm-comp prec ord I a ta (TaylorModel p e) t = (
        let t-sub-ta = tm-sub t (tm-const ta);
            pt = eval-poly-at-tm prec ord I a p t-sub-ta
        in tm-inc-err e pt
      )
```

tm-max, *tm-min* and *tm-abs* are implemented extremely naively, because I don't expect them to be very useful. But the implementation is fairly modular, i.e. *tm-*{*abs*,*min*,*max*} all can easily be swapped out, as long as the corresponding correctness lemmas *tm-*{*abs*,*min*,*max*}*-range* are updated

as well.

fun *tm-abs* :: *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model*
where *tm-abs prec I a t* = (
let bound = *compute-bound-tm prec I a t*; *abs-bound*=*Ivl* (*0*::*float*) (*max* (*abs* (*lower bound*)) (*abs* (*upper bound*)))
in TaylorModel (*poly.C* (*mid abs-bound*)) (*centered abs-bound*))

fun *tm-union* :: *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model* ⇒ *taylor-model*
where *tm-union prec I a t1 t2* = (
let b1 = *compute-bound-tm prec I a t1*; *b2* = *compute-bound-tm prec I a t2*;
b-combined = *sup b1 b2*
in TaylorModel (*poly.C* (*mid b-combined*)) (*centered b-combined*))

fun *tm-min* :: *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model* ⇒ *taylor-model*
where *tm-min prec I a t1 t2* = *tm-union prec I a t1 t2*

fun *tm-max* :: *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *taylor-model* ⇒ *taylor-model*
where *tm-max prec I a t1 t2* = *tm-union prec I a t1 t2*

Rangeity of is preserved by our operations on Taylor models.

lemma *insertion-polyadd*[*simp*]: *insertion e* (*a* +$_p$ *b*) = *insertion e a* + *insertion e b*
for *a b*::′*a*::*ring-1 poly*
⟨*proof*⟩

lemma *insertion-polyneg*[*simp*]: *insertion e* (~$_p$ *b*) = − *insertion e b*
for *b*::′*a*::*ring-1 poly*
⟨*proof*⟩

lemma *insertion-polysub*[*simp*]: *insertion e* (*a* −$_p$ *b*) = *insertion e a* − *insertion e b*
for *a b*::′*a*::*ring-1 poly*
⟨*proof*⟩

lemma *insertion-polymul*[*simp*]: *insertion e* (*a* ∗$_p$ *b*) = *insertion e a* ∗ *insertion e b*
for *a b*::′*a*::*comm-ring-1 poly*
⟨*proof*⟩

lemma *insertion-polypow*[*simp*]: *insertion e* (*a* ^$_p$ *b*) = *insertion e a* ^ *b*
for *a*::′*a*::*comm-ring-1 poly*
⟨*proof*⟩

lemma *insertion-polynate* [*simp*]:

insertion bs (polynate p) = (insertion bs p :: 'a::comm-ring-1)
⟨*proof*⟩

lemma *tm-norm-poly-range*:
assumes *x* ∈$_i$ *range-tm e t*
shows *x* ∈$_i$ *range-tm e (tm-norm-poly t)*
⟨*proof*⟩

lemma *split-by-degree-correct-insertion*:
fixes *x* :: *nat* ⇒ *real* **and** *p* :: *float poly*
assumes *split-by-degree ord p* = (*l*, *r*)
shows *maxdegree l* ≤ *ord* (**is** *?P1*)
and *insertion x p* = *insertion x l* + *insertion x r* (**is** *?P2*)
and *num-params l* ≤ *num-params p* (**is** *?P3*)
and *num-params r* ≤ *num-params p* (**is** *?P4*)
⟨*proof*⟩

lemma *split-by-prec-correct-insertion*:
fixes *x* :: *nat* ⇒ *real* **and** *p* :: *float poly*
assumes *split-by-prec ord p* = (*l*, *r*)
shows *insertion x p* = *insertion x l* + *insertion x r* (**is** *?P1*)
and *num-params l* ≤ *num-params p* (**is** *?P2*)
and *num-params r* ≤ *num-params p* (**is** *?P3*)
⟨*proof*⟩

lemma *tm-lower-order-of-normed-range*:
assumes *x* ∈$_i$ *range-tm e t*
assumes *dev*: *develops-at-within e a I*
assumes *num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-lower-order-of-normed prec ord I a t*)
⟨*proof*⟩

lemma *num-params-tm-norm-poly-le*: *num-params* (*tm-poly* (*tm-norm-poly t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
⟨*proof*⟩

lemma *tm-lower-order-range*:
assumes *x* ∈$_i$ *range-tm e t*
assumes *dev*: *develops-at-within e a I*
assumes *num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-lower-order prec ord I a t*)
⟨*proof*⟩

lemma *tm-round-floats-of-normed-range*:
assumes *x* ∈$_i$ *range-tm e t*
assumes *dev*: *develops-at-within e a I*
assumes *num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-round-floats-of-normed prec I a t*)

— TODO: this is a clone of ⟦*?x* ∈$_i$ *range-tm ?e ?t*; *develops-at-within ?e ?a ?I*; *num-params* (*tm-poly ?t*) ≤ *length ?I*⟧ ⟹ *?x* ∈$_i$ *range-tm ?e* (*tm-lower-order-of-normed ?prec ?ord ?I ?a ?t*) -> general sweeping method!
⟨*proof*⟩

lemma *num-params-split-by-degree-le*: *num-params* (*fst* (*split-by-degree ord x*)) ≤ *K*
num-params (*snd* (*split-by-degree ord x*)) ≤ *K*
if *num-params x* ≤ *K* **for** *x*::*float poly*
⟨*proof*⟩

lemma *num-params-split-by-prec-le*: *num-params* (*fst* (*split-by-prec ord x*)) ≤ *K*
num-params (*snd* (*split-by-prec ord x*)) ≤ *K*
if *num-params x* ≤ *K* **for** *x*::*float poly*
⟨*proof*⟩

lemma *num-params-tm-norm′-le*:
num-params (*tm-poly* (*tm-round-floats-of-normed prec I a t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
⟨*proof*⟩

lemma *tm-round-floats-range*:
assumes *x* ∈$_i$ *range-tm e t develops-at-within e a I num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-round-floats prec I a t*)
⟨*proof*⟩

lemma *num-params-tm-lower-order-of-normed-le*: *num-params* (*tm-poly* (*tm-lower-order-of-normed prec ord I a t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
⟨*proof*⟩

lemma *tm-norm′-range*:
assumes *x* ∈$_i$ *range-tm e t develops-at-within e a I num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-norm′ prec ord I a t*)
⟨*proof*⟩

lemma *num-params-tm-norm′*:
num-params (*tm-poly* (*tm-norm′ prec ord I a t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
⟨*proof*⟩

lemma *tm-norm-range*:
assumes *x* ∈$_i$ *range-tm e t develops-at-within e a I num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-norm prec ord I a t*)
⟨*proof*⟩

lemmas [*simp del*] = *tm-norm.simps*

lemma *tm-neg-range*:
assumes $x \in_i$ *range-tm e t*
shows $-x \in_i$ *range-tm e* (*tm-neg t*)
⟨*proof*⟩
lemmas [*simp del*] = *tm-neg.simps*

lemma *tm-bound-tm-add*[*simp*]: *tm-bound* (*tm-add t1 t2*) = *tm-bound t1* + *tm-bound t2*
⟨*proof*⟩

lemma *interval-of-add*: *interval-of* (*a* + *b*) = *interval-of a* + *interval-of b*
⟨*proof*⟩

lemma *tm-add-range*:
$x + y \in_i$ *range-tm e* (*tm-add t1 t2*)
if $x \in_i$ *range-tm e t1*
$y \in_i$ *range-tm e t2*
⟨*proof*⟩
lemmas [*simp del*] = *tm-add.simps*

lemma *tm-sub-range*:
assumes $x \in_i$ *range-tm e t1*
assumes $y \in_i$ *range-tm e t2*
shows $x - y \in_i$ *range-tm e* (*tm-sub t1 t2*)
⟨*proof*⟩
lemmas [*simp del*] = *tm-sub.simps*

lemma *set-of-intervalI*: *set-of* (*interval-of y*) ⊆ *set-of Y* **if** $y \in_i$ *Y* **for** *y*::*'a*::*order*
⟨*proof*⟩

lemma *set-of-real-intervalI*: *set-of* (*interval-of y*) ⊆ *set-of* (*real-interval Y*) **if** $y \in_r$ *Y*
⟨*proof*⟩

lemma *tm-mul-range*:
assumes $x \in_i$ *range-tm e t1*
assumes $y \in_i$ *range-tm e t2*
assumes *dev*: *develops-at-within e a I*
assumes *params*: *num-params* (*tm-poly t1*) ≤ *length I num-params* (*tm-poly t2*) ≤ *length I*
shows $x * y \in_i$ *range-tm e* (*tm-mul prec ord I a t1 t2*)
⟨*proof*⟩

lemma *num-params-tm-mul-le*:
num-params (*tm-poly* (*tm-mul prec ord I a t1 t2*)) ≤ *X*
if *num-params* (*tm-poly t1*) ≤ *X*

num-params (*tm-poly t2*) ≤ *X*
⟨*proof*⟩

lemmas [*simp del*] = *tm-pow.simps*— TODO: make a systematic decision

lemma
shows *tm-pow-range*: *num-params* (*tm-poly t*) ≤ *length I* ⟹
develops-at-within e a I ⟹
x ∈$_i$ *range-tm e t* ⟹
x ^ *n* ∈$_i$ *range-tm e* (*tm-pow prec ord I a t n*)
and *num-params-tm-pow-le*[*THEN order-trans*]:
num-params (*tm-poly* (*tm-pow prec ord I a t n*)) ≤ *num-params* (*tm-poly t*)
⟨*proof*⟩

lemma *num-params-tm-add-le*:
num-params (*tm-poly* (*tm-add t1 t2*)) ≤ *X*
if *num-params* (*tm-poly t1*) ≤ *X*
num-params (*tm-poly t2*) ≤ *X*
⟨*proof*⟩

lemma *num-params-tm-neg-eq*[*simp*]:
num-params (*tm-poly* (*tm-neg t1*)) = *num-params* (*tm-poly t1*)
⟨*proof*⟩

lemma *num-params-tm-sub-le*:
num-params (*tm-poly* (*tm-sub t1 t2*)) ≤ *X*
if *num-params* (*tm-poly t1*) ≤ *X*
num-params (*tm-poly t2*) ≤ *X*
⟨*proof*⟩

lemma *num-params-eval-poly-le*: *num-params* (*tm-poly* (*eval-poly-at-tm prec ord I a p t*)) ≤ *x*
if *num-params* (*tm-poly t*) ≤ *x num-params p* ≤ *max 1 x*
⟨*proof*⟩

lemma *eval-poly-at-tm-range*:
assumes *num-params p* ≤ *1*
assumes *tg-def*: *e′ 0* ∈$_i$ *range-tm e tg*
assumes *dev*: *develops-at-within e a I* **and** *params*: *num-params* (*tm-poly tg*) ≤ *length I*
shows *insertion e′ p* ∈$_i$ *range-tm e* (*eval-poly-at-tm prec ord I a p tg*)
⟨*proof*⟩

lemma *tm-inc-err-range*: *x* ∈$_i$ *range-tm e* (*tm-inc-err i t*)
if *x* ∈$_i$ *range-tm e t* + *real-interval i*
⟨*proof*⟩

lemma *num-params-tm-inc-err*: *num-params* (*tm-poly* (*tm-inc-err i t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*

⟨*proof*⟩

lemma *num-params-tm-comp-le*: *num-params* (*tm-poly* (*tm-comp prec ord I a ga tf tg*)) ≤ *X*
if *num-params* (*tm-poly tf*) ≤ *max 1 X num-params* (*tm-poly tg*) ≤ *X*
⟨*proof*⟩

lemma *tm-comp-range*:
assumes *tf-def*: *x* ∈$_i$ *range-tm e′ tf*
assumes *tg-def*: *e′ 0* ∈$_i$ *range-tm e* (*tm-sub tg* (*tm-const ga*))
assumes *params*: *num-params* (*tm-poly tf*) ≤ *1 num-params* (*tm-poly tg*) ≤ *length I*
assumes *dev*: *develops-at-within e a I*
shows *x* ∈$_i$ *range-tm e* (*tm-comp prec ord I a ga tf tg*)
⟨*proof*⟩

lemma *mid-centered-collapse*:
interval-of (*real-of-float* (*mid abs-bound*)) + *real-interval* (*centered abs-bound*) = *real-interval abs-bound*
⟨*proof*⟩

lemmas [*simp del*] = *tm-abs.simps*
lemma *tm-abs-range*:
assumes *x*: *x* ∈$_i$ *range-tm e t*
assumes *n*: *num-params* (*tm-poly t*) ≤ *length I* **and** *d*: *develops-at-within e a I*
shows *abs x* ∈$_i$ *range-tm e* (*tm-abs prec I a t*)
⟨*proof*⟩

lemma *num-params-tm-abs-le*: *num-params* (*tm-poly* (*tm-abs prec I a t*)) ≤ *X* **if** *num-params* (*tm-poly t*) ≤ *X*
⟨*proof*⟩

lemma *real-interval-sup*: *real-interval* (*sup a b*) = *sup* (*real-interval a*) (*real-interval b*)
⟨*proof*⟩

lemma *in-interval-supI1*: *x* ∈$_i$ *a* ⟹ *x* ∈$_i$ *sup a b*
and *in-interval-supI2*: *x* ∈$_i$ *b* ⟹ *x* ∈$_i$ *sup a b*
for *x*::*'a*::*lattice*
⟨*proof*⟩

lemma *tm-union-range-left*:
assumes *x* ∈$_i$ *range-tm e t1*
num-params (*tm-poly t1*) ≤ *length I develops-at-within e a I*
shows *x* ∈$_i$ *range-tm e* (*tm-union prec I a t1 t2*)
⟨*proof*⟩

lemma *tm-union-range-right*:
assumes *x* ∈$_i$ *range-tm e t2*

num-params (*tm-poly t2*) ≤ *length I develops-at-within e a I*
shows *x* ∈$_i$ *range-tm e* (*tm-union prec I a t1 t2*)
⟨*proof*⟩

lemma *num-params-tm-union-le*:
num-params (*tm-poly* (*tm-union prec I a t1 t2*)) ≤ *X*
if *num-params* (*tm-poly t1*) ≤ *X num-params* (*tm-poly t2*) ≤ *X*
⟨*proof*⟩

lemmas [*simp del*] = *tm-union.simps tm-min.simps tm-max.simps*

lemma *tm-min-range*:
assumes *x* ∈$_i$ *range-tm e t1*
assumes *y* ∈$_i$ *range-tm e t2*
num-params (*tm-poly t1*) ≤ *length I*
num-params (*tm-poly t2*) ≤ *length I*
develops-at-within e a I
shows *min x y* ∈$_i$ *range-tm e* (*tm-min prec I a t1 t2*)
⟨*proof*⟩

lemma *tm-max-range*:
assumes *x* ∈$_i$ *range-tm e t1*
assumes *y* ∈$_i$ *range-tm e t2*
num-params (*tm-poly t1*) ≤ *length I*
num-params (*tm-poly t2*) ≤ *length I*
develops-at-within e a I
shows *max x y* ∈$_i$ *range-tm e* (*tm-max prec I a t1 t2*)
⟨*proof*⟩

5.6 Computing Taylor models for multivariate expressions

Compute Taylor models for expressions of the form "f (g x)", where f is an elementary function like exp or cos, by composing Taylor models for f and g. For our correctness proof, we need to make it explicit that the range of g on I is inside the domain of f, by introducing the *f-exists-on* predicate.

fun *compute-tm-by-comp* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *floatarith* ⇒ *taylor-model option* ⇒ (*float interval* ⇒ *bool*) ⇒ *taylor-model option*
where *compute-tm-by-comp prec ord I a f g f-exists-on* = (
case g
of Some tg ⇒ (
let gI = *compute-bound-tm prec I a tg*;
ga = *mid* (*compute-bound-tm prec a a tg*)
in if f-exists-on gI
then map-option (λ*tf*. *tm-comp prec ord I a ga tf tg*) (*tm-floatarith prec ord* [*gI*] [*ga*] *f*)
else None)
| - ⇒ *None*
)

Compute Taylor models with numerical precision *prec* of degree *ord*, with Taylor models in the environment *env* whose variables are jointly interpreted with domain *I* and expanded around point *a*. from floatarith expressions on a rectangular domain.

fun *approx-tm* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *floatarith* ⇒ *taylor-model list* ⇒
taylor-model option
where *approx-tm - - I - (Num c) env = Some (tm-const c)*
| *approx-tm - - I a (Var n) env = (if n < length env then Some (env ! n) else None)*
| *approx-tm prec ord I a (Add l r) env = (*
case (approx-tm prec ord I a l env, approx-tm prec ord I a r env)
of (Some t1, Some t2) ⇒ Some (tm-add t1 t2)
| - ⇒ None)
| *approx-tm prec ord I a (Minus f) env*
= map-option tm-neg (approx-tm prec ord I a f env)
| *approx-tm prec ord I a (Mult l r) env = (*
case (approx-tm prec ord I a l env, approx-tm prec ord I a r env)
of (Some t1, Some t2) ⇒ Some (tm-mul prec ord I a t1 t2)
| - ⇒ None)
| *approx-tm prec ord I a (Power f k) env*
= map-option (λt. tm-pow prec ord I a t k)
(approx-tm prec ord I a f env)
| *approx-tm prec ord I a (Inverse f) env*
= compute-tm-by-comp prec ord I a (Inverse (Var 0)) (approx-tm prec ord I a f env) (λx. 0 < lower x ∨ upper x < 0)
| *approx-tm prec ord I a (Cos f) env*
= compute-tm-by-comp prec ord I a (Cos (Var 0)) (approx-tm prec ord I a f env) (λx. True)
| *approx-tm prec ord I a (Arctan f) env*
= compute-tm-by-comp prec ord I a (Arctan (Var 0)) (approx-tm prec ord I a f env) (λx. True)
| *approx-tm prec ord I a (Exp f) env*
= compute-tm-by-comp prec ord I a (Exp (Var 0)) (approx-tm prec ord I a f env) (λx. True)
| *approx-tm prec ord I a (Ln f) env*
= compute-tm-by-comp prec ord I a (Ln (Var 0)) (approx-tm prec ord I a f env) (λx. 0 < lower x)
| *approx-tm prec ord I a (Sqrt f) env*
= compute-tm-by-comp prec ord I a (Sqrt (Var 0)) (approx-tm prec ord I a f env) (λx. 0 < lower x)
| *approx-tm prec ord I a Pi env = Some (tm-pi prec)*
| *approx-tm prec ord I a (Abs f) env*
= map-option (tm-abs prec I a) (approx-tm prec ord I a f env)
| *approx-tm prec ord I a (Min l r) env = (*
case (approx-tm prec ord I a l env, approx-tm prec ord I a r env)
of (Some t1, Some t2) ⇒ Some (tm-min prec I a t1 t2)
| - ⇒ None)
| *approx-tm prec ord I a (Max l r) env = (*

case (*approx-tm prec ord I a l env*, *approx-tm prec ord I a r env*)
of (*Some t1*, *Some t2*) ⇒ *Some* (*tm-max prec I a t1 t2*)
| - ⇒ *None*)
| *approx-tm prec ord I a* (*Powr l r*) *env* = *None* — TODO
| *approx-tm prec ord I a* (*Floor l*) *env* = *None* — TODO

lemma *mid-in-real-interval*: *mid i* ∈$_r$ *i*
⟨*proof*⟩

lemma *set-of-real-interval-mono*:*set-of* (*real-interval x*) ⊆ *set-of* (*real-interval y*)
if *set-of x* ⊆ *set-of y*
⟨*proof*⟩

lemmas [*simp del*] = *compute-bound-poly.simps tm-floatarith.simps*

lemmas [*simp del*] = *tmf-ivl-cs.simps compute-bound-tm.simps tmf-polys.simps*

lemma *tm-floatarith-eq-Some-num-params*:
tm-floatarith prec ord a b f = *Some tf* ⟹ *num-params* (*tm-poly tf*) ≤ *1*
⟨*proof*⟩

lemma *compute-tm-by-comp-range*:
assumes *max-Var-floatarith f* ≤ *1*
assumes *a*: *a all-subset I*
assumes *tx-range*: *x* ∈$_i$ *range-tm e tg*
assumes *t-def*: *compute-tm-by-comp prec ord I a f* (*Some tg*) *c* = *Some t*
assumes *f-deriv*:
⋀*x*. *x* ∈$_r$ *compute-bound-tm prec I a tg* ⟹ *c* (*compute-bound-tm prec I a tg*)
⟹ *isDERIV 0 f* [*x*]
assumes *params*: *num-params* (*tm-poly tg*) ≤ *length I*
and *dev*: *develops-at-within e a I*
shows *interpret-floatarith f* [*x*] ∈$_i$ *range-tm e t*
⟨*proof*⟩

lemmas [*simp del*] = *compute-tm-by-comp.simps*

lemma *compute-tm-by-comp-num-params-le*:
assumes *compute-tm-by-comp prec ord I a f* (*Some t0*) *i* = *Some t*
assumes *1* ≤ *X num-params* (*tm-poly t0*) ≤ *X*
shows *num-params* (*tm-poly t*) ≤ *X*
⟨*proof*⟩

lemma *compute-tm-by-comp-eq-Some-iff*: *compute-tm-by-comp prec ord I a f t0 i*
= *Some t* ⟷
(∃ *z x2*. *t0* = *Some x2* ∧
tm-floatarith prec ord [*compute-bound-tm prec I a x2*]
[*mid* (*compute-bound-tm prec a a x2*)] *f* =

Some z
∧ *tm-comp prec ord I a*
(mid (compute-bound-tm prec a a x2)) z x2 = t
∧ *i (compute-bound-tm prec I a x2))*
⟨*proof*⟩

lemma *num-params-approx-tm*:
assumes *approx-tm prec ord I a f env = Some t*
assumes ⋀*tm. tm ∈ set env ⟹ num-params (tm-poly tm) ≤ length I*
shows *num-params (tm-poly t) ≤ length I*
⟨*proof*⟩

lemma *in-interval-realI*: *a ∈$_i$ I* **if** *a ∈$_r$ I* ⟨*proof*⟩

lemma *all-subset-all-inI*: *map interval-of a all-subset I* **if** *a all-in I*
⟨*proof*⟩

lemma *compute-tm-by-comp-None*: *compute-tm-by-comp p ord I a x None k = None*
⟨*proof*⟩

lemma *approx-tm-num-Vars-None*:
assumes *max-Var-floatarith f > length env*
shows *approx-tm p ord I a f env = None*
⟨*proof*⟩

lemma *approx-tm-num-Vars*:
assumes *approx-tm prec ord I a f env = Some t*
shows *max-Var-floatarith f ≤ length env*
⟨*proof*⟩

definition *range-tms e xs = map (range-tm e) xs*

lemma *approx-tm-range*:
assumes *a*: *a all-subset I*
assumes *t-def*: *approx-tm prec ord I a f env = Some t*
assumes *allin*: *xs all-in$_i$ range-tms e env*
assumes *devs*: *develops-at-within e a I*
assumes *env*: ⋀*tm. tm ∈ set env ⟹ num-params (tm-poly tm) ≤ length I*
shows *interpret-floatarith f xs ∈$_i$ range-tm e t*
⟨*proof*⟩

Evaluate expression with Taylor models in environment.

5.7 Computing bounds for floatarith expressions

TODO: compare parametrization of input vs. uncertainty for input...

definition *tm-of-ivl-par n ivl = TaylorModel (CN (C ((upper ivl + lower ivl)∗Float 1 (−1))) n*

(*C* ((*upper ivl* − *lower ivl*)∗*Float 1* (−*1*)))) *0*
— track uncertainty in parameter *n*, which is to be interpreted over standardized domain [−*1*, *1*].

value *tm-of-ivl-par 3* (*Ivl* (−*1*) *1*)

definition *tms-of-ivls ivls* = *map* (λ(*i*, *ivl*). *tm-of-ivl-par i ivl*) (*zip* [*0*..<*length ivls*] *ivls*)

value *tms-of-ivls* [*Ivl 1 2*, *Ivl 4 5*]

primrec *approx-slp'*::*nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *slp* ⇒ *taylor-model list* ⇒ *taylor-model list option*
where
approx-slp' p ord I a [] *xs* = *Some xs*
| *approx-slp' p ord I a* (*ea* # *eas*) *xs* =
do {
r ← *approx-tm p ord I a ea xs*;
approx-slp' p ord I a eas (*r*#*xs*)
}

lemma *mem-range-tms-Cons-iff*[*simp*]: *x*#*xs all-in*$_i$ *range-tms e* (*X*#*XS*) ⟷ *x* ∈$_i$ *range-tm e X* ∧ *xs all-in*$_i$ *range-tms e XS*
⟨*proof*⟩

lemma *approx-slp'-range*:
assumes *i*: *i all-subset I*
assumes *dev*: *develops-at-within e i I*
assumes *vs*: *vs all-in*$_i$ *range-tms e VS* (⋀*tm*. *tm* ∈ *set VS* ⟹ *num-params* (*tm-poly tm*) ≤ *length I*)
assumes *appr*: *approx-slp' p ord I i ra VS* = *Some X*
shows *interpret-slp ra vs all-in*$_i$ *range-tms e X*
⟨*proof*⟩

definition *approx-slp*::*nat* ⇒ *nat* ⇒ *nat* ⇒ *slp* ⇒ *taylor-model list* ⇒ *taylor-model list option*
where
approx-slp p ord d slp tms =
map-option (*take d*)
(*approx-slp' p ord* (*replicate* (*length tms*) (*Ivl* (−*1*) *1*)) (*replicate* (*length tms*) *0*) *slp tms*)

lemma *length-range-tms*[*simp*]: *length* (*range-tms e VS*) = *length VS*
⟨*proof*⟩

lemma *set-of-Ivl*: *set-of* (*Ivl a b*) = {*a* .. *b*} **if** *a* ≤ *b*
⟨*proof*⟩

lemma *set-of-zero*[*simp*]: *set-of 0* = {*0*::'*a*::*ordered-comm-monoid-add*}

⟨*proof*⟩

theorem *approx-slp-range-tms*:
assumes *approx-slp p ord d slp VS = Some X*
assumes *slp-def*: *slp = slp-of-fas fas*
assumes *d-def*: *d = length fas*
assumes *e*: *e ∈ UNIV → {−1 .. 1}*
assumes *vs*: *vs all-in$_i$ range-tms e VS*
assumes *lens*: *⋀tm. tm ∈ set VS ⟹ num-params (tm-poly tm) ≤ length vs*
shows *interpret-floatarith fas vs all-in$_i$ range-tms e X*
⟨*proof*⟩

end

end
theory *Experiments*
imports *Taylor-Models*
Affine-Arithmetic.Affine-Arithmetic
begin

instantiation *interval*::({*show*, *preorder*}) *show* **begin**

context includes *interval.lifting* **begin**
lift-definition *shows-prec-interval*::
nat ⇒ 'a interval ⇒ char list ⇒ char list
is *λp ivl s. (shows-string ''Interval'' o shows ivl) s* ⟨*proof*⟩

lift-definition *shows-list-interval*::
'a interval list ⇒ char list ⇒ char list
is *λivls s. shows-list ivls s* ⟨*proof*⟩

instance
⟨*proof*⟩
end

end

definition *split-largest-interval* :: *float interval list ⇒ float interval list × float interval list* **where**
split-largest-interval xs = (case sort-key (uminus o snd) (zip [0..<length xs] (map (λx. upper x − lower x) xs)) of Nil ⇒ ([], [])
| (i, -)#- ⇒ let x = xs! i in (xs[i:=Ivl (lower x) ((upper x + lower x)∗Float 1 (−1))],
xs[i:=Ivl ((upper x + lower x)∗Float 1 (−1)) (upper x)]))

definition *Inf-tm p params tm =*
lower (compute-bound-tm p (replicate params (Ivl (−1) (1))) (replicate params (Ivl 0 0)) tm)

primrec *prove-pos*::*bool* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒
(*nat* ⇒ *nat* ⇒ *taylor-model list* ⇒ *taylor-model option*) ⇒ *float interval list list* ⇒ *bool* **where**
prove-pos prnt 0 p ord F X = (*let* - = *if prnt then print* (*STR* ''# *depth limit exceeded*↩'') *else* () *in False*)
| *prove-pos prnt* (*Suc i*) *p ord F XXS* =
(*case XXS of* [] ⇒ *True* | (*X*#*XS*) ⇒
let
params = *length X*;
R = *F p ord* (*tms-of-ivls X*);
- = *if prnt then print* (*String.implode* ((*shows* ''# '' *o shows* (*map* (λ*ivl*. (*lower ivl*, *upper ivl*)) *X*)) ''↩'')) *else* ()
in
if R ≠ *None* ∧ 0 < *Inf-tm p params* (*the R*)
then let - = *if prnt then print* (*STR* ''# *Success*↩'') *else* () *in prove-pos prnt i p ord F XS*
else let - = *if prnt then print* (*String.implode* ((*shows* ''# *Split* ('' *o shows* ((*map-option* (*Inf-tm p params*)) *R*) *o shows* '')'') ''↩'')) *else* () *in case split-largest-interval X of* (*a*, *b*) ⇒
prove-pos prnt i p ord F (*a*#*b*#*XS*))

hide-const (**open**) *prove-pos-slp*

definition *prove-pos-slp prnt prec ord fa i xs* = (*let slp* = *slp-of-fas* [*fa*] *in prove-pos prnt i prec ord* (λ*p ord xs*.
case approx-slp prec ord 1 slp xs of None ⇒ *None* | *Some* [*x*] ⇒ *Some x* | *Some* - ⇒ *None*) *xs*)

experiment begin

unbundle *floatarith-syntax*

abbreviation *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 *prove-pos-slp True 30 0 schwefel 100000* [*replicate 3* (*Ivl* (−*10*) *10*)]
⟨*proof*⟩

abbreviation *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*)

lemma *prove-pos-slp True 30 3 delta6 10000* [*replicate 6* (*Ivl 4* (*Float 104045* (−*14*)))]
⟨*proof*⟩

abbreviation *caprasse* ≡ (*3.1801* + − *Var 0* ∗ (*Var 2*) ^*e* *3* + *4* ∗ *Var 1* ∗ (*Var 2*)^*e*2 ∗ *Var 3* +
4 ∗ *Var 0* ∗ *Var 2* ∗ (*Var 3*)^*e*2 + *2* ∗ *Var 1* ∗ (*Var 3*)^*e*3 + *4* ∗ *Var 0* ∗ *Var 2* + *4* ∗ (*Var 2*)^*e*2 − *10* ∗ *Var 1* ∗ *Var 3* +
−*10* ∗ (*Var 3*)^*e*2 + *2*)

lemma *prove-pos-slp True 30 2 caprasse 10000* [*replicate 4* (*Ivl* (−*Float 1* (−*1*)) (*Float 1* (−*1*)))]
⟨*proof*⟩

abbreviation *magnetism* ≡
0.25001 + (*Var 0*)^*e*2 + *2* ∗ (*Var 1*)^*e*2 + *2* ∗ (*Var 2*)^*e*2 + *2* ∗ (*Var 3*)^*e*2 + *2* ∗ (*Var 4*)^*e*2 + *2* ∗ (*Var 5*)^*e*2 +
2 ∗ (*Var 6*)^*e*2 − *Var 0*

end

end