

# Taylor Models

Christoph Traut and Fabian Immler

May 26, 2024

## 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

<b>1</b>	<b>Topology for Floating Point Numbers</b>	<b>1</b>
<b>2</b>	<b>Horner Evaluation</b>	<b>6</b>
<b>3</b>	<b>Splitting polynomials to reduce floating point precision</b>	<b>11</b>
<b>4</b>	<b>Splitting polynomials by degree</b>	<b>13</b>
<b>5</b>	<b>Multivariate Taylor Models</b>	<b>19</b>
5.1	Computing interval bounds on arithmetic expressions . . . . .	19
5.2	Definition of Taylor models and notion of rangeity . . . . .	19
5.3	Interval bounds for Taylor models . . . . .	20
5.4	Computing taylor models for basic, univariate functions . . .	25
5.4.1	Derivations of floatarith expressions . . . . .	26
5.4.2	Computing Taylor models for arbitrary univariate expressions . . . . .	29
5.5	Operations on Taylor models . . . . .	35
5.6	Computing Taylor models for multivariate expressions . . . .	51
5.7	Computing bounds for floatarith expressions . . . . .	62

# 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  $\Rightarrow$  float  $\Rightarrow$  real is dist .*

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

**by** *transfer simp*

**lemma** *dist-float-triangle2: dist x y  $\leq$  dist x z + dist y z for x y z::float*

**by** *transfer (rule dist-triangle2)*

**instance ..**

**end**

**instantiation** *float :: uniformity*

**begin**

**definition** *uniformity-float :: (float  $\times$  float) filter*

**where** *uniformity-float = (INF e $\in$ {0<..}. principal {(x, y). dist x y < e})*

**instance ..**

**end**

**lemma** *float-dense-in-real:*

**fixes** *x :: real*

**assumes** *x < y*

**shows**  $\exists r \in \text{float}. x < r \wedge r < y$

**proof** –

**from**  $\langle x < y \rangle$  **have**  $0 < y - x$  **by** *simp*

**with** *reals-Archimedean* **obtain** *q' :: nat* **where** *q': inverse (real q') < y - x*

**and**  $0 < q'$

**by** *blast*

**define** *q::nat* **where**  $q \equiv 2 \wedge \text{nat } |\text{bitlen } q'|$

**from** *bitlen-bounds[of q']*  $\langle 0 < q' \rangle$  **have**  $q' < q$

**by** *(auto simp: q-def)*

**then have** *inverse q < inverse q'*

**using**  $\langle 0 < q' \rangle$

```

    by (auto simp: divide-simps)
  with ‹ $q' < q$ ›  $q'$  have  $q: \text{inverse } (\text{real } q) < y - x$  and  $0 < q$ 
    by (auto simp: split: if-splits)
  define  $p$  where  $p = \lceil y * \text{real } q \rceil - 1$ 
  define  $r$  where  $r = \text{of-int } p / \text{real } q$ 
  from  $q$  have  $x < y - \text{inverse } (\text{real } q)$ 
    by simp
  also from ‹ $0 < q$ › have  $y - \text{inverse } (\text{real } q) \leq r$ 
    by (simp add: r-def p-def le-divide-eq left-diff-distrib)
  finally have  $x < r$  .
  moreover from ‹ $0 < q$ › have  $r < y$ 
    by (simp add: r-def p-def divide-less-eq diff-less-eq less-ceiling-iff [symmetric])
  moreover have  $r \in \text{float}$ 
    by (simp add: r-def q-def)
  ultimately show ?thesis by blast
qed

```

```

lemma real-of-float-dense:
  fixes  $x y :: \text{real}$ 
  assumes  $x < y$ 
  shows  $\exists q :: \text{float}. x < \text{real-of-float } q \wedge \text{real-of-float } q < y$ 
  using float-dense-in-real [OF ‹ $x < y$ ›]
  by (auto elim: real-of-float-cases)

```

```

instantiation float :: linorder-topology
begin

```

```

definition open-float::float set  $\Rightarrow$  bool where
  open-float  $S = (\forall x \in S. \exists e > 0. \forall y. \text{dist } y x < e \longrightarrow y \in S)$ 

```

```

instance
proof (standard, intro ext iffI)
  fix  $U :: \text{float set}$ 
  assume generate-topology (range lessThan  $\cup$  range greaterThan)  $U$ 
  then show open  $U$ 
    unfolding open-float-def uniformity-float-def
  proof (induction  $U$ )
    case UNIV
    then show ?case by (auto intro!: zero-less-one)
  next
    case (Int  $a b$ )
    show ?case
    proof safe
      fix  $x$  assume  $x \in a$   $x \in b$ 
      with Int(3,4) obtain  $e1 e2$ 
        where  $\text{dist } (y) (x) < e1 \implies y \in a$ 
          and  $\text{dist } (y) (x) < e2 \implies y \in b$ 
          and  $0 < e1$   $0 < e2$ 
      for  $y$ 

```

```

      by (auto dest!: bspec)
    then show  $\exists e > 0. \forall y. \text{dist } y \ x < e \longrightarrow y \in a \cap b$ 
      by (auto intro!: exI[where x=min e1 e2])
  qed
next
  case (UN K)
  show ?case
  proof safe
    fix x X assume x:  $x \in X$  and X:  $X \in K$ 
    from UN[OF X] x obtain e where
       $\text{dist } (y) \ (x) < e \implies y \in X \ e > 0$  for y
    by auto
    then show  $\exists e > 0. \forall y. \text{dist } (\text{real-of-float } y) \ (\text{real-of-float } x) < e \longrightarrow y \in \bigcup K$ 
      using x X
      by (auto intro!: exI[where x=e])
  qed
next
  case (Basis s)
  then show ?case
  proof safe
    fix x u::float
    assume  $x < u$ 
    then show  $\exists e > 0. \forall y. \text{dist } (\text{real-of-float } y) \ (\text{real-of-float } x) < e \longrightarrow y \in \{..<u\}$ 
      by (force simp add: eventually-principal dist-float-def
        dist-real-def abs-real-def
        intro!: exI[where x=(u - x)/2])
  next
    fix x l::float
    assume  $l < x$ 
    then show  $\exists e > 0. \forall y. \text{dist } (\text{real-of-float } y) \ (\text{real-of-float } x) < e \longrightarrow y \in \{l<..\}$ 
      by (force simp add: eventually-principal dist-float-def
        dist-real-def abs-real-def
        intro!: exI[where x=(x - l)/2])
  qed
qed
next
  fix U::float set
  assume open U
  then obtain e where e:
     $x \in U \implies e \ x > 0$ 
     $x \in U \implies \text{dist } (y) \ (x) < e \ x \implies y \in U$  for x y
  unfolding open-float-def uniformity-float-def
  by metis
  {
    fix x
    assume x:  $x \in U$ 
    obtain e' where e':  $e' > 0 \text{ real-of-float } e' < e \ x$ 

```

```

    using real-of-float-dense[of 0 e x]
    using e(1)[OF x]
    by auto
  then have dist (y) (x) < e'  $\implies$  y  $\in$  U for y
    by (intro e[OF x]) auto
  then have  $\exists e' > 0. \forall y. \text{dist } (y) (x) < \text{real-of-float } e' \longrightarrow y \in U$ 
    using e'
    by auto
} then
obtain e' where e':
  x  $\in$  U  $\implies$  0 < e' x
  x  $\in$  U  $\implies$  dist y x < real-of-float (e' x)  $\implies$  y  $\in$  U for x y
  by metis
then have U = ( $\bigcup_{x \in U}. \text{greaterThan } (x - e' x) \cap \text{lessThan } (x + e' x)$ )
  by (auto simp: dist-float-def dist-commute dist-real-def)
also have generate-topology (range lessThan  $\cup$  range greaterThan) ...
  by (intro generate-topology-Union generate-topology.Int generate-topology.Basis)
auto
finally show generate-topology (range lessThan  $\cup$  range greaterThan) U .
qed

end

instance float :: metric-space
proof standard
  fix U::float set
  show open U = ( $\forall x \in U. \forall_F (x', y) \text{ in uniformity. } x' = x \longrightarrow y \in U$ )
    unfolding open-float-def open-dist uniformity-float-def uniformity-real-def
  proof safe
    fix x
    assume  $\forall x \in U. \exists e > 0. \forall y. \text{dist } (\text{real-of-float } y) (\text{real-of-float } x) < e \longrightarrow y \in U$ 
    x  $\in$  U
    then obtain e where e > 0 dist (y) (x) < e  $\implies$  y  $\in$  U for y
      by auto
    then show  $\forall_F (x', y) \text{ in INF } e \in \{0 < ..\}. \text{principal } \{(x, y). \text{dist } x y < e\}. x' = x \longrightarrow y \in U$ 
      by (intro eventually-INF1[where i=e])
        (auto simp: eventually-principal dist-commute dist-float-def)
    next
      fix u
      assume  $\forall x \in U. \forall_F (x', y) \text{ in INF } e \in \{0 < ..\}. \text{principal } \{(x, y). \text{dist } x y < e\}. x' = x \longrightarrow y \in U$ 
      x' = x  $\longrightarrow$  y  $\in$  U
      u  $\in$  U
      from this obtain E where E: E  $\subseteq$  {0 < ..} finite E
         $\forall (x', y) \in \bigcap_{x \in E}. \{(y', y). \text{dist } y' y < x\}. x' = u \longrightarrow y \in U$ 
      by (subst (asm) eventually-INF) (auto simp: INF-principal-finite eventually-principal)
      then show  $\exists e > 0. \forall y. \text{dist } (\text{real-of-float } y) (\text{real-of-float } u) < e \longrightarrow y \in U$ 
        by (intro exI[where x=if E = {} then 1 else Min E])

```

```

      (auto simp: dist-commute dist-float-def)
    qed
  qed (use dist-float-eq-0-iff dist-float-triangle2 in
    ‹auto simp add: uniformity-float-def dist-float-def›)

instance float::topological-ab-group-add
proof
  fix a b::float
  show ((λx. fst x + snd x) ⟶ a + b) (nhds a ×F nhds b)
  proof (rule tendstoI)
    fix e::real
    assume e > 0
    have 1: (fst ⟶ a) (nhds a ×F nhds b)
    and 2: (snd ⟶ b) (nhds a ×F nhds b)
    by (auto intro!: tendsto-eq-intros filterlim-ident simp: nhds-prod[symmetric])
    have ∀F x in nhds a ×F nhds b. dist (fst x) (a) < e/2
    by (rule tendstoD[OF 1]) (use ‹e > 0› in auto)
    moreover have ∀F x in nhds a ×F nhds b. dist (snd x) (b) < e/2
    by (rule tendstoD[OF 2]) (use ‹e > 0› in auto)
    ultimately show ∀F x in nhds a ×F nhds b. dist (fst x + snd x) (a + b) < e
  proof eventually-elim
    case (elim x)
    then show ?case
    by (auto simp: dist-float-def) norm
  qed
  qed
  show (uminus ⟶ - a) (nhds a)
  using filterlim-ident[of nhds a]
  by (auto intro!: tendstoI dest!: tendstoD simp: dist-float-def dist-minus)
qed

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

```

$horner\text{-}eval' f x v 0 = v$   
 $| horner\text{-}eval' f x v (Suc i) = horner\text{-}eval' f x (f i + x * v) i$

**definition** *horner-eval*

**where**  $horner\text{-}eval f x n = horner\text{-}eval' f x 0 n$

**lemma** *horner-eval-cong*:

**assumes**  $\bigwedge i. i < n \implies f i = g i$

**assumes**  $x = y$

**assumes**  $n = m$

**shows**  $horner\text{-}eval f x n = horner\text{-}eval g y m$

**proof** –

$\{$   
 $\text{fix } v \text{ have } horner\text{-}eval' f x v n = horner\text{-}eval' g x v n$   
 $\text{using } assms(1) \text{ by } (induction n arbitrary: v, simp-all)$   
 $\}$

**thus** *?thesis*

**by** (*simp add: assms(2,3) horner-eval-def*)

**qed**

**lemma** *horner-eval-eq-setsum*:

**fixes**  $x::'a::linordered-idom$

**shows**  $horner\text{-}eval f x n = (\sum i < n. f i * x^i)$

**proof** –

$\{$   
 $\text{fix } v \text{ have } horner\text{-}eval' f x v n = (\sum i < n. f i * x^i) + v * x^n$   
 $\text{by } (induction n arbitrary: v, simp-all add: distrib-left mult.commute)$   
 $\}$

**thus** *?thesis* **by** (*simp add: horner-eval-def*)

**qed**

**lemma** *horner-eval-Suc[*simp*]*:

**fixes**  $x::'a::linordered-idom$

**shows**  $horner\text{-}eval f x (Suc n) = horner\text{-}eval f x n + (f n) * x^n$

**unfolding** *horner-eval-eq-setsum*

**by** *simp*

**lemma** *horner-eval-Suc'[*simp*]*:

**fixes**  $x::'a::\{comm-monoid-add, times\}$

**shows**  $horner\text{-}eval f x (Suc n) = f 0 + x * (horner\text{-}eval (\lambda i. f (Suc i)) x n)$

**proof** –

$\{$   
 $\text{fix } v \text{ have } horner\text{-}eval' f x v (Suc n) = f 0 + x * horner\text{-}eval' (\lambda i. f (Suc i))$   
 $x v n$

**by** (*induction n arbitrary: v, simp-all*)

$\}$

**thus** *?thesis* **by** (*simp add: horner-eval-def*)

**qed**

```

lemma horner-eval-0[simp]:
  shows horner-eval f x 0 = 0
  by (simp add: horner-eval-def)

lemma horner-eval'-interval:
  fixes x::'a::linordered-ring
  assumes  $\bigwedge i. i < n \implies f i \in \text{set-of } (g i)$ 
  assumes  $x \in_i I \ v \in_i V$ 
  shows horner-eval' f x v n  $\in_i$  horner-eval' g I V n
  using assms
  by (induction n arbitrary: v V) (auto intro!: plus-in-intervalI times-in-intervalI)

lemma horner-eval-interval:
  fixes x::'a::linordered-idom
  assumes  $\bigwedge i. i < n \implies f i \in \text{set-of } (g i)$ 
  assumes  $x \in \text{set-of } I$ 
  shows horner-eval f x n  $\in_i$  horner-eval g I n
  unfolding horner-eval-def
  using assms
  by (rule horner-eval'-interval) (auto simp: set-of-eq)

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

lemma real-of-float-eq-zero-iff[simp]:  $\text{real-of-float } x = 0 \longleftrightarrow x = 0$ 
  by (simp add: real-of-float-eq)

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 $\Rightarrow$ 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) = 0p  $\longleftrightarrow$  p +p q = 0p
  if [symmetric, simp]:  $\bigwedge x y. f (x + y) = f x + f y \ \bigwedge x. f x = 0 \longleftrightarrow x = 0$ 
  by (induction p q rule: polyadd.induct) auto

lemma zero-iffD:  $(\bigwedge x. f x = 0 \longleftrightarrow x = 0) \implies f 0 = 0$ 

```



**by** *auto*

**lemma** *map-poly-homo-polyadd*:

*map-poly f (p1 +<sub>p</sub> p2) = map-poly f p1 +<sub>p</sub> map-poly f p2*

**if** [*simp*]:  $\bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0$

**by** (*induction p1 p2 rule: polyadd.induct*)

(*auto simp: zero-iffD[OF that(2)] Let-def map-poly-homo-polyadd-eq-zero-iff*)

**lemma** *map-poly-homo-polyneg*:

*map-poly f (~<sub>p</sub> p1) = ~<sub>p</sub> (map-poly f p1)*

**if** [*simp*]:  $\bigwedge x y. f (-x) = -f x$

**by** (*induction p1 (auto simp: Let-def map-poly-homo-polyadd-eq-zero-iff)*)

**lemma** *map-poly-homo-polysub*:

*map-poly f (p1 -<sub>p</sub> p2) = map-poly f p1 -<sub>p</sub> map-poly f p2*

**if** [*simp*]:  $\bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0 \bigwedge x y. f (-x) = -f x$

**by** (*auto simp: polysub-def map-poly-homo-polyadd map-poly-homo-polyneg*)

**lemma** *map-poly-homo-polymul*:

*map-poly f (p1 \*<sub>p</sub> p2) = map-poly f p1 \*<sub>p</sub> map-poly f p2*

**if** [*simp*]:  $\bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0 \bigwedge x y. f (x * y) = f x * f y$

**by** (*induction p1 p2 rule: polymul.induct*)

(*auto simp: zero-iffD[OF that(2)] map-poly-homo-polyadd*)

**lemma** *map-poly-homo-polypow*:

*map-poly f (p1 ^<sub>p</sub> n) = map-poly f p1 ^<sub>p</sub> n*

**if** [*simp*]:  $\bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0 \bigwedge x y. f (x * y) = f x * f y$   
 $f 1 = 1$

**proof**(*induction n rule: nat-less-induct*)

**case** (*1 n*)

**then show** ?*case*

**apply** (*cases n*)

**apply** (*auto simp: map-poly-homo-polyadd map-poly-homo-polymul*)

**by** (*smt Suc-less-eq div2-less-self even-Suc odd-Suc-div-two map-poly-homo-polymul that*)

**qed**

**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*  $\Rightarrow$  *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*  
**by** (*induction p, simp-all*)

**lemma** *num-params-polyadd*:  
**shows** *num-params (p1 +<sub>p</sub> p2) ≤ max (num-params p1) (num-params p2)*  
**proof** (*induction p1 p2 rule: polyadd.induct*)  
**case** (*4 c n p c' n' p'*)  
**then show** *?case*  
**apply** (*simp only: num-params.simps polyadd.simps ac-simps not-less Let-def*  
*le-max-iff-disj max.bounded-iff split: if-split*)  
**apply** *simp*  
**apply** (*smt (verit, ccfv-SIG) dual-order.trans le-cases3*)  
**done**  
**qed** *auto*

**lemma** *num-params-polyneg*:  
**shows** *num-params (~<sub>p</sub> p) = num-params p*  
**by** (*induction p rule: polyneg.induct*) *simp-all*

**lemma** *num-params-polymul*:  
**shows** *num-params (p1 \*<sub>p</sub> p2) ≤ max (num-params p1) (num-params p2)*  
**proof** (*induction p1 p2 rule: polymul.induct*)  
**case** (*4 c n p c' n' p'*)  
**then show** *?case*  
**apply** (*cases n n' rule: linorder-cases*)  
**apply** (*simp-all only: num-params.simps polyadd.simps polymul.simps ac-simps*  
*not-less Let-def le-max-iff-disj max.bounded-iff split: if-split*)  
**apply** *simp-all*  
**apply** (*smt (verit, best) le-cases3 order-trans*)  
**apply** (*smt (verit, del-insts) le-max-iff-disj max-0L max-def num-params.simps(1)*  
*num-params.simps(8) num-params-polyadd*)  
**apply** (*smt (z3) dual-order.trans nle-le*)  
**done**  
**qed** *auto*

**lemma** *num-params-polypow*:  
**shows** *num-params (p ^<sub>p</sub> n) ≤ num-params p*  
**apply** (*induction n rule: polypow.induct*)  
**unfolding** *polypow.simps*  
**by** (*auto intro!: order-trans[OF num-params-polymul]*  
*simp: Let-def simp del: polypow.simps*)

```

lemma num-params-polynate:
  shows num-params (polynate p) ≤ num-params p
proof(induction p rule: polynate.induct)
  case (2 l r)
  thus ?case
    using num-params-polyadd[of polynate l polynate r]
    by simp
next
  case (3 l r)
  thus ?case
    using num-params-polyadd[of polynate l ~p (polynate r)]
    by (simp add: polysub-def num-params-polyneg)
next
  case (4 l r)
  thus ?case
    using num-params-polymul[of polynate l polynate r]
    by simp
next
  case (5 p)
  thus ?case
    by (simp add: num-params-polyneg)
next
  case (6 p n)
  thus ?case
    using num-params-polypow[of n polynate p]
    by simp
qed simp-all

```

```

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)
  by (induction p) (simp-all add: map-poly-homo-polyarith)

```

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

```

lemma Ipoly-real-float-equiv:
  fixes p::float poly and xs::float list
  assumes num-params p ≤ length xs
  shows Ipoly xs (p::real poly) = Ipoly xs p
  using assms by (induction p, simp-all)

```

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-ini xs
  assumes num-params p ≤ length xs
  shows Ipoly x p ∈ set-of (Ipoly xs (map-poly interval-of p))

```

```

using assms
by (induction p)
  (auto simp: all-in-i-def plus-in-intervalI minus-in-intervalI times-in-intervalI
    uminus-in-intervalI set-of-power-mono)

```

```

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))
using assms
by (induction p)
  (simp-all add: set-of-add-inc set-of-sub-inc set-of-mul-inc set-of-neg-inc set-of-pow-inc)

```

### 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 `polynat` 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)
unfolding atomize-conj
using assms
proof(induction p arbitrary: l r)
  case (Add p1 p2 l r)
  thus ?case
    apply(simp add: Add(1,2)[OF prod.collapse] split-beta)
    using max.coboundedI1 max.coboundedI2 prod.collapse
    by metis
  next
  case (Sub p1 p2 l r)
  thus ?case
    apply(simp add: Sub(1,2)[OF prod.collapse] split-beta)
    using max.coboundedI1 max.coboundedI2 prod.collapse
    by metis
  next
  case (Mul p1 p2 l r)
  thus ?case
    apply(simp add: Mul(1,2)[OF prod.collapse] split-beta algebra-simps)
    using max.coboundedI1 max.coboundedI2 prod.collapse
    by metis
  next
  case (Neg p l r)
  thus ?case by (simp add: Neg(1)[OF prod.collapse] split-beta)
  next
  case (Pw p n l r)
  thus ?case by (cases n) (simp-all add: Pw(1)[OF prod.collapse] split-beta)
  next
  case (CN c n p2)
  thus ?case
    apply(simp add: CN(1,2)[OF prod.collapse] split-beta algebra-simps)
    by (meson le-max-iff-disj prod.collapse)
qed (simp-all add: Let-def)

```

## 4 Splitting polynomials by degree

```

fun maxdegree :: ('a::zero) poly  $\Rightarrow$  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  $\Rightarrow$  'a::zero poly  $\Rightarrow$  '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  $\leq$  ord (is ?P1)
    and Ipoly x p = Ipoly x l + Ipoly x r (is ?P2)
    and num-params l  $\leq$  num-params p (is ?P3)
    and num-params r  $\leq$  num-params p (is ?P4)
  unfolding atomize-conj
  using assms
proof(induction p arbitrary: l r ord)
  case (C c l r ord)
  thus ?case by simp
next
  case (Bound v l r ord)
  thus ?case by (cases ord) simp-all
next
  case (Add p1 p2 l r ord)
  thus ?case by (cases ord) simp-all
next
  case (Sub p1 p2 l r ord)
  thus ?case by (cases ord) simp-all
next
  case (Mul p1 p2 l r ord)
  thus ?case by (cases ord) simp-all
next
  case (Neg p l r ord)

```

```

    thus ?case by (cases ord) simp-all
next
  case (Pw p k l r ord)
  thus ?case by (cases ord) simp-all
next
  case (CN c v p l r ord)
  then show ?case
  proof(cases ord)
    case (Suc m)
    obtain cl cr where cl-cr-def: (cl, cr) = split-by-degree (Suc m) c
    by (cases split-by-degree (Suc m) c, simp)
    obtain pl pr where pl-pr-def: (pl, pr) = split-by-degree m p
    by (cases split-by-degree m p, simp)
    have [simp]:  $Ipoly\ x\ p = Ipoly\ x\ pl + Ipoly\ x\ pr$ 
    using CN(2)[OF pl-pr-def]
    by (cases ord) simp-all
  from CN(3)
  have l-decomp:  $l = CN\ cl\ v\ pl$  and r-decomp:  $r = CN\ cr\ v\ pr$ 
  by (simp-all add: Suc cl-cr-def[symmetric] pl-pr-def[symmetric])
  show ?thesis
  using CN(1)[OF cl-cr-def] CN(2)[OF pl-pr-def]
  unfolding l-decomp
  by (cases p) (auto simp add: l-decomp r-decomp algebra-simps Suc)
qed simp
qed

```

Operations on lists.

```

lemma length-map2[simp]:  $length\ (map2\ f\ a\ b) = \min\ (length\ a)\ (length\ b)$ 
proof(induction map2 f a b arbitrary: a b)
  case (Nil a b)
  hence  $a = [] \mid b = []$ 
  by(cases a, simp, cases b, simp-all)
  then show ?case
  by auto
next
  case (Cons x c a b)
  have  $0 < length\ a \wedge 0 < length\ b$ 
  using Cons(2)
  by (cases a, simp, cases b, simp-all)
  then obtain xa ar xb br
  where a-decomp[simp]:  $a = xa \# ar$ 
  and b-decomp[simp]:  $b = xb \# br$ 
  by (cases a, simp-all, cases b, simp-all)
  show ?case
  using Cons
  by simp
qed

```

lemma map2-nth[simp]:

```

    assumes  $n < \text{length } a$ 
    assumes  $n < \text{length } b$ 
    shows  $(\text{map2 } f \ a \ b)!n = f \ (a!n) \ (b!n)$ 
    using assms
  proof(induction  $n$  arbitrary:  $a \ b$ )
    case (0  $a \ b$ )
      have  $0 < \text{length } a$  and  $0 < \text{length } b$ 
      using 0
      by simp-all
    thus ?case
      using 0
      by simp
  next
    case (Suc  $n \ a \ b$ )
    from Suc.prems have  $0 < \text{length } a$   $0 < \text{length } b$   $n < \text{length } (tl \ a)$   $n < \text{length } (tl \ b)$ 
    using Suc.prems by auto
    have  $\text{map2 } f \ a \ b = \text{map2 } f \ (hd \ a \ \# \ tl \ a) \ (hd \ b \ \# \ tl \ b)$ 
    using  $\langle 0 < \text{length } a \rangle \ \langle 0 < \text{length } b \rangle$ 
    by simp
    also have  $\dots ! \text{Suc } n = \text{map2 } f \ (tl \ a) \ (tl \ b) ! n$ 
    by simp
    also have  $\dots = f \ (tl \ a ! n) \ (tl \ b ! n)$ 
    using  $\langle n < \text{length } (tl \ a) \rangle \ \langle n < \text{length } (tl \ b) \rangle$  by (rule Suc.IH)
    also have  $tl \ a ! n = (hd \ a \ \# \ tl \ a) ! \text{Suc } n$  by simp
    also have  $(hd \ a \ \# \ tl \ a) = a$  using  $\langle 0 < \text{length } a \rangle$  by simp
    also have  $tl \ b ! n = (hd \ b \ \# \ tl \ b) ! \text{Suc } n$  by simp
    also have  $(hd \ b \ \# \ tl \ b) = b$  using  $\langle 0 < \text{length } b \rangle$  by simp
    finally show ?case .
  qed

```

Translating a polynomial by a vector.

```

fun poly-translate :: 'a list  $\Rightarrow$  'a poly  $\Rightarrow$  '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  $\text{num-params } p \leq \text{length } x$ 

```



```

assumes  $\text{length } x = \text{length } v$ 
shows  $\text{Ipoly } x (\text{poly-translate } v \ p) = \text{Ipoly } (\text{map2 } (+) \ x \ v) \ p$ 
using assms
by (induction p, simp-all)

lemma real-poly-translate:
  assumes  $\text{num-params } p \leq \text{length } v$ 
  shows  $\text{Ipoly } x (\text{map-poly real-of-float } (\text{poly-translate } v \ p)) = \text{Ipoly } x (\text{poly-translate } v \ (\text{map-poly real-of-float } p))$ 
  using assms
  by (induction p, simp-all)

lemma num-params-poly-translate[simp]:
  shows  $\text{num-params } (\text{poly-translate } v \ p) = \text{num-params } p$ 
  by (induction p, simp-all)

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:  $(\text{of-nat } n :: \text{real}) = (\text{of-nat } n :: \text{float})$ 
by (induction n, simp-all add: of-nat-def)

lemma fact-real-float-equiv:  $(\text{fact } n :: \text{float}) = (\text{fact } n :: \text{real})$ 
by (induction n simp-all)

lemma Some-those-length:
   $\text{those } ys = \text{Some } xs \implies \text{length } xs = \text{length } ys$ 
by (induction ys arbitrary: xs (auto split: option.splits))

lemma those-eq-None-iff:  $\text{those } ys = \text{None} \longleftrightarrow \text{None} \in \text{set } ys$ 
by (induction ys (auto simp: split: option.splits))

lemma those-eq-Some-iff:  $\text{those } ys = (\text{Some } xs) \longleftrightarrow (ys = \text{map } \text{Some } xs)$ 
by (induction ys arbitrary: xs (auto simp: split: option.splits))

lemma Some-those-nth:
  assumes  $\text{those } ys = \text{Some } xs$ 
  assumes  $i < \text{length } xs$ 
  shows  $\text{Some } (xs!i) = ys!i$ 
  using Some-those-length[OF assms(1)] assms
  by (induction xs ys arbitrary: i rule: list-induct2)
  (auto split: option.splits nat.splits simp: nth-Cons)

```

**lemma** *fun-pow*:  $f^{\wedge}n = (\lambda x. (f\ x)^{\wedge}n)$

**by** (*induction n*, *simp-all*)

**context includes** *floatarith-notation* **begin**

Translate floatarith expressions by a vector of floats.

**fun** *fa-translate* :: *float list*  $\Rightarrow$  *floatarith*  $\Rightarrow$  *floatarith*

**where** *fa-translate* *v* (*Add* *a* *b*) = *Add* (*fa-translate* *v* *a*) (*fa-translate* *v* *b*)  
| *fa-translate* *v* (*Minus* *a*) = *Minus* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Mult* *a* *b*) = *Mult* (*fa-translate* *v* *a*) (*fa-translate* *v* *b*)  
| *fa-translate* *v* (*Inverse* *a*) = *Inverse* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Cos* *a*) = *Cos* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Arctan* *a*) = *Arctan* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Min* *a* *b*) = *Min* (*fa-translate* *v* *a*) (*fa-translate* *v* *b*)  
| *fa-translate* *v* (*Max* *a* *b*) = *Max* (*fa-translate* *v* *a*) (*fa-translate* *v* *b*)  
| *fa-translate* *v* (*Abs* *a*) = *Abs* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Sqrt* *a*) = *Sqrt* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Exp* *a*) = *Exp* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Ln* *a*) = *Ln* (*fa-translate* *v* *a*)  
| *fa-translate* *v* (*Var* *n*) = *Add* (*Var* *n*) (*Num* (*v!**n*))  
| *fa-translate* *v* (*Power* *a* *n*) = *Power* (*fa-translate* *v* *a*) *n*  
| *fa-translate* *v* (*Powr* *a* *b*) = *Powr* (*fa-translate* *v* *a*) (*fa-translate* *v* *b*)  
| *fa-translate* *v* (*Floor* *x*) = *Floor* (*fa-translate* *v* *x*)  
| *fa-translate* *v* (*Num* *c*) = *Num* *c*  
| *fa-translate* *v* *Pi* = *Pi*

**lemma** *fa-translate-correct*:

**assumes** *max-Var-floatarith* *f*  $\leq$  *length I*

**assumes** *length v* = *length I*

**shows** *interpret-floatarith* (*fa-translate* *v* *f*) *I* = *interpret-floatarith* *f* (*map2* (+) *I* *v*)

**using** *assms*

**by** (*induction f*, *simp-all*)

**primrec** *vars-floatarith* **where**

*vars-floatarith* (*Add* *a* *b*) = (*vars-floatarith* *a*)  $\cup$  (*vars-floatarith* *b*)  
| *vars-floatarith* (*Mult* *a* *b*) = (*vars-floatarith* *a*)  $\cup$  (*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*)  $\cup$  (*vars-floatarith* *b*)  
| *vars-floatarith* (*Min* *a* *b*) = (*vars-floatarith* *a*)  $\cup$  (*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)
  by (induction x) auto

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)))
  by (induction fa) (auto split: if-splits simp: Max-Un Max-eq-iff max-def)

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 interval 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 –

```

```

have bounded: bounded-by i (map Some I)
  using that(2)
  unfolding bounded-by-def
  by (auto simp: bounds-of-interval-eq-lower-upper set-of-eq)
from that have Some: approx prec f (map Some I) = Some ivl
  by (auto simp: lower-Interval upper-Interval min-def split: option.splits if-splits)
from approx[OF bounded Some]
show ?thesis by (auto simp: set-of-eq)
qed

```

## 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  $\Rightarrow$  'a)  $\Rightarrow$  'a *poly*  $\Rightarrow$  '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  $\Rightarrow$  real)  $\Rightarrow$  *taylor-model*  $\Rightarrow$  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  $\bigwedge i. i < \text{num-params } p \implies xs \ ! \ i = ys \ ! \ i$   
 using *that*  
 by (*induction* *p*; *auto*)

**lemma** *insertion-num-params-cong*: *insertion* e *p* = *insertion* f *p*  
 if  $\bigwedge i. i < \text{num-params } p \implies e \ i = f \ i$   
 using *that*  
 by (*induction* *p*; *auto*)

**lemma** *insertion-eq-IPolyI*: *insertion* *xs* *p* = *Ipoly* *ys* *p*  
 if  $\bigwedge i. i < \text{num-params } p \implies xs \ i = ys \ ! \ i$   
 using *that*  
 by (*induction* *p*; *auto*)

**lemma** *Ipoly-eq-insertionI*: *Ipoly* *ys* *p* = *insertion* *xs* *p*  
 if  $\bigwedge i. i < \text{num-params } p \implies xs \ i = ys \ ! \ i$

**using** *that*  
**by** (*induction p; auto*)

**lemma** *range-tmI*:  
 $x \in_i \text{range-tm } e \text{ } tm$   
**if**  $x: x \in_i \text{interval-of } (\text{insertion } e ((tm\text{-poly } tm))) + \text{real-interval } (tm\text{-bound } tm)$   
**for**  $e::nat \Rightarrow real$   
**by** (*auto simp: range-tm-def x*)

**lemma** *range-tmD*:  
 $x \in_i \text{interval-of } (\text{insertion } e (tm\text{-poly } tm)) + \text{real-interval } (tm\text{-bound } tm)$   
**if**  $x \in_i \text{range-tm } e \text{ } tm$   
**for**  $e::nat \Rightarrow real$   
**using** *that*  
**by** (*auto simp: range-tm-def*)

### 5.3 Interval bounds for Taylor models

Bound a polynomial by simply approximating it with interval arguments.

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

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

**fun** *compute-bound-tm* ::  $nat \Rightarrow float \text{ interval } list \Rightarrow float \text{ interval } list \Rightarrow \text{taylor-model} \Rightarrow float \text{ interval}$   
**where**  $\text{compute-bound-tm } prec \text{ } I \text{ } a \text{ } (TaylorModel \text{ } p \text{ } e) = \text{compute-bound-poly } prec$   
 $p \text{ } I \text{ } a + e$

**lemma** *compute-bound-tm-def*:  
 $\text{compute-bound-tm } \text{prec } I \ a \ tm = \text{compute-bound-poly } \text{prec } (tm\text{-poly } tm) \ I \ a + (tm\text{-bound } tm)$   
**by** (*cases tm*) *auto*

**lemma** *real-of-float-in-real-interval-of*[*intro, simp*]:  $\text{real-of-float } x \in_r X \text{ if } x \in_i X$   
**using** *that*  
**by** (*auto simp: set-of-eq*)

**lemma** *in-set-of-round-interval*[*intro, simp*]:  
 $x \in_r \text{round-interval } \text{prec } X \text{ if } x \in_r X$   
**using** *round-ivl-correct*[*of X prec*] *that*  
**by** (*auto simp: set-of-eq*)

**lemma** *in-set-real-minus-interval*[*intro, simp*]:  
 $x - y \in_r X - Y \text{ if } x \in_r X \ y \in_r Y$   
**using** *that*  
**by** (*auto simp: set-of-eq*)

**lemma** *real-interval-plus*:  $\text{real-interval } (a + b) = \text{real-interval } a + \text{real-interval } b$   
**by** (*simp add: interval-eqI*)

**lemma** *real-interval-uminus*:  $\text{real-interval } (-b) = - \text{real-interval } b$   
**by** (*simp add: interval-eqI*)

**lemma** *real-interval-of*:  $\text{real-interval } (\text{interval-of } b) = \text{interval-of } b$   
**by** (*simp add: interval-eqI*)

**lemma** *real-interval-minus*:  $\text{real-interval } (a - b) = \text{real-interval } a - \text{real-interval } b$   
**using** *real-interval-plus*[*of a -b*] *real-interval-uminus*[*of b*]  
**by** (*auto simp: interval-eq-iff*)

**lemma** *in-set-real-plus-interval*[*intro, simp*]:  
 $x + y \in_r X + Y \text{ if } x \in_r X \ y \in_r Y$   
**using** *that*  
**by** (*auto simp: set-of-eq*)

**lemma** *in-set-neg-plus-interval*[*intro, simp*]:  
 $-y \in_r -Y \text{ if } y \in_r Y$   
**using** *that*  
**by** (*auto simp: set-of-eq*)

**lemma** *in-set-real-times-interval*[*intro, simp*]:  
 $x * y \in_r X * Y \text{ if } x \in_r X \ y \in_r Y$   
**using** *that*  
**by** (*auto simp: real-interval-times intro!: times-in-intervalI*)

**lemma** *real-interval-one*:  $\text{real-interval } 1 = 1$

**by** (*simp add: interval-eqI*)

**lemma** *real-interval-zero*: *real-interval 0 = 0*  
**by** (*simp add: interval-eqI*)

**lemma** *real-interval-power*: *real-interval (a ^ b) = real-interval a ^ b*  
**by** (*induction b arbitrary: a*)  
*(auto simp: real-interval-times real-interval-one)*

**lemma** *in-set-real-power-interval*[*intro, simp*]:  
 $x \wedge n \in_r X \wedge n$  **if**  $x \in_r X$   
**using** *that*  
**by** (*auto simp: real-interval-power intro!: set-of-power-mono*)

**lemma** *power-float-interval-real-interval*[*intro, simp*]:  
 $x \wedge n \in_r \text{power-float-interval prec } n X$  **if**  $x \in_r X$   
**by** (*auto simp: real-interval-power that intro!: power-float-intervalI*)

**lemma** *in-set-mult-float-interval*[*intro, simp*]:  
 $x * y \in_r \text{mult-float-interval prec } X Y$  **if**  $x \in_r X$   $y \in_r Y$   
**using** *mult-float-interval[of X Y] in-set-real-times-interval[OF that] that(1) that(2)*  
**by** *blast*

**lemma** *in-set-real-minus-swapI*:  $e \ i \in_r I \ ! \ i - a \ ! \ i$   
**if**  $x - e \ i \in_r a \ ! \ i \ x \in_r I \ ! \ i$   
**using** *that*  
**by** (*auto simp: set-of-eq*)

**definition** *develops-at-within*::(*nat*  $\Rightarrow$  *real*)  $\Rightarrow$  *float interval list*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *bool*  
**where** *develops-at-within*  $e \ a \ I \longleftrightarrow (a \text{ all-subset } I) \wedge (\forall i < \text{length } I. e \ i \in_r I \ ! \ i - a \ ! \ i)$

**lemma** *develops-at-withinI*:  
**assumes** *all-in*:  $a \text{ all-subset } I$   
**assumes**  $e$ :  $\bigwedge i. i < \text{length } I \implies e \ i \in_r I \ ! \ i - a \ ! \ i$   
**shows** *develops-at-within*  $e \ a \ I$   
**using** *assms* **by** (*auto simp: develops-at-within-def*)

**lemma** *develops-at-withinD*:  
**assumes** *develops-at-within*  $e \ a \ I$   
**shows**  $a \text{ all-subset } I$   
 $\bigwedge i. i < \text{length } I \implies e \ i \in_r I \ ! \ i - a \ ! \ i$   
**using** *assms* **by** (*auto simp: develops-at-within-def*)

**lemma** *compute-bound-poly-correct*:  
**fixes**  $p$ ::*float poly*  
**assumes** *num-params*  $p \leq \text{length } I$   
**assumes** *dev*: *develops-at-within*  $e \ a \ I$

```

shows insertion  $e$  ( $p :: \text{real poly}$ )  $\in_r \text{compute-bound-poly prec (map-poly interval-of } p) I a$ 
using assms(1)
proof (induction p)
  case ( $C\ x$ )
  then show ?case by auto
next
  case ( $Bound\ i$ )
  then show ?case
    using dev
    by (auto simp: develops-at-within-def)
next
  case ( $Add\ p1\ p2$ )
  then show ?case by force
next
  case ( $Sub\ p1\ p2$ )
  then show ?case by force
next
  case ( $Mul\ p1\ p2$ )
  then show ?case by force
next
  case ( $Neg\ p$ )
  then show ?case by force
next
  case ( $Pw\ p\ x2a$ )
  then show ?case by force
next
  case ( $CN\ p1\ i\ p2$ )
  then show ?case
    using dev
    by (auto simp: develops-at-within-def)
qed

```

```

lemma compute-bound-tm-correct:
  fixes  $I :: \text{float interval list}$  and  $f :: \text{real list} \Rightarrow \text{real}$ 
  assumes  $n: \text{num-params (tm-poly } t) \leq \text{length } I$ 
  assumes  $dev: \text{develops-at-within } e\ a\ I$ 
  assumes  $x0: x0 \in_i \text{range-tm } e\ t$ 
  shows  $x0 \in_r \text{compute-bound-tm prec } I\ a\ t$ 
proof –
  let ? $I = \text{insertion } e\ (\text{tm-poly } t)$ 
  have  $x0 = ?I + (x0 - ?I)$  by simp
  also have  $\dots \in_r \text{compute-bound-tm prec } I\ a\ t$ 
    unfolding compute-bound-tm-def
    apply (rule in-set-real-plus-interval)
    apply (rule compute-bound-poly-correct)
    apply (rule assms)
    apply (rule dev)
    using range-tmD[OF x0]

```



by (auto simp: set-of-eq)  
 finally show  $x0 \in_r \text{compute-bound-tm prec } I \text{ a } t$  .  
 qed

**lemma** *compute-bound-tm-correct-subset*:  
 fixes  $I :: \text{float interval list}$  and  $f :: \text{real list} \Rightarrow \text{real}$   
 assumes  $n: \text{num-params } (\text{tm-poly } t) \leq \text{length } I$   
 assumes  $\text{dev: develops-at-within } e \text{ a } I$   
 shows  $\text{set-of } (\text{range-tm } e \text{ t}) \subseteq \text{set-of } (\text{real-interval } (\text{compute-bound-tm prec } I \text{ a } t))$   
 using *assms*  
 by (auto intro!: *compute-bound-tm-correct*)

**lemma** *compute-bound-poly-mono*:  
 assumes  $\text{num-params } p \leq \text{length } I$   
 assumes  $\text{mem: } I \text{ all-subset } J \text{ a all-subset } I$   
 shows  $\text{set-of } (\text{compute-bound-poly prec } p \text{ I a}) \subseteq \text{set-of } (\text{compute-bound-poly prec } p \text{ J a})$   
 using *assms*(1)  
**proof** (*induction p arbitrary: a*)  
 case (C x)  
 then show ?case by auto  
**next**  
 case (Bound x)  
 then show ?case using *mem*  
 by (*simp add: round-interval-mono set-of-sub-inc*)  
**next**  
 case (Add p1 p2)  
 then show ?case using *mem*  
 by (*simp add: round-interval-mono set-of-add-inc*)  
**next**  
 case (Sub p1 p2)  
 then show ?case using *mem*  
 by (*simp add: round-interval-mono set-of-sub-inc*)  
**next**  
 case (Mul p1 p2)  
 then show ?case using *mem*  
 by (*simp add: round-interval-mono mult-float-interval-mono*)  
**next**  
 case (Neg p)  
 then show ?case using *mem*  
 by (*simp add: round-interval-mono set-of-neg-inc*)  
**next**  
 case (Pw p x2a)  
 then show ?case using *mem*  
 by (*simp add: power-float-interval-mono*)  
**next**  
 case (CN p1 x2a p2)  
 then show ?case using *mem*

```

    by (simp add: round-interval-mono mult-float-interval-mono'
        set-of-add-inc set-of-sub-inc)
qed

lemma compute-bound-tm-mono:
  fixes I :: float interval list and f :: real list  $\Rightarrow$  real
  assumes num-params (tm-poly t)  $\leq$  length I
  assumes I all-subset J
  assumes a all-subset I
  shows set-of (compute-bound-tm prec I a t)  $\subseteq$  set-of (compute-bound-tm prec J
    a t)
  apply (simp add: compute-bound-tm-def)
  apply (rule set-of-add-inc-left)
  apply (rule compute-bound-poly-mono)
  using assms
  by (auto simp: set-of-eq)

```

## 5.4 Computing taylor models for basic, univariate functions

```

definition tm-const :: float  $\Rightarrow$  taylor-model
  where tm-const c = TaylorModel (poly.C c) 0

```

context includes floatarith-notation begin

```

definition tm-pi :: nat  $\Rightarrow$  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  $\in_r$  0
  by (auto simp: set-of-eq)

```

```

lemma range-TM-tm-const[simp]: range-tm e (tm-const c) = interval-of c
  by (auto simp: range-tm-def real-interval-zero tm-const-def)

```

```

lemma num-params-tm-const[simp]: num-params (tm-poly (tm-const c)) = 0
  by (auto simp: tm-const-def)

```

```

lemma num-params-tm-pi[simp]: num-params (tm-poly (tm-pi prec)) = 0
  by (auto simp: tm-pi-def Let-def)

```

```

lemma range-tm-tm-pi: pi  $\in_i$  range-tm e (tm-pi prec)

```

proof –

```

  have  $\bigwedge$ prec. real-of-float (lb-pi prec)  $\leq$  real-of-float (ub-pi prec)
    using iffD1[OF atLeastAtMost-iff, OF pi-boundaries]
    using order-trans by auto
  then obtain ivl-pi where ivl-pi-def: compute-bound-fa prec Pi [] = Some ivl-pi
    by (simp add: approx.simps)

```

```

show ?thesis
  unfolding range-tm-def Let-def
  using compute-bound-fa-correct[OF ivl-pi-def, of []]
  by (auto simp: set-of-eq Let-def centered-def ivl-pi-def tm-pi-def
    simp del: compute-bound-fa.simps)
qed

```

#### 5.4.1 Derivations of floatarith expressions

Compute the nth derivative of a floatarith expression

```

fun deriv :: nat  $\Rightarrow$  floatarith  $\Rightarrow$  nat  $\Rightarrow$  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
  using assms
proof(induction f)
  case (Power f m)
  then show ?case
    by (cases m) (auto simp: isDERIV-Power)
qed (simp-all add: numeral-eq-Suc add-nonneg-eq-0-iff )

```

```

lemma isDERIV-is-analytic:
  isDERIV i (Taylor-Models.deriv i f n) xs
  if isDERIV i f xs
  using isDERIV-DERIV-floatarith that
  by(induction n) auto

```

```

lemma deriv-correct:
  assumes isDERIV i f (xs[i:=t]) i < length xs
  shows (( $\lambda x$ . 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)
  apply(simp)
  apply (rule has-field-derivative-at-within)
  apply(rule DERIV-floatarith)
  apply fact
  apply (rule isDERIV-is-analytic)
  apply fact
  done

```

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

TODO: Extend to Arctan, Log!

```

fun deriv-rec :: floatarith  $\Rightarrow$  nat  $\Rightarrow$  floatarith
  where deriv-rec (Exp (Var 0)) - = Exp (Var 0)

```

```

| deriv-rec (Cos (Var 0)) n = (case n mod 4
  of 0  $\Rightarrow$  Cos (Var 0)
   | Suc 0  $\Rightarrow$  Minus (Sin (Var 0))
   | Suc (Suc 0)  $\Rightarrow$  Minus (Cos (Var 0))
   | Suc (Suc (Suc 0))  $\Rightarrow$  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 (( $\lambda x$ . 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)
  apply(cases (f, n) rule: deriv-rec.cases)

```

**apply**(safe)

**using** *assms deriv-correct[OF assms]*

**proof**–

**assume**  $f = \text{Cos } (\text{Var } 0)$

**have** *n-mod-4-cases*:  $n \bmod 4 = 0 \mid n \bmod 4 = 1 \mid n \bmod 4 = 2 \mid n \bmod 4 = 3$

**by** *auto*

**have** *Sin-sin*:  $(\lambda xs. \text{interpret-floatarith } (\text{Sin } (\text{Var } 0)) xs) = (\lambda xs. \text{sin } (xs!0))$

**by** *simp*

**show**  $((\lambda x. \text{interpret-floatarith } (\text{deriv-rec } (\text{Cos } (\text{Var } 0)) n) (xs[0:=x])) \text{ has-real-derivative } \text{interpret-floatarith } (\text{deriv-rec } (\text{Cos } (\text{Var } 0)) (\text{Suc } n)) (xs[0:=t]))$   
(at t within S)

**using** *n-mod-4-cases assms*

**by** (*auto simp add: mod-Suc Sin-sin field-differentiable-minus*  
*intro!: derivative-eq-intros*)

**next**

**assume** *f-def*:  $f = \text{Inverse } (\text{Var } 0)$  **and** *isDERIV* 0 f (xs[0:=t])

**hence**  $t \neq 0$  **using** *assms*

**by** *simp*

{

**fix**  $n::\text{nat}$  **and**  $x::\text{real}$

**assume**  $x \neq 0$

**moreover** **have**  $(n \bmod 2 = 0 \wedge \text{Suc } n \bmod 2 = 1) \vee (n \bmod 2 = 1 \wedge \text{Suc } n \bmod 2 = 0)$

**by** (*cases n rule: parity-cases*) *auto*

**ultimately** **have**  $\text{interpret-floatarith } (\text{deriv-rec } f n) (xs[0:=x]) = \text{fact } n * (-1::\text{real})^n / (x^{\text{Suc } n})$

**using** *assms* **by** (*auto simp add: f-def field-simps fact-real-float-equiv*)

}

**note** *closed-formula* = *this*

**have**  $((\lambda x. \text{inverse } (x^{\text{Suc } n})) \text{ has-real-derivative } -\text{real } (\text{Suc } n) * \text{inverse } (t^{\text{Suc } (\text{Suc } n)}))$  (at t)

**using** *DERIV-inverse-fun[OF DERIV-pow[where n=Suc n], where s=UNIV]*

**apply**(*rule iffD1[OF DERIV-cong-ev[OF refl], rotated 2]*)

```

using  $\langle t \neq 0 \rangle$ 
by (simp-all add: divide-simps)
hence  $((\lambda x. \text{fact } n * (-1::\text{real})^n * \text{inverse } (x \wedge \text{Suc } n)) \text{ has-real-derivative } \text{fact } ( \text{Suc } n) * (-1) \wedge \text{Suc } n / t \wedge \text{Suc } (\text{Suc } n)) \text{ (at } t)$ 
apply (rule iffD1[OF DERIV-cong-ev, OF refl - - DERIV-cmult[where c=fact n * (-1::real)^n], rotated 2])
using  $\langle t \neq 0 \rangle$ 
by (simp-all add: field-simps distrib-left)
then show  $((\lambda x. \text{interpret-floatarith } (\text{deriv-rec } (\text{Inverse } (\text{Var } 0)) \text{ } n) \text{ } (xs[0:=x])) \text{ has-real-derivative } \text{interpret-floatarith } (\text{deriv-rec } (\text{Inverse } (\text{Var } 0)) \text{ } (\text{Suc } n)) \text{ } (xs[0:=t])) \text{ (at } t \text{ within } S)$ 
apply -
apply (rule has-field-derivative-at-within)
apply (rule iffD1[OF DERIV-cong-ev[OF refl - closed-formula[OF  $\langle t \neq 0 \rangle$ , symmetric]], unfolded f-def, rotated 1])
apply simp
using assms
by (simp, safe, simp-all add: fact-real-float-equiv inverse-eq-divide even-iff-mod-2-eq-zero)
qed (use assms in  $\langle \text{simp-all add: has-field-derivative-subset[OF DERIV-exp subset-UNIV] \rangle$ )

```

```

lemma deriv-rec-0-idem[simp]:
  shows  $\text{deriv-rec } f \ 0 = f$ 
  by (cases (f, 0::nat) rule: deriv-rec.cases, simp-all)

```

#### 5.4.2 Computing Taylor models for arbitrary univariate expressions

```

fun tmf-c ::  $\text{nat} \Rightarrow \text{float interval list} \Rightarrow \text{floatarith} \Rightarrow \text{nat} \Rightarrow \text{float interval option}$ 
  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 ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{float interval list} \Rightarrow \text{float list} \Rightarrow \text{floatarith} \Rightarrow \text{float interval list option}$ 
  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 ::  $\text{float interval list} \Rightarrow \text{float poly} \times \text{float interval poly}$ 
  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*  $\Rightarrow$  *nat*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *float list*  $\Rightarrow$  *floatarith*  $\Rightarrow$  *taylor-model option*

**where** *tm-floatarith prec ord I a f* = (  
*map-option* ( $\lambda cs.$   
*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*  
**using** *compute-bound-fa-correct[OF assms(2)[unfolded tmf-c.simps], where i=a]*  
*assms(1)*  
**by** (*simp add: divide-real-def fact-real-float-equiv*)

**lemma** *tmf-ivl-cs-length*:

**assumes** *tmf-ivl-cs prec n A a f = Some cs*  
**shows** *length cs = n + 1*  
**by** (*simp add: Some-those-length[OF assms[unfolded tmf-ivl-cs.simps]]*)

**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**  $\bigwedge i. i < \text{ord} \implies \text{tmf-c prec (map interval-of a) f i = Some (cs!i)}$   
**and** *tmf-c prec I f ord = Some (cs!ord)*  
**and** *length cs = Suc ord*

**proof**—

**from** *tmf-ivl-cs-length[OF assms(2)]*  
**show** *tmf-c prec I f ord = Some (cs!ord)*  
**by** (*metis Some-those-nth assms(2) diff-zero length-map length-upt less-add-one nth-append-length tmf-ivl-cs.simps*)

**next**

**fix** *i* **assume** *i < ord*  
**have** *Some (cs!i) = (map (tmf-c prec a f) [0..*ord*] @ [tmf-c prec I f *ord*]) ! i*  
**apply**(*rule Some-those-nth*)  
**using** *assms(2) tmf-ivl-cs-length  $\langle i < \text{ord} \rangle$*   
**by** *simp-all*  
**then show** *tmf-c prec a f i = Some (cs!i)*  
**using**  $\langle i < \text{ord} \rangle$

```

    by (simp add: nth-append)
next
  show length cs = Suc ord
    using assms
    by (auto simp: split-beta' those-eq-Some-iff list-eq-iff-nth-eq)
qed

lemma Ipoly-fst-tmf-polys:
  
$$\text{Ipoly } xs \text{ (fst (tmf-polys } z)) = (\sum i < \text{length } z. xs ! i * (\text{mid } (z ! i)))$$

  for  $xs :: \text{real list}$ 
proof (induction z)
  case (Cons z zs)
  show ?case
    unfolding list.size add-Suc-right sum.lessThan-Suc-shift
    by (auto simp: split-beta' Let-def nth-Cons Cons sum-distrib-left ac-simps)
qed simp

lemma insertion-fst-tmf-polys:
  
$$\text{insertion } e \text{ (fst (tmf-polys } z)) = (\sum i < \text{length } z. e * 0^i * (\text{mid } (z ! i)))$$

  for  $e :: \text{nat} \Rightarrow \text{real}$ 
proof (induction z)
  case (Cons z zs)
  show ?case
    unfolding list.size add-Suc-right sum.lessThan-Suc-shift
    by (auto simp: split-beta' Let-def nth-Cons Cons sum-distrib-left ac-simps)
qed simp

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

proof (induction z)
  case (Cons z zs)
  show ?case
    using Cons[THEN set-of-mul-inc-right]
    unfolding list.size add-Suc-right sum.lessThan-Suc-shift
    by (auto simp: split-beta' Let-def nth-Cons sum-distrib-left ac-simps
      elim!: plus-in-intervalE intro!: plus-in-intervalI)
qed (auto simp: real-interval-zero)

lemma zero-interval[intro,simp]:  $0 \in_i 0$ 
  by (simp add: set-of-eq)

lemma sum-in-intervalI:  $\text{sum } f X \in_i \text{sum } g X \text{ if } \bigwedge x. x \in X \implies f x \in_i g x$ 
  for  $f :: - \Rightarrow 'a :: \text{ordered-comm-monoid-add}$ 
  using that
proof (induction X rule: infinite-finite-induct)
  case (insert x F)
  then show ?case
    by (auto intro!: plus-in-intervalI)

```

**qed** *simp-all*

**lemma** *set-of-sum-subset*:  $\text{set-of } (\text{sum } f \ X) \subseteq \text{set-of } (\text{sum } g \ X)$   
**if**  $\bigwedge x. x \in X \implies \text{set-of } (f \ x) \subseteq \text{set-of } (g \ x)$   
**for**  $f :: \rightarrow 'a::\text{linordered-ab-group-add interval}$   
**using** *that*  
**by** (*induction X rule: infinite-finite-induct*) (*simp-all add: set-of-add-inc*)

**lemma** *interval-of-plus*:  $\text{interval-of } (a + b) = \text{interval-of } a + \text{interval-of } b$   
**by** (*simp add: interval-eqI*)

**lemma** *interval-of-uminus*:  $\text{interval-of } (- a) = - \text{interval-of } a$   
**by** (*simp add: interval-eqI*)

**lemma** *interval-of-zero*:  $\text{interval-of } 0 = 0$   
**by** (*simp add: interval-eqI*)

**lemma** *interval-of-sum*:  $\text{interval-of } (\text{sum } f \ X) = \text{sum } (\lambda x. \text{interval-of } (f \ x)) \ X$   
**by** (*induction X rule: infinite-finite-induct*) (*auto simp: interval-of-plus interval-of-zero*)

**lemma** *interval-of-prod*:  $\text{interval-of } (a * b) = \text{interval-of } a * \text{interval-of } b$   
**by** (*simp add: lower-times upper-times interval-eqI*)

**lemma** *in-set-of-interval-of[simp]*:  $x \in_i (\text{interval-of } y) \longleftrightarrow x = y$  **for**  $x \ y::'a::\text{order}$   
**by** (*auto simp: set-of-eq*)

**lemma** *real-interval-Ipoly*:  $\text{real-interval } (\text{Ipoly } xs \ p) = \text{Ipoly } (\text{map } \text{real-interval } xs)$   
*(map-poly real-interval p)*  
**if**  $\text{num-params } p \leq \text{length } xs$   
**using** *that*  
**by** (*induction p*)  
*(auto simp: real-interval-plus real-interval-minus real-interval-times real-interval-uminus real-interval-power)*

**lemma** *num-params-tmf-polys1*:  $\text{num-params } (\text{fst } (\text{tmf-polys } z)) \leq \text{Suc } 0$   
**by** (*induction z*) (*auto simp: split-beta' Let-def*)

**lemma** *num-params-tmf-polys2*:  $\text{num-params } (\text{snd } (\text{tmf-polys } z)) \leq \text{Suc } 0$   
**by** (*induction z*) (*auto simp: split-beta' Let-def*)

**lemma** *set-of-real-interval-subset*:  $\text{set-of } (\text{real-interval } x) \subseteq \text{set-of } (\text{real-interval } y)$   
**if**  $\text{set-of } x \subseteq \text{set-of } y$   
**using** *that*  
**by** (*auto simp: set-of-eq*)

**theorem** *tm-floatarith*:  
**assumes**  $t$ : *tm-floatarith prec ord I xs f = Some t*  
**assumes**  $a$ : *xs all-in I and x: x ∈<sub>τ</sub> I ! 0*



```

assumes xs-ne:  $xs \neq []$ 
assumes deriv:  $\bigwedge x. x \in_r I \Rightarrow isDERIV\ 0\ f\ (xs[0 := x])$ 
assumes  $\bigwedge i. 0 < i \Rightarrow i < length\ xs \Rightarrow e\ i = real-of-float\ (xs\ !\ i)$ 
assumes diff-e:  $(x - real-of-float\ (xs\ !\ 0)) = e\ 0$ 
shows interpret-floatarith  $f\ (xs[0:=x]) \in_i\ range-tm\ e\ t$ 
proof –
  from xs-ne a have I-ne[simp]:  $I \neq []$  by auto
  have xs'-in:  $xs[0 := x]\ all-in\ I$ 
    using a
    by (auto simp: nth-list-update x)
  from t obtain z where z: tmf-ivl-cs prec ord I xs f = Some z
    and tz: tm-poly t = fst (tmf-polys z)
    and tb: tm-bound t = round-interval prec (Ipoly (List.map2 (–) I xs) (snd
      (tmf-polys z)))
```

**using** *assms(1)*  
**by** (*cases t*) (*auto simp: those-eq-Some-iff split-beta' Let-def simp del: tmf-ivl-cs.simps*)  
**from** *tmf-ivl-cs-correct[OF a z(1)]*  
**have** *z-less*:  $i < ord \Rightarrow tmf-c\ prec\ (map\ interval-of\ xs)\ f\ i = Some\ (z\ !\ i)$   
**and** *lz*:  $length\ z = Suc\ ord\ length\ z - 1 = ord$   
**and** *z-ord*:  $tmf-c\ prec\ I\ f\ ord = Some\ (z\ !\ ord)$  **for** *i*  
**by** *auto*  
**have** *rewr*:  $\{..ord\} = insert\ ord\ \{..<ord\}$  **by** *auto*  
**let** *?diff* =  $\lambda(i::nat)\ (x::real). interpret-floatarith\ (deriv-rec\ f\ i)\ (xs[0:=x])$   
**let** *?c* = *real-of-float (xs ! 0)*  
**let** *?n* = *ord*  
**let** *?a* = *real-of-float (lower (I!0))*  
**let** *?b* = *real-of-float (upper (I!0))*  
**let** *?x* = *x::real*  
**let** *?f* =  $\lambda x::real. interpret-floatarith\ f\ (xs[0 := x])$   
**have** *2*: *?diff 0 = ?f* **using**  $\langle xs \neq [] \rangle$   
**by** (*simp add: map-update*)  
**have** *3*:  $\forall m\ t. m < ?n \wedge ?a \leq t \wedge t \leq ?b \longrightarrow (?diff\ m\ has-real-derivative\ ?diff$   
 (*Suc m*) *t*) (*at t*)  
**by** (*auto intro!: derivative-eq-intros deriv-rec-correct deriv simp: set-of-eq xs-ne*)  
**have** *4*:  $?a \leq ?c\ ?c \leq ?b\ ?a \leq ?x\ ?x \leq ?b$   
**using** *a xs-ne x*  
**by** (*force simp: set-of-eq*)  
**define** *cr* **where** *cr*  $\equiv \lambda s\ m. if\ m < ord\ then\ ?diff\ m\ ?c\ /\ fact\ m - mid\ (z\ !\ m)$   
                    $else\ ?diff\ m\ s\ /\ fact\ ord - mid\ (z\ !\ ord)$   
**define** *ci* **where** *ci*  $\equiv \lambda i. real-interval\ (z\ !\ i) - interval-of\ (real-of-float\ (mid\ (z$   
 $\ !\ i)))$   
  
**have** *cr-ord*:  $cr\ x\ ord \in_i\ ci\ ord$   
**using** *tmf-c-correct[OF xs'-in z-ord]*  
**by** (*auto simp: ci-def set-of-eq cr-def*)  
  
**have** *enclosure*:  $(\sum m < ord. cr\ s\ m * (x - (xs\ !\ 0)) \wedge m) + cr\ s\ ord * (x - (xs$   
 $\ !\ 0)) \wedge ord$

```

    ∈r round-interval prec (Ipoly (List.map2 (−) I (map interval-of xs)) (snd
(tmfm-polys z)))
  if cr-ord: cr s ord ∈i ci ord for s
proof −
  have (∑ m<ord. cr s m * (x − xs!0) ^ m) + cr s ord * (x − xs!0) ^ ord =
    horner-eval (cr s) (x − xs!0) (Suc ord)
  by (simp add: horner-eval-eq-setsum)
  also have ... ∈i horner-eval ci (real-interval (I ! 0 − xs ! 0)) (Suc ord)
proof (rule horner-eval-interval)
  fix i assume i < Suc ord
  then consider i < ord | i = ord by arith
  then show cr s i ∈i ci i
proof cases
  case 1
  then show ?thesis
    by (auto simp: cr-def ci-def not-less less-Suc-eq-le
      intro!: minus-in-intervalI tmf-c-correct[OF - z-less])
    (metis in-set-of-interval-of list-update-id map-update nth-map real-interval-of)
  qed (simp add: cr-ord)
  qed (auto intro!: minus-in-intervalI simp: real-interval-minus x)
  also have ... = set-of (horner-eval (real-interval o centered o (!) z)
    (real-interval (I ! 0 − xs ! 0)) (length z))
  by (auto simp: ci-def centered-def real-interval-minus real-interval-of lz)
  also have ... ⊆ set-of (Ipoly [real-interval (I ! 0 − xs ! 0)]
    (map-poly real-interval (snd (tmfm-polys z))))
  (is - ⊆ set-of ?x)
  by (rule Ipoly-snd-tmfm-polys)
  also have ... = set-of (real-interval (Ipoly [(I ! 0 − xs ! 0)] (snd (tmfm-polys
z))))
  by (auto simp: real-interval-Ipoly num-params-tmfm-polys2)
  also have ... ⊆ set-of (real-interval (round-interval prec (Ipoly [(I ! 0 − xs !
0)] (snd (tmfm-polys z)))))
  by (rule set-of-real-interval-subset) (rule round-ivl-correct)
  also
  have Ipoly [I ! 0 − interval-of (xs ! 0)] (snd (tmfm-polys z)) = Ipoly (List.map2
(−) I (map interval-of xs)) (snd (tmfm-polys z))
  using a
  apply (auto intro!: Ipoly-num-params-cong nth-equalityI
    simp: nth-Cons simp del:length-greater-0-conv split: nat.splits dest!:
less-le-trans[OF - num-params-tmfm-polys2[of z]])
  apply (subst map2-nth)
  by simp-all
  finally show ?thesis .
qed
consider 0 < ord x ≠ xs ! 0 | 0 < ord x = xs ! 0 | ord = 0 by arith
then show ?thesis
proof cases
  case hyps: 1
  then have 1: 0 < ord and 5: x ≠ xs ! 0 by simp-all

```

```

from Taylor[OF 1 2 3 4 5] obtain s where s: (if ?x < ?c then ?x < s ∧ s <
?c else ?c < s ∧ s < ?x)
  and tse: ?f ?x = (∑ m<?n. ?diff m ?c / fact m * (?x - ?c) ^ m) + ?diff ?n
s / fact ?n * (?x - ?c) ^ ?n
  by blast

have interpret-floatarith f ((map real-of-float xs)[0 := x]) -
Ipoly (List.map2 (-) [x] [xs!0]) (fst (tmf-polys z)) =
(∑ m<?n. ?diff m ?c / fact m * (?x - ?c) ^ m) + ?diff ?n s / fact ?n * (?x
- ?c) ^ ?n -
(∑ m≤?n. (x - xs!0) ^ m * mid (z ! m))
  unfolding tse
  by (simp add: Ipoly-fst-tmf-polys rewr lz)
also have ... = (∑ m<ord. cr s m * (x - xs!0) ^ m) + cr s ord * (x - xs!0)
^ ord
  unfolding rewr
  by (simp add: algebra-simps cr-def sum.distrib sum-subtractf)
also have cr s ord ∈i ci ord
  using a
  apply (auto simp: cr-def ci-def intro!: minus-in-intervalI
tmf-c-correct[OF - z-ord])
  by (smt 4(1) 4(2) 4(3) 4(4) a all-in-def in-real-intervalI length-greater-0-conv
nth-list-update s xs-ne)
note enclosure[OF this]
  also have Ipoly (List.map2 (-) [x] (map real-of-float [xs ! 0])) (map-poly
real-of-float (fst (tmf-polys z))) =
insertion e (map-poly real-of-float (fst (tmf-polys z)))
  using diff-e
  by (auto intro!: Ipoly-eq-insertionI simp: nth-Cons split: nat.splits dest:
less-le-trans[OF - num-params-tmf-polys1[of z]])
  finally
show ?thesis
  by (simp add: tz tb range-tm-def set-of-eq)
next
case 3
with 3 have length z = Suc 0 by (simp add: lz)
then have fst (tmf-polys z) = fst (tmf-polys [z ! 0])
  by (cases z) auto
also have ... = CN (mid (z ! 0))p 0 0p
  by simp
finally have fst (tmf-polys z) = CN (mid (z ! 0))p 0 0p .
with enclosure[OF cr-ord]
show ?thesis
  by (simp add: cr-def 3 range-tm-def tz tb set-of-eq)
next
case 2
have rewr: {..by (auto simp: lz)
from 2 enclosure[OF cr-ord]

```

```

    show ?thesis
  by (auto simp: zero-power 2 cr-def range-tm-def tz tb insertion-fst-tmf-polys
      diff-e[symmetric] rewr set-of-eq)
qed
qed

```

## 5.5 Operations on Taylor models

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

**fun** *tm-lower-order* *tm-lower-order-of-normed* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *taylor-model*  $\Rightarrow$  *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*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *taylor-model*  $\Rightarrow$  *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*  $\Rightarrow$  *nat*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *taylor-model*  $\Rightarrow$  *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 taylor models, also converts the polynomial into horner form.

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

**fun** *tm-add* :: *taylor-model*  $\Rightarrow$  *taylor-model*  $\Rightarrow$  *taylor-model*  
**where** *tm-add* (*TaylorModel* *p1* *e1*) (*TaylorModel* *p2* *e2*) = *TaylorModel* (*p1* +<sub>*p*</sub> *p2*) (*e1* + *e2*)

```

fun tm-sub :: taylor-model  $\Rightarrow$  taylor-model  $\Rightarrow$  taylor-model
  where tm-sub t1 t2 = tm-add t1 (tm-neg t2)

```

```

fun tm-mul :: nat  $\Rightarrow$  nat  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  taylor-model
 $\Rightarrow$  taylor-model  $\Rightarrow$  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  $\Rightarrow$  nat  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  taylor-model
 $\Rightarrow$  nat  $\Rightarrow$  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  $\Rightarrow$  nat  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  float
poly  $\Rightarrow$  taylor-model  $\Rightarrow$  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  $\Rightarrow$  taylor-model  $\Rightarrow$  taylor-model
  where tm-inc-err i (TaylorModel p e) = TaylorModel p (e + i)

```

```

fun tm-comp :: nat  $\Rightarrow$  nat  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  float  $\Rightarrow$ 
taylor-model  $\Rightarrow$  taylor-model  $\Rightarrow$  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  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  taylor-model  $\Rightarrow$ 
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  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  taylor-model  $\Rightarrow$ 
taylor-model  $\Rightarrow$  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  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  taylor-model  $\Rightarrow$ 
taylor-model  $\Rightarrow$  taylor-model
  where tm-min prec I a t1 t2 = tm-union prec I a t1 t2

```

```

fun tm-max :: nat  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  taylor-model  $\Rightarrow$ 
taylor-model  $\Rightarrow$  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
  apply (induction a b rule: polyadd.induct)
  apply (auto simp: algebra-simps Let-def)
  by (metis (no-types) mult-zero-right ring-class.ring-distrib(1))

```

```

lemma insertion-polyneg[simp]: insertion e ( $\sim_p$  b) = - insertion e b
  for b::'a::ring-1 poly

```

```

    by (induction b rule: polyneg.induct) (auto simp: algebra-simps Let-def)

lemma insertion-polysub[simp]: insertion e (a -p b) = insertion e a - insertion
e b
  for a b::'a::ring-1 poly
  by (simp add: polysub-def)

lemma insertion-polymul[simp]: insertion e (a *p b) = insertion e a * insertion e
b
  for a b::'a::comm-ring-1 poly
  by (induction a b rule: polymul.induct)
    (auto simp: algebra-simps Let-def)

lemma insertion-polypow[simp]: insertion e (a ^p b) = insertion e a ^ b
  for a::'a::comm-ring-1 poly
proof (induction b rule: nat-less-induct)
  case (1 n)
  then show ?case
  proof (cases n)
    case (Suc nat)
    then show ?thesis
    apply auto
    apply (auto simp: Let-def div2-less-self 1 simp del: polypow.simps)
    apply (metis even-Suc even-two-times-div-two odd-Suc-div-two semiring-normalization-rules(27)
semiring-normalization-rules(36))
    apply (metis even-two-times-div-two semiring-normalization-rules(36))
    done
  qed simp
qed

lemma insertion-polynate [simp]:
  insertion bs (polynate p) = (insertion bs p :: 'a::comm-ring-1)
  by (induct p rule: polynate.induct) auto

lemma tm-norm-poly-range:
  assumes x ∈i range-tm e t
  shows x ∈i range-tm e (tm-norm-poly t)
  using assms
  by (cases t) (simp add: range-tm-def)

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 -
  define xs where xs = map x [0..

```

```

have xs:  $i < \text{num-params } p \implies x\ i = xs\ !\ i$  for  $i$ 
  by (auto simp: xs-def)
have insertion  $x\ p = \text{Ipoly } xs\ p$ 
  by (auto intro!: insertion-eq-IpolyI xs)
also
from split-by-degree-correct[OF assms(1)[symmetric]]
have maxdegree  $l \leq \text{ord}$ 
  and  $p: \text{Ipoly } xs\ (\text{map-poly real-of-float } p) =$ 
     $\text{Ipoly } xs\ (\text{map-poly real-of-float } l) + \text{Ipoly } xs\ (\text{map-poly real-of-float } r)$ 
  and  $l: \text{num-params } l \leq \text{num-params } p$ 
  and  $r: \text{num-params } r \leq \text{num-params } p$ 
  by auto
show ?P1 ?P3 ?P4 by fact+
note p
also have  $\text{Ipoly } xs\ (\text{map-poly real-of-float } l) = \text{insertion } x\ l$ 
  using l
  by (auto intro!: xs Ipoly-eq-insertionI)
also have  $\text{Ipoly } xs\ (\text{map-poly real-of-float } r) = \text{insertion } x\ r$ 
  using r
  by (auto intro!: xs Ipoly-eq-insertionI)
finally show ?P2 .
qed

```

```

lemma split-by-prec-correct-insertion:
  fixes  $x :: \text{nat} \Rightarrow \text{real}$  and  $p :: \text{float poly}$ 
  assumes split-by-prec  $\text{ord } p = (l, r)$ 
  shows  $\text{insertion } x\ p = \text{insertion } x\ l + \text{insertion } x\ r$  (is ?P1)
    and  $\text{num-params } l \leq \text{num-params } p$  (is ?P2)
    and  $\text{num-params } r \leq \text{num-params } p$  (is ?P3)
proof -
  define xs where  $xs = \text{map } x\ [0..<\text{num-params } p]$ 
  have xs:  $i < \text{num-params } p \implies x\ i = xs\ !\ i$  for  $i$ 
    by (auto simp: xs-def)
  have insertion  $x\ p = \text{Ipoly } xs\ p$ 
    by (auto intro!: insertion-eq-IpolyI xs)
  also
  from split-by-prec-correct[OF assms(1)[symmetric]]
  have  $p: \text{Ipoly } xs\ (\text{map-poly real-of-float } p) =$ 
     $\text{Ipoly } xs\ (\text{map-poly real-of-float } l) + \text{Ipoly } xs\ (\text{map-poly real-of-float } r)$ 
    and  $l: \text{num-params } l \leq \text{num-params } p$ 
    and  $r: \text{num-params } r \leq \text{num-params } p$ 
    by auto
  show ?P2 ?P3 by fact+
  note p
  also have  $\text{Ipoly } xs\ (\text{map-poly real-of-float } l) = \text{insertion } x\ l$ 
    using l
    by (auto intro!: xs Ipoly-eq-insertionI)
  also have  $\text{Ipoly } xs\ (\text{map-poly real-of-float } r) = \text{insertion } x\ r$ 
    using r

```



```

    by (auto intro!: xs Ipoly-eq-insertionI)
    finally show ?P1 .
qed

lemma tm-lower-order-of-normed-range:
  assumes  $x \in_i \text{range-tm } e \ t$ 
  assumes  $\text{dev: develops-at-within } e \ a \ I$ 
  assumes  $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$ 
  shows  $x \in_i \text{range-tm } e \ (\text{tm-lower-order-of-normed } \text{prec } \text{ord } I \ a \ t)$ 
proof -
  obtain  $p \ \text{err}$  where  $t\text{-decomp: } t = \text{TaylorModel } p \ \text{err}$ 
  by (cases  $t$ ) simp
  obtain  $pl \ pr$  where  $p\text{-split: } \text{split-by-degree } \text{ord } p = (pl, pr)$ 
  by (cases  $\text{split-by-degree } \text{ord } p$ , simp)

  from  $\text{split-by-degree-correct-insertion}[OF \ p\text{-split}]$ 
  have  $\text{params: } \text{maxdegree } pl \leq \text{ord } \text{num-params } pl \leq \text{num-params } p \ \text{num-params}$ 
   $pr \leq \text{num-params } p$ 
  and  $\text{ins: } \text{insertion } e \ (\text{map-poly } \text{real-of-float } p) =$ 
   $\text{insertion } e \ (\text{map-poly } \text{real-of-float } pl) + \text{insertion } e \ (\text{map-poly } \text{real-of-float } pr)$ 
  by auto
  from  $\text{assms } \text{params}$  have  $\text{params-pr: } \text{num-params } pr \leq \text{length } I$  by (auto simp:  $t\text{-decomp}$ )

  have  $\text{range-tm } e \ t =$ 
   $\text{interval-of } (\text{insertion } e \ (\text{map-poly } \text{real-of-float } pl)) +$ 
   $(\text{interval-of } (\text{insertion } e \ (\text{map-poly } \text{real-of-float } pr)) + \text{real-interval } \text{err})$ 
  by (auto simp:  $t\text{-decomp } \text{range-tm-def } \text{ins } \text{ac-simps } \text{interval-of-plus}$ ) term
  round-interval
  also have  $\text{set-of } \dots \subseteq \text{set-of } (\text{interval-of } (\text{insertion } e \ pl)) +$ 
   $\text{set-of } (\text{real-interval } (\text{round-interval } \text{prec } (\text{err} + \text{compute-bound-poly } \text{prec } pr \ I$ 
   $a)))$ 
  unfolding  $\text{set-of-plus } \text{real-interval-plus } \text{add.commute}[of \ \text{err}]$ 
  apply (rule  $\text{set-plus-mono2}[OF \ \text{order-refl}]$ )
  apply (rule  $\text{order-trans}$ ) prefer 2
  apply (rule  $\text{set-of-real-interval-subset}$ )
  apply (rule  $\text{round-ivl-correct}$ )
  unfolding  $\text{set-of-plus } \text{real-interval-plus}$ 
  apply (rule  $\text{set-plus-mono2}[OF \ - \ \text{order-refl}]$ )
  apply (rule  $\text{subsetI}$ )
  apply simp
  apply (rule  $\text{compute-bound-poly-correct}$ )
  apply (rule  $\text{params-pr}$ )
  by (rule  $\text{assms}$ )
  also have  $\dots = \text{set-of } (\text{range-tm } e \ (\text{tm-lower-order-of-normed } \text{prec } \text{ord } I \ a \ t))$ 
  by (simp add:  $t\text{-decomp } \text{split-beta}' \ \text{Let-def } p\text{-split } \text{range-tm-def } \text{set-of-plus}$ )
  finally show ?thesis using  $\text{assms}$  by auto
qed

```

**lemma** *num-params-tm-norm-poly-le*:  $\text{num-params } (\text{tm-poly } (\text{tm-norm-poly } t)) \leq X$

**if**  $\text{num-params } (\text{tm-poly } t) \leq X$

**using** *that*

**by** (*cases*  $t$ ) (*auto simp*: *intro!*:  $\text{num-params-polynate}[THEN \text{order-trans}]$ )

**lemma** *tm-lower-order-range*:

**assumes**  $x \in_i \text{range-tm } e \ t$

**assumes** *dev*: *develops-at-within*  $e \ a \ I$

**assumes**  $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$

**shows**  $x \in_i \text{range-tm } e \ (\text{tm-lower-order } \text{prec } \text{ord } I \ a \ t)$

**by** (*auto simp add*: *intro!*: *tm-lower-order-of-normed-range* *tm-norm-poly-range* *assms*

*num-params-tm-norm-poly-le*)

**lemma** *tm-round-floats-of-normed-range*:

**assumes**  $x \in_i \text{range-tm } e \ t$

**assumes** *dev*: *develops-at-within*  $e \ a \ I$

**assumes**  $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$

**shows**  $x \in_i \text{range-tm } e \ (\text{tm-round-floats-of-normed } \text{prec } I \ a \ t)$

— TODO: this is a clone of  $\llbracket ?x \in_i \text{range-tm } ?e \ ?t; \text{develops-at-within } ?e \ ?a \ ?I; \text{num-params } (\text{tm-poly } ?t) \leq \text{length } ?I \rrbracket \implies ?x \in_i \text{range-tm } ?e \ (\text{tm-lower-order-of-normed } ?\text{prec } ?\text{ord } ?I \ ?a \ ?t)$  -> general sweeping method!

**proof**—

**obtain**  $p \ \text{err}$  **where** *t-decomp*:  $t = \text{TaylorModel } p \ \text{err}$

**by** (*cases*  $t$ ) *simp*

**obtain**  $pl \ pr$  **where** *p-prec*: *split-by-prec*  $\text{prec } p = (pl, pr)$

**by** (*cases* *split-by-prec*  $\text{prec } p$ , *simp*)

**from** *split-by-prec-correct-insertion*[*OF* *p-prec*]

**have** *params*:  $\text{num-params } pl \leq \text{num-params } p \ \text{num-params } pr \leq \text{num-params } p$

**and** *ins*: *insertion*  $e \ (\text{map-poly } \text{real-of-float } p) =$

*insertion*  $e \ (\text{map-poly } \text{real-of-float } pl) + \text{insertion } e \ (\text{map-poly } \text{real-of-float } pr)$

**by** *auto*

**from** *assms* *params* **have** *params-pr*:  $\text{num-params } pr \leq \text{length } I$

**by** (*auto simp*: *t-decomp*)

**have** *range-tm*  $e \ t =$

*interval-of* (*insertion*  $e \ (\text{map-poly } \text{real-of-float } pl)) +$

*interval-of* (*insertion*  $e \ (\text{map-poly } \text{real-of-float } pr)) + \text{real-interval } \text{err}$ )

**by** (*auto simp*: *t-decomp* *range-tm-def* *ins* *ac-simps* *interval-of-plus*)

**also have** *set-of*  $\dots \subseteq \text{set-of } (\text{interval-of } (\text{insertion } e \ pl)) +$

*set-of* (*real-interval* (*round-interval*  $\text{prec } (\text{err} + \text{compute-bound-poly } \text{prec } pr \ I \ a)))$ )

**unfolding** *set-of-plus* *real-interval-plus* *add commute*[*of* *err*]

**apply** (*rule* *set-plus-mono2*[*OF* *order-refl*])

**apply** (*rule* *order-trans*) **prefer** 2

**apply** (*rule* *set-of-real-interval-subset*)

**apply** (*rule* *round-ivl-correct*)

**unfolding** *set-of-plus real-interval-plus*  
**apply** (*rule set-plus-mono2*[*OF* - *order-refl*])  
**apply** (*rule subsetI*)  
**apply** *simp*  
**apply** (*rule compute-bound-poly-correct*)  
**apply** (*rule params-pr*)  
**by** (*rule assms*)  
**also have**  $\dots = \text{set-of } (\text{range-tm } e \text{ (tm-round-floats-of-normed prec } I \text{ a } t))$   
**by** (*simp add: t-decomp split-beta' Let-def p-prec range-tm-def set-of-plus*)  
**finally show** *?thesis* **using** *assms* **by** *auto*  
**qed**

**lemma** *num-params-split-by-degree-le*:  $\text{num-params } (\text{fst } (\text{split-by-degree ord } x)) \leq K$   
 $\text{num-params } (\text{snd } (\text{split-by-degree ord } x)) \leq K$   
**if**  $\text{num-params } x \leq K$  **for**  $x::\text{float poly}$   
**using** *split-by-degree-correct-insertion*(3,4)[*of ord x, OF surjective-pairing*] **that**  
**by** *auto*

**lemma** *num-params-split-by-prec-le*:  $\text{num-params } (\text{fst } (\text{split-by-prec ord } x)) \leq K$   
 $\text{num-params } (\text{snd } (\text{split-by-prec ord } x)) \leq K$   
**if**  $\text{num-params } x \leq K$  **for**  $x::\text{float poly}$   
**using** *split-by-prec-correct-insertion*(2,3)[*of ord x, OF surjective-pairing*] **that**  
**by** *auto*

**lemma** *num-params-tm-norm'-le*:  
 $\text{num-params } (\text{tm-poly } (\text{tm-round-floats-of-normed prec } I \text{ a } t)) \leq X$   
**if**  $\text{num-params } (\text{tm-poly } t) \leq X$   
**using** *that*  
**by** (*cases t*) (*auto simp: tm-norm'.simps split-beta' Let-def intro!: num-params-split-by-prec-le*)

**lemma** *tm-round-floats-range*:  
**assumes**  $x \in_i \text{range-tm } e \text{ t develops-at-within } e \text{ a } I \text{ num-params } (\text{tm-poly } t) \leq$   
 $\text{length } I$   
**shows**  $x \in_i \text{range-tm } e \text{ (tm-round-floats prec } I \text{ a } t)$   
**by** (*auto intro!: tm-round-floats-of-normed-range assms tm-norm-poly-range num-params-tm-norm-poly-le*)

**lemma** *num-params-tm-lower-order-of-normed-le*:  $\text{num-params } (\text{tm-poly } (\text{tm-lower-order-of-normed}$   
 $\text{prec ord } I \text{ a } t)) \leq X$   
**if**  $\text{num-params } (\text{tm-poly } t) \leq X$   
**using** *that*  
**apply** (*cases t*)  
**apply** (*auto simp: split-beta' Let-def intro!: num-params-polynate[THEN or-*  
 $\text{der-trans}]$ )  
**apply** (*rule order-trans[OF split-by-degree-correct(3)]*)  
**by** (*auto simp: prod-eq-iff*)

**lemma** *tm-norm'-range*:

**assumes**  $x \in_i \text{range-tm } e \text{ } t \text{ develops-at-within } e \text{ } a \text{ } I \text{ num-params } (tm\text{-poly } t) \leq$   
*length I*  
**shows**  $x \in_i \text{range-tm } e \text{ } (tm\text{-norm}' \text{ prec ord } I \text{ } a \text{ } t)$   
**by** (*auto intro!*: *tm-round-floats-of-normed-range tm-lower-order-of-normed-range*  
*assms*  
*num-params-tm-norm-poly-le num-params-tm-lower-order-of-normed-le*  
*simp: tm-norm'.simps*)

**lemma** *num-params-tm-norm'*:  
 $\text{num-params } (tm\text{-poly } (tm\text{-norm}' \text{ prec ord } I \text{ } a \text{ } t)) \leq X$   
**if**  $\text{num-params } (tm\text{-poly } t) \leq X$   
**using** *that*  
**by** (*cases t*) (*auto simp: tm-norm'.simps split-beta' Let-def*  
*intro!*: *num-params-tm-norm'-le num-params-split-by-prec-le num-params-split-by-degree-le*)

**lemma** *tm-norm-range*:  
**assumes**  $x \in_i \text{range-tm } e \text{ } t \text{ develops-at-within } e \text{ } a \text{ } I \text{ num-params } (tm\text{-poly } t) \leq$   
*length I*  
**shows**  $x \in_i \text{range-tm } e \text{ } (tm\text{-norm prec ord } I \text{ } a \text{ } t)$   
**by** (*auto intro!*: *assms tm-norm'-range tm-norm-poly-range num-params-tm-norm-poly-le*)

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

**lemma** *tm-neg-range*:  
**assumes**  $x \in_i \text{range-tm } e \text{ } t$   
**shows**  $-x \in_i \text{range-tm } e \text{ } (tm\text{-neg } t)$   
**using** *assms*  
**by** (*cases t*)  
*(auto simp: set-of-eq range-tm-def interval-of-plus interval-of-uminus map-poly-homo-polyneg)*

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

**lemma** *tm-bound-tm-add*[*simp*]:  $tm\text{-bound } (tm\text{-add } t1 \text{ } t2) = tm\text{-bound } t1 + tm\text{-bound } t2$   
**by** (*cases t1; cases t2*) *auto*

**lemma** *interval-of-add*:  $\text{interval-of } (a + b) = \text{interval-of } a + \text{interval-of } b$   
**by** (*auto intro!*: *interval-eqI*)

**lemma** *tm-add-range*:  
 $x + y \in_i \text{range-tm } e \text{ } (tm\text{-add } t1 \text{ } t2)$   
**if**  $x \in_i \text{range-tm } e \text{ } t1$   
 $y \in_i \text{range-tm } e \text{ } t2$   
**proof** –  
**from** *range-tmD[OF that(1)] range-tmD[OF that(2)]*  
**show** *?thesis*  
**apply** (*cases t1; cases t2*)  
**apply** (*rule range-tmI*)  
**by** (*auto simp: map-poly-homo-polyadd real-interval-plus ac-simps interval-of-add*  
*num-params-polyadd insertion-polyadd set-of-eq*)

```

dest: less-le-trans[OF - num-params-polyadd])
qed
lemmas [simp del] = tm-add.simps

lemma tm-sub-range:
  assumes  $x \in_i \text{range-tm } e \ t1$ 
  assumes  $y \in_i \text{range-tm } e \ t2$ 
  shows  $x - y \in_i \text{range-tm } e \ (tm-sub \ t1 \ t2)$ 
  using tm-add-range[OF assms(1) tm-neg-range[OF assms(2)]]
  by simp
lemmas [simp del] = tm-sub.simps

lemma set-of-intervalI:  $\text{set-of } (\text{interval-of } y) \subseteq \text{set-of } Y$  if  $y \in_i Y$  for  $y::'a::order$ 
  using that by (auto simp: set-of-eq)

lemma set-of-real-intervalI:  $\text{set-of } (\text{interval-of } y) \subseteq \text{set-of } (\text{real-interval } Y)$  if  $y \in_r Y$ 
  using that by (auto simp: set-of-eq)

lemma tm-mul-range:
  assumes  $x \in_i \text{range-tm } e \ t1$ 
  assumes  $y \in_i \text{range-tm } e \ t2$ 
  assumes dev: develops-at-within  $e \ a \ I$ 
  assumes params:  $\text{num-params } (tm-poly \ t1) \leq \text{length } I \ \text{num-params } (tm-poly \ t2)$ 
  ≤ length  $I$ 
  shows  $x * y \in_i \text{range-tm } e \ (tm-mul \ \text{prec } \text{ord } I \ a \ t1 \ t2)$ 
proof -
  define p1 where  $p1 = tm-poly \ t1$ 
  define p2 where  $p2 = tm-poly \ t2$ 
  define e1 where  $e1 = tm-bound \ t1$ 
  define e2 where  $e2 = tm-bound \ t2$ 
  have t1-def:  $t1 = \text{TaylorModel } p1 \ e1$  and t2-def:  $t2 = \text{TaylorModel } p2 \ e2$ 
  by (auto simp: p1-def e1-def p2-def e2-def)
  from params have params:  $\text{num-params } p1 \leq \text{length } I \ \text{num-params } p2 \leq \text{length } I$ 
  by (auto simp: p1-def p2-def)
  from range-tmD[OF assms(1)]
  obtain xe where  $x = \text{insertion } e \ p1 + xe$ 
  (is - = ?x' + -)
  and xe:  $xe \in_r e1$ 
  by (auto simp: p1-def e1-def elim!: plus-in-intervalE)
  from range-tmD[OF assms(2)]
  obtain ye where  $y = \text{insertion } e \ p2 + ye$ 
  (is - = ?y' + -)
  and ye:  $ye \in_r e2$ 
  by (auto simp: p2-def e2-def elim!: plus-in-intervalE)
  have  $x * y = \text{insertion } e \ (p1 *_p p2) + (xe * ?y' + ?x' * ye + xe * ye)$ 
  by (simp add: algebra-simps x y map-poly-homo-polymul)
  also have ...  $\in_i \text{range-tm } e \ (tm-mul \ \text{prec } \text{ord } I \ a \ t1 \ t2)$ 

```

```

  by (auto intro!: tm-round-floats-of-normed-range assms tm-norm'-range
      simp: split-beta' Let-def t1-def t2-def)
  (auto simp: range-tm-def real-interval-plus real-interval-times intro!: plus-in-intervalI
      times-in-intervalI xe ye params compute-bound-poly-correct dev
      num-params-polymul[THEN order-trans])
  finally show ?thesis .
qed

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
  using that
  by (cases t1; cases t2)
    (auto simp: intro!: num-params-tm-norm' num-params-polymul[THEN or-
der-trans])

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)
  unfolding atomize-conj atomize-imp
proof(induction n arbitrary: x t rule: nat-less-induct)
  case (1 n)
  note IH1 = 1(1)[rule-format, THEN conjunct1, rule-format]
  note IH2 = 1(1)[rule-format, THEN conjunct2, THEN order-trans]
  show ?case
  proof (cases n)
    case 0
    then show ?thesis by (auto simp: tm-const-def range-tm-def set-of-eq tm-pow.simps)
  next
    case (Suc nat)
    have eq: odd nat ⇒ x * x ^ nat = x ^ ((Suc nat) div 2) * x ^ ((Suc nat) div
2)
    apply (subst power-add[symmetric])
    unfolding div2-plus-div2
    by simp
  show ?thesis
  unfolding tm-pow.simps Suc
  using Suc
  apply (auto )
  subgoal
    apply (rule tm-mul-range) apply (assumption)
    apply (rule IH1) apply force

```

```

      apply assumption+
      apply (rule IH2) apply force
      apply assumption
    done
  subgoal
    apply (rule num-params-tm-mul-le) apply force
    apply (rule IH2) apply force
    apply force
    done
  subgoal
    apply (auto simp: Let-def)
    unfolding eq odd-Suc-div-two
    apply (rule tm-mul-range)
    subgoal by (rule IH1) (auto intro!: tm-mul-range num-params-tm-mul-le
IH1 IH2 1
      simp: Let-def div2-less-self)
    subgoal by (rule IH1) (auto intro!: tm-mul-range num-params-tm-mul-le
IH1 IH2 1
      simp: Let-def div2-less-self)
    subgoal by assumption
    subgoal by (rule IH2) (auto simp: div2-less-self 1)
    subgoal by (rule IH2) (auto simp: div2-less-self 1)
    done
  subgoal
    by (auto simp: Let-def div2-less-self 1 intro!: IH2 num-params-tm-mul-le)
  done
qed
qed

```

```

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
using that
by (cases t1; cases t2)
  (auto simp: tm-add.simps
    intro!: num-params-tm-norm' num-params-polymul[THEN order-trans]
    num-params-polyadd[THEN order-trans])

```

```

lemma num-params-tm-neg-eq[simp]:
  num-params (tm-poly (tm-neg t1)) = num-params (tm-poly t1)
by (cases t1) (auto simp: tm-neg.simps num-params-polyneg)

```

```

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
using that
by (cases t1; cases t2) (auto simp: tm-sub.simps intro!: num-params-tm-add-le)

```

**lemma** *num-params-eval-poly-le*: *num-params* (*tm-poly* (*eval-poly-at-tm* *prec* *ord* *I* *a* *p* *t*))  $\leq x$   
**if** *num-params* (*tm-poly* *t*)  $\leq x$  *num-params* *p*  $\leq \max 1 x$   
**using** *that*  
**by** (*induction* *prec* *ord* *I* *a* *p* *t* *rule*: *eval-poly-at-tm.induct*)  
(*auto intro!*: *num-params-tm-add-le* *num-params-tm-sub-le* *num-params-tm-mul-le*  
*num-params-tm-pow-le*)

**lemma** *eval-poly-at-tm-range*:  
**assumes** *num-params* *p*  $\leq 1$   
**assumes** *tg-def*:  $e' 0 \in_i \text{range-tm } e \text{ } tg$   
**assumes** *dev*: *develops-at-within* *e* *a* *I* **and** *params*: *num-params* (*tm-poly* *tg*)  $\leq$   
*length* *I*  
**shows** *insertion*  $e' p \in_i \text{range-tm } e \text{ } (\text{eval-poly-at-tm } \text{prec } \text{ord } I \text{ } a \text{ } p \text{ } tg)$   
**using** *assms*(1) *params*  
**proof**(*induction* *p*)  
**case** (*C* *c*) **thus** ?*case*  
**using** *tg-def*  
**by** (*cases* *tg*) (*auto simp*: *tm-const-def* *range-tm-def* *real-interval-zero*)  
**next**  
**case** (*Bound* *n*) **thus** ?*case*  
**using** *tg-def*  
**by** *simp*  
**next**  
**case** (*Add* *p1l* *p1r*) **thus** ?*case*  
**using** *tm-add-range* **by** (*simp add*: *func-plus*)  
**next**  
**case** (*Sub* *p1l* *p1r*) **thus** ?*case*  
**using** *tm-sub-range* **by** (*simp add*: *fun-diff-def*)  
**next**  
**case** (*Mul* *p1l* *p1r*) **thus** ?*case*  
**by** (*auto intro!*: *tm-mul-range* *Mul* *dev* *num-params-eval-poly-le*)  
**next**  
**case** (*Neg* *p1'*) **thus** ?*case*  
**using** *tm-neg-range* **by** (*simp add*: *fun-Compl-def*)  
**next**  
**case** (*Pw* *p1'* *n*) **thus** ?*case*  
**by** (*auto intro!*: *tm-pow-range* *Pw* *dev* *num-params-eval-poly-le*)  
**next**  
**case** (*CN* *p1l* *n* *p1r*) **thus** ?*case*  
**by** (*auto intro!*: *tm-mul-range* *tm-pow-range* *CN* *dev* *num-params-eval-poly-le*  
*tm-add-range* *tg-def*)  
**qed**

**lemma** *tm-inc-err-range*:  $x \in_i \text{range-tm } e \text{ } (\text{tm-inc-err } i \text{ } t)$   
**if**  $x \in_i \text{range-tm } e \text{ } t + \text{real-interval } i$   
**using** *that*  
**by** (*cases* *t*) (*auto simp*: *range-tm-def* *real-interval-plus* *ac-simps*)



**lemma** *num-params-tm-inc-err*:  $\text{num-params } (\text{tm-poly } (\text{tm-inc-err } i \ t)) \leq X$   
**if**  $\text{num-params } (\text{tm-poly } t) \leq X$   
**using** *that*  
**by** (*cases t*) *auto*

**lemma** *num-params-tm-comp-le*:  $\text{num-params } (\text{tm-poly } (\text{tm-comp } \text{prec } \text{ord } I \ a \ ga \ tf \ tg)) \leq X$   
**if**  $\text{num-params } (\text{tm-poly } tf) \leq \max 1 \ X \ \text{num-params } (\text{tm-poly } tg) \leq X$   
**using** *that*  
**by** (*cases tf*) (*auto intro!*: *num-params-tm-inc-err num-params-eval-poly-le num-params-tm-sub-le*)

**lemma** *tm-comp-range*:  
**assumes** *tf-def*:  $x \in_i \text{range-tm } e' \ tf$   
**assumes** *tg-def*:  $e' \ 0 \in_i \text{range-tm } e \ (\text{tm-sub } tg \ (\text{tm-const } ga))$   
**assumes** *params*:  $\text{num-params } (\text{tm-poly } tf) \leq 1 \ \text{num-params } (\text{tm-poly } tg) \leq \text{length } I$   
**assumes** *dev*: *develops-at-within*  $e \ a \ I$   
**shows**  $x \in_i \text{range-tm } e \ (\text{tm-comp } \text{prec } \text{ord } I \ a \ ga \ tf \ tg)$   
**proof**–  
**obtain** *pf ef* **where** *tf-decomp*:  $tf = \text{TaylorModel } pf \ ef$  **using** *taylor-model.exhaust*  
**by** *auto*  
**obtain** *pg eg* **where** *tg-decomp*:  $tg = \text{TaylorModel } pg \ eg$  **using** *taylor-model.exhaust*  
**by** *auto*

**from** *params* **have** *params*:  $\text{num-params } pf \leq \text{Suc } 0 \ \text{num-params } pg \leq \text{length } I$   
**by** (*auto simp: tf-decomp tg-decomp*)  
**from** *tf-def* **obtain** *xe* **where** *x-def*:  $x = \text{insertion } e' \ pf + xe \ xe \in_r \ ef$   
**by** (*auto simp: tf-decomp range-tm-def elim!: plus-in-intervalE*)  
**show** *?thesis*  
**using** *tg-def*  
**by** (*auto simp: tf-decomp tg-decomp x-def params dev*  
*intro!: tm-inc-err-range eval-poly-at-tm-range plus-in-intervalI num-params-tm-sub-le*)  
**qed**

**lemma** *mid-centered-collapse*:  
 $\text{interval-of } (\text{real-of-float } (\text{mid } \text{abs-bound})) + \text{real-interval } (\text{centered } \text{abs-bound}) =$   
 $\text{real-interval } \text{abs-bound}$   
**by** (*auto simp: centered-def interval-eq-iff*)

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

**lemma** *tm-abs-range*:  
**assumes** *x*:  $x \in_i \text{range-tm } e \ t$   
**assumes** *n*:  $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$  **and** *d*: *develops-at-within*  $e \ a \ I$   
**shows**  $\text{abs } x \in_i \text{range-tm } e \ (\text{tm-abs } \text{prec } I \ a \ t)$   
**proof**–  
**obtain** *p e* **where** *t-def[simp]*:  $t = \text{TaylorModel } p \ e$  **using** *taylor-model.exhaust*  
**by** *auto*  
**define** *bound* **where**  $\text{bound} = \text{compute-bound-tm } \text{prec } I \ a \ t$

```

have bound:  $x \in_r$  bound
  unfolding bound-def
  using  $n \ d \ x$ 
  by (rule compute-bound-tm-correct)
define abs-bound where abs-bound  $\equiv$   $Ivl \ 0 \ (\max \ |lower \ bound| \ |upper \ bound|)$ 
have abs-bound:  $|x| \in_r$  abs-bound using bound
  by (auto simp: abs-bound-def set-of-eq abs-real-def max-def min-def)
have tm-abs-decomp: tm-abs prec  $I \ a \ t = TaylorModel \ (poly.C \ (mid \ abs-bound))$ 
(centered abs-bound)
  by (simp add: bound-def abs-bound-def Let-def tm-abs.simps)
show ?thesis
  unfolding tm-abs-decomp
  by (rule range-tmI) (auto simp: mid-centered-collapse abs-bound)
qed

```

```

lemma num-params-tm-abs-le: num-params (tm-poly (tm-abs prec  $I \ a \ t$ ))  $\leq X$  if
num-params (tm-poly  $t$ )  $\leq X$ 
  using that
  by (auto simp: tm-abs.simps Let-def)

```

```

lemma real-interval-sup: real-interval (sup  $a \ b$ ) = sup (real-interval  $a$ ) (real-interval
b)
  by (auto simp: interval-eq-iff inf-real-def inf-float-def sup-float-def sup-real-def
min-def max-def)

```

```

lemma in-interval-supI1:  $x \in_i \ a \implies x \in_i \ sup \ a \ b$ 
and in-interval-supI2:  $x \in_i \ b \implies x \in_i \ sup \ a \ b$ 
for  $x::'a::lattice$ 
by (auto simp: set-of-eq le-infI1 le-infI2 le-supI1 le-supI2)

```

```

lemma tm-union-range-left:
  assumes  $x \in_i \ range\text{-}tm \ e \ t1$ 
  num-params (tm-poly  $t1$ )  $\leq length \ I \ develops\text{-}at\text{-}within \ e \ a \ I$ 
  shows  $x \in_i \ range\text{-}tm \ e \ (tm\text{-}union \ prec \ I \ a \ t1 \ t2)$ 
proof–
  define  $b1$  where  $b1 \equiv compute\text{-}bound\text{-}tm \ prec \ I \ a \ t1$ 
  define  $b2$  where  $b2 \equiv compute\text{-}bound\text{-}tm \ prec \ I \ a \ t2$ 
  define  $b\text{-}combined$  where  $b\text{-}combined \equiv sup \ b1 \ b2$ 

```

```

obtain  $p \ e$  where tm-union-decomp: tm-union prec  $I \ a \ t1 \ t2 = TaylorModel \ p \ e$ 
  using taylor-model.exhaust by auto
then have  $p\text{-}def$ :  $p = (mid \ b\text{-}combined)_p$ 
  and  $e\text{-}def$ :  $e = centered \ b\text{-}combined$ 
  by (auto simp: Let-def b1-def b2-def b-combined-def interval-eq-iff)
have  $x \in_r \ b1$ 
  by (auto simp: b1-def intro!: compute-bound-tm-correct assms)
then have  $x \in_r \ b\text{-}combined$ 
  by (auto simp: b-combined-def real-interval-sup in-interval-supI1)
then show ?thesis

```

**unfolding** *tm-union-decomp*  
**by** (*auto simp: range-tm-def p-def e-def mid-centered-collapse*)  
**qed**

**lemma** *tm-union-range-right*:  
**assumes**  $x \in_i \text{range-tm } e \ t2$   
 $\text{num-params } (tm\text{-poly } t2) \leq \text{length } I \text{ develops-at-within } e \ a \ I$   
**shows**  $x \in_i \text{range-tm } e \ (tm\text{-union } prec \ I \ a \ t1 \ t2)$   
**using** *tm-union-range-left[OF assms]*  
**by** (*simp add: interval-union-commute*)

**lemma** *num-params-tm-union-le*:  
 $\text{num-params } (tm\text{-poly } (tm\text{-union } prec \ I \ a \ t1 \ t2)) \leq X$   
**if**  $\text{num-params } (tm\text{-poly } t1) \leq X \ \text{num-params } (tm\text{-poly } t2) \leq X$   
**using** *that*  
**by** (*auto simp: Let-def*)

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

**lemma** *tm-min-range*:  
**assumes**  $x \in_i \text{range-tm } e \ t1$   
**assumes**  $y \in_i \text{range-tm } e \ t2$   
 $\text{num-params } (tm\text{-poly } t1) \leq \text{length } I$   
 $\text{num-params } (tm\text{-poly } t2) \leq \text{length } I$   
 $\text{develops-at-within } e \ a \ I$   
**shows**  $\min \ x \ y \in_i \text{range-tm } e \ (tm\text{-min } prec \ I \ a \ t1 \ t2)$   
**using** *assms*  
**by** (*auto simp: Let-def tm-min.simps min-def intro: tm-union-range-left tm-union-range-right*)

**lemma** *tm-max-range*:  
**assumes**  $x \in_i \text{range-tm } e \ t1$   
**assumes**  $y \in_i \text{range-tm } e \ t2$   
 $\text{num-params } (tm\text{-poly } t1) \leq \text{length } I$   
 $\text{num-params } (tm\text{-poly } t2) \leq \text{length } I$   
 $\text{develops-at-within } e \ a \ I$   
**shows**  $\max \ x \ y \in_i \text{range-tm } e \ (tm\text{-max } prec \ I \ a \ t1 \ t2)$   
**using** *assms*  
**by** (*auto simp: Let-def tm-max.simps max-def intro: tm-union-range-left tm-union-range-right*)

## 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* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{float interval list} \Rightarrow \text{float interval list} \Rightarrow$   
 $\text{floatarith} \Rightarrow \text{taylor-model option} \Rightarrow (\text{float interval} \Rightarrow \text{bool}) \Rightarrow \text{taylor-model option}$   
**where** *compute-tm-by-comp* *prec ord I a f g f-exists-on* = (

```

    case g
    of Some tg  $\Rightarrow$  (
      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 ( $\lambda$ tf. tm-comp prec ord I a ga tf tg) (tm-floatarith prec
ord [gI] [ga] f)
      else None)
    | -  $\Rightarrow$  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  $\Rightarrow$  nat  $\Rightarrow$  float interval list  $\Rightarrow$  float interval list  $\Rightarrow$  floatarith
 $\Rightarrow$  taylor-model list  $\Rightarrow$ 
  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)  $\Rightarrow$  Some (tm-add t1 t2)
  | -  $\Rightarrow$  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)  $\Rightarrow$  Some (tm-mul prec ord I a t1 t2)
  | -  $\Rightarrow$  None)
| approx-tm prec ord I a (Power f k) env
  = map-option ( $\lambda$ t. 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) ( $\lambda$ x. 0 < lower x  $\vee$  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) ( $\lambda$ x. 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) ( $\lambda$ x. 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) ( $\lambda$ x. 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) ( $\lambda$ x. 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*:  $\text{mid } i \in_r i$   
**using** *lower-le-upper*[*of i*]  
**by** (*auto simp: mid-def set-of-eq powr-minus*)

**lemma** *set-of-real-interval-mono*:  $\text{set-of } (\text{real-interval } x) \subseteq \text{set-of } (\text{real-interval } y)$   
**if**  $\text{set-of } x \subseteq \text{set-of } y$   
**using** *that* **by** (*auto simp: set-of-eq*)

**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*:  
 $\text{tm-floatarith prec ord a b f} = \text{Some tf} \implies \text{num-params } (\text{tm-poly tf}) \leq 1$   
**by** (*auto simp: tm-floatarith.simps split-beta' Let-def those-eq-Some-iff num-params-tmf-polys1*)

**lemma** *compute-tm-by-comp-range*:  
**assumes**  $\text{max-Var-floatarith } f \leq 1$   
**assumes**  $a: a \text{ all-subset } I$   
**assumes**  $\text{tx-range}: x \in_i \text{range-tm } e \text{ tg}$   
**assumes**  $\text{t-def}: \text{compute-tm-by-comp prec ord I a f } (\text{Some tg}) \text{ c} = \text{Some } t$   
**assumes**  $\text{f-deriv}$ :  
 $\bigwedge x. x \in_r \text{compute-bound-tm prec I a tg} \implies c (\text{compute-bound-tm prec I a tg})$   
 $\implies \text{isDERIV } 0 \text{ f } [x]$   
**assumes**  $\text{params}: \text{num-params } (\text{tm-poly tg}) \leq \text{length } I$   
**and**  $\text{dev}: \text{develops-at-within } e \text{ a } I$   
**shows**  $\text{interpret-floatarith } f [x] \in_i \text{range-tm } e \text{ t}$   
**proof** –  
**from**  $\text{t-def}$ [*simplified, simplified Let-def*]  
**obtain**  $\text{tf}$

```

where t1-def: tm-floatarith prec ord [compute-bound-tm prec I (a) tg]
  [mid (compute-bound-tm prec a a tg)] f =
  Some tf
  and t-decomp: t = tm-comp prec ord I a (mid (compute-bound-tm prec a a
tg)) tf tg
  and c-true: c (compute-bound-tm prec I a tg)
  by (auto simp: split-beta' Let-def split: if-splits)
have a1: mid (compute-bound-tm prec a a tg) ∈r (compute-bound-tm prec I a tg)
  apply (rule rev-subsetD[OF mid-in-real-interval])
  apply (rule set-of-real-interval-mono)
  apply (rule compute-bound-tm-mono)
  using params a
  by (auto simp add: set-of-eq elim!: range-tmD)
from ⟨max-Var-floatarith f ≤ 1⟩
have [simp]:  $\bigwedge x. 0 \leq \text{length } x \implies (\lambda x. \text{interpret-floatarith } f [x ! 0]) x = \text{interpret-floatarith } f x$ 
  by (induction f, simp-all)

let ?mid = real-of-float (mid (compute-bound-tm prec a a tg))
have 1: interpret-floatarith f [x] ∈i range-tm (λ-. x - ?mid) tf
  apply (rule tm-floatarith[OF t1-def, simplified])
  subgoal
    apply (rule rev-subsetD)
    apply (rule mid-in-real-interval)
    apply (rule set-of-real-interval-mono)
    apply (rule compute-bound-tm-mono)
    using assms
    by (auto)
  subgoal
    by (rule compute-bound-tm-correct assms) +
  subgoal by (auto intro!: assms c-true)
  subgoal by auto
  done
show ?thesis
  unfolding t-decomp
  apply (rule tm-comp-range)
  apply (rule 1)
  using tm-floatarith-eq-Some-num-params[OF t1-def]
  by (auto simp: intro!: tm-sub-range assms)
qed

```

**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**  $1 \leq X \text{ num-params } (tm\text{-poly } t0) \leq X$

**shows**  $\text{num-params } (tm\text{-poly } t) \leq X$

**using** *assms*

**by** (*auto simp: compute-tm-by-comp.simps Let-def intro!: num-params-tm-comp-le*)

*dest!*: *tm-floatarith-eq-Some-num-params*  
*split*: *option.splits if-splits*)

**lemma** *compute-tm-by-comp-eq-Some-iff*: *compute-tm-by-comp prec ord I a f t0 i*  
 $= \text{Some } t \iff$   
 $(\exists z \ x2. \ t0 = \text{Some } x2 \wedge$   
 $\quad \text{tm-floatarith } \text{prec } \text{ord} \ [\text{compute-bound-tm } \text{prec } I \ a \ x2]$   
 $\quad [\text{mid } (\text{compute-bound-tm } \text{prec } a \ a \ x2)] \ f =$   
 $\quad \text{Some } z$   
 $\wedge \text{tm-comp } \text{prec } \text{ord } I \ a$   
 $\quad (\text{mid } (\text{compute-bound-tm } \text{prec } a \ a \ x2)) \ z \ x2 = t$   
 $\wedge \ i \ (\text{compute-bound-tm } \text{prec } I \ a \ x2))$   
**by** (*auto simp: compute-tm-by-comp.simps Let-def split: option.splits*)

**lemma** *num-params-approx-tm*:  
**assumes** *approx-tm prec ord I a f env = Some t*  
**assumes**  $\bigwedge \text{tm}. \text{tm} \in \text{set } \text{env} \implies \text{num-params } (\text{tm-poly } \text{tm}) \leq \text{length } I$   
**shows**  $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$   
**using** *assms*  
**proof** (*induction f arbitrary: t*)  
**case** (*Add f1 f2*)  
**then show** *?case* **by** (*auto split: option.splits intro!: num-params-tm-add-le*)  
**next**  
**case** (*Minus f*)  
**then show** *?case* **by** (*auto split: option.splits*)  
**next**  
**case** (*Mult f1 f2*)  
**then show** *?case* **by** (*auto split: option.splits intro!: num-params-tm-mul-le*)  
**next**  
**case** (*Inverse f*)  
**then show** *?case*  
**by** (*auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff*  
*intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params*)  
**next**  
**case** (*Cos f*)  
**then show** *?case*  
**by** (*auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff*  
*intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params*)  
**next**  
**case** (*Arctan f*)  
**then show** *?case*  
**by** (*auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff*  
*intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params*)  
**next**  
**case** (*Abs f*)  
**then show** *?case*  
**by** (*auto simp: tm-abs.simps Let-def intro!: num-params-tm-union-le*)  
**next**  
**case** (*Max f1 f2*)

```

    then show ?case
      by (auto simp: tm-max.simps Let-def intro!: num-params-tm-union-le split:
option.splits)
  next
    case (Min f1 f2)
    then show ?case
      by (auto simp: tm-min.simps Let-def intro!: num-params-tm-union-le split:
option.splits)
  next
    case Pi
    then show ?case
      by (auto )
  next
    case (Sqrt f)
    then show ?case
      by (auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff
intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params)
  next
    case (Exp f)
    then show ?case
      by (auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff
intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params)
  next
    case (Powr f1 f2)
    then show ?case
      by (auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff
intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params)
  next
    case (Ln f)
    then show ?case
      by (auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff
intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params)
  next
    case (Power f x2a)
    then show ?case
      by (auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff
intro!: num-params-tm-pow-le dest!: tm-floatarith-eq-Some-num-params)
  next
    case (Floor f)
    then show ?case
      by (auto split: option.splits simp: Let-def compute-tm-by-comp-eq-Some-iff
intro!: num-params-tm-comp-le dest!: tm-floatarith-eq-Some-num-params)
  next
    case (Var x)
    then show ?case by (auto split: if-splits)
  next
    case (Num x)
    then show ?case by auto
qed

```



**lemma** *in-interval-realI*:  $a \in_i I$  if  $a \in_r I$  **using** *that* **by** (*auto simp: set-of-eq*)

**lemma** *all-subset-all-inI*: *map interval-of a all-subset I* if *a all-in I* **using** *that* **by** (*auto simp: in-interval-realI*)

**lemma** *compute-tm-by-comp-None*: *compute-tm-by-comp p ord I a x None k = None* **by** (*rule ccontr*) (*auto simp: compute-tm-by-comp-eq-Some-iff*)

**lemma** *approx-tm-num-Vars-None*:  
*assumes* *max-Var-floatarith f > length env*  
*shows* *approx-tm p ord I a f env = None*  
**using** *assms*  
**by** (*induction f*) (*auto split: option.splits if-splits simp: max-def compute-tm-by-comp-None*)

**lemma** *approx-tm-num-Vars*:  
*assumes* *approx-tm prec ord I a f env = Some t*  
*shows* *max-Var-floatarith f ≤ length env*  
**apply** (*rule ccontr*)  
**using** *approx-tm-num-Vars-None[of env f prec ord I a] assms*  
**by** *auto*

**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<sub>i</sub> range-tms e env*  
*assumes* *devs: develops-at-within e a I*  
*assumes* *env:  $\bigwedge tm. tm \in \text{set env} \implies \text{num-params (tm-poly tm)} \leq \text{length I}$*   
*shows* *interpret-floatarith f xs  $\in_i$  range-tm e t*  
**using** *t-def*  
**proof**(*induct f arbitrary: t*)  
  **case** (*Var n*)  
  **thus** *?case*  
    **using** *assms(2) allin approx-tm-num-Vars[of prec ord I a Var n env t]*  
    **by** (*auto simp: all-in-i-def range-tms-def*)  
  **next**  
  **case** (*Num c*)  
  **thus** *?case*  
    **using** *assms(2) by (auto simp add: assms(3))*  
  **next**  
  **case** (*Add l r t*)  
  **obtain** *t1 where t1-def: approx-tm prec ord I a l env = Some t1*  
  **by** (*metis (no-types, lifting) Add(3) approx-tm.simps(3) option.case-eq-if option.collapse prod.case*)  
  **obtain** *t2 where t2-def: approx-tm prec ord I a r env = Some t2*  
  **by** (*smt Add(3) approx-tm.simps(3) option.case-eq-if option.collapse prod.case*)

```

have t-def: t = tm-add t1 t2
  using Add(3) t1-def t2-def
  by (metis approx-tm.simps(3) option.case(2) option.inject prod.case)

have [simp]: interpret-floatarith (floatarith.Add l r) = interpret-floatarith l +
interpret-floatarith r
  by auto
show ?case
  using Add
  by (auto simp: t-def intro!: tm-add-range Add t1-def t2-def)
next
case (Minus f t)
have [simp]: interpret-floatarith (Minus f) = -interpret-floatarith f
  by auto

obtain t1 where t1-def: approx-tm prec ord I a f env = Some t1
  by (metis Minus.premis(1) approx-tm.simps(4) map-option-eq-Some)
have t-def: t = tm-neg t1
  by (metis Minus.premis(1) approx-tm.simps(4) option.inject option.simps(9)
t1-def)

show ?case
  by (auto simp: t-def intro!: tm-neg-range t1-def Minus)
next
case (Mult l r t)
obtain t1 where t1-def: approx-tm prec ord I a l env = Some t1
  by (metis (no-types, lifting) Mult(3) approx-tm.simps(5) option.case-eq-if op-
tion.collapse prod.case)
obtain t2 where t2-def: approx-tm prec ord I a r env = Some t2
  by (smt Mult(3) approx-tm.simps(5) option.case-eq-if option.collapse prod.case)
have t-def: t = tm-mul prec ord I a t1 t2
  using Mult(3) t1-def t2-def
  by (metis approx-tm.simps(5) option.case(2) option.inject prod.case)

have [simp]: interpret-floatarith (floatarith.Mult l r) = interpret-floatarith l *
interpret-floatarith r
  by auto
show ?case
  using env Mult
  by (auto simp add: t-def intro!: tm-mul-range Mult t1-def t2-def devs
num-params-approx-tm[OF t1-def] num-params-approx-tm[OF t2-def])
next
case (Power f k t)
from Power(2)
obtain tm-f where tm-f-def: approx-tm prec ord I a f env = Some tm-f
  apply (simp) by metis
have t-decomp: t = tm-pow prec ord I a tm-f k
  using Power(2) by (simp add: tm-f-def)
show ?case

```

```

    using env Power
    by (auto simp add: t-def tm-f-def intro!: tm-pow-range Power devs
        num-params-approx-tm[OF tm-f-def])
next
case (Inverse f t)
from Inverse obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  by (auto simp: compute-tm-by-comp-eq-Some-iff)
have safe:  $\bigwedge x. x \in_r (\text{compute-bound-tm prec } I \text{ a } tf) \implies$ 
   $0 < \text{lower } (\text{compute-bound-tm prec } I \text{ a } tf) \vee \text{upper } (\text{compute-bound-tm prec } I \text{ a } tf) < 0 \implies$ 
  isDERIV 0 (Inverse (Var 0)) [x]
  by (simp add: set-of-eq, safe, simp-all)
have np: num-params (tm-poly tf)  $\leq$  length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
from compute-tm-by-comp-range[OF - a
  Inverse(1)[OF tf-def]
  Inverse(2)[unfolded approx-tm.simps tf-def]
  safe np devs]
show ?case by simp
next
case hys: (Cos f t)
from hys obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  by (auto simp: compute-tm-by-comp-eq-Some-iff)
have np: num-params (tm-poly tf)  $\leq$  length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
from compute-tm-by-comp-range[OF - a
  hys(1)[OF tf-def]
  hys(2)[unfolded approx-tm.simps tf-def]
  - np devs]
show ?case by simp
next
case hys: (Arctan f t)
from hys obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  by (auto simp: compute-tm-by-comp-eq-Some-iff)
have np: num-params (tm-poly tf)  $\leq$  length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
from compute-tm-by-comp-range[OF - a
  hys(1)[OF tf-def]
  hys(2)[unfolded approx-tm.simps tf-def]
  - np devs]
show ?case by simp
next
case hys: (Exp f t)

```

```

from hyps obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  by (auto simp: compute-tm-by-comp-eq-Some-iff)
have np: num-params (tm-poly tf) ≤ length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
from compute-tm-by-comp-range[OF - a
  hyps(1)[OF tf-def]
  hyps(2)[unfolded approx-tm.simps tf-def]
  - np devs]
show ?case by simp
next
case hyps: (Ln f t)
from hyps obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  by (auto simp: compute-tm-by-comp-eq-Some-iff)
have safe:  $\bigwedge x. x \in_r \text{compute-bound-tm prec } I \text{ a } tf \implies$ 
   $0 < \text{lower} (\text{compute-bound-tm prec } I \text{ a } tf) \implies \text{isDERIV } 0 \text{ (Ln (Var } 0)) \text{ [x]}$ 
  by (auto simp: set-of-eq)
have np: num-params (tm-poly tf) ≤ length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
from compute-tm-by-comp-range[OF - a
  hyps(1)[OF tf-def]
  hyps(2)[unfolded approx-tm.simps tf-def]
  safe np devs]
show ?case by simp
next
case hyps: (Sqrt f t)
from hyps obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  by (auto simp: compute-tm-by-comp-eq-Some-iff)
have safe:  $\bigwedge x. x \in_r \text{compute-bound-tm prec } I \text{ a } tf \implies$ 
   $0 < \text{lower} (\text{compute-bound-tm prec } I \text{ a } tf) \implies \text{isDERIV } 0 \text{ (Sqrt (Var } 0))$ 
   $\text{[x]}$ 
  by (auto simp: set-of-eq)
have np: num-params (tm-poly tf) ≤ length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
from compute-tm-by-comp-range[OF - a
  hyps(1)[OF tf-def]
  hyps(2)[unfolded approx-tm.simps tf-def]
  safe np devs]
show ?case by simp
next
case (Pi t)
hence t = tm-pi prec by simp
then show ?case
  by (auto intro!: range-tm-tm-pi)

```

```

next
  case (Abs f t)
  from Abs(2) obtain tf where tf-def: approx-tm prec ord I a f env = Some tf
  and t-def: t = tm-abs prec I a tf
  by (metis (no-types, lifting) approx-tm.simps(14) map-option-eq-Some)
  have np: num-params (tm-poly tf) ≤ length I
  using tf-def
  apply (rule num-params-approx-tm)
  using assms by auto
  from tm-abs-range[OF Abs(1)[OF tf-def] np devs]
  show ?case
    unfolding t-def interpret-floatarith.simps(9) comp-def
    by assumption
next
  case hyps: (Min l r t)
  from hyps(3)
  obtain t1 t2 where t-decomp: t = tm-min prec I a t1 t2
  and t1-def: Some t1 = approx-tm prec ord I a l env
  and t2-def: approx-tm prec ord I a r env = Some t2
  by (smt approx-tm.simps(15) option.case-eq-if option.collapse option.distinct(2)
option.inject split-conv)
  from this(2,3) hyps(1-3)
  have t1-range: (interpret-floatarith l xs) ∈i range-tm e t1
  and t2-range: (interpret-floatarith r xs) ∈i range-tm e t2
  by auto

  have [simp]: interpret-floatarith (floatarith.Min l r) = (λvs. min (interpret-floatarith
l vs) (interpret-floatarith r vs))
  by auto
  have np1: num-params (tm-poly t1) ≤ length I
  using t1-def[symmetric]
  apply (rule num-params-approx-tm)
  using assms by auto
  have np2: num-params (tm-poly t2) ≤ length I
  using t2-def
  apply (rule num-params-approx-tm)
  using assms by auto
  show ?case
    unfolding t-decomp(1)
    apply(simp del: tm-min.simps)
    using t1-range t2-range np1 np2
    by (auto intro!: tm-min-range devs)
next
  case hyps: (Max l r t)
  from hyps(3)
  obtain t1 t2 where t-decomp: t = tm-max prec I a t1 t2
  and t1-def: Some t1 = approx-tm prec ord I a l env
  and t2-def: approx-tm prec ord I a r env = Some t2
  by (smt approx-tm.simps(16) option.case-eq-if option.collapse option.distinct(2)

```

```

option.inject split-conv)
from this(2,3) hysps(1-3)
have t1-range: (interpret-floatarith l xs) ∈i range-tm e t1
  and t2-range: (interpret-floatarith r xs) ∈i range-tm e t2
by auto

have [simp]: interpret-floatarith (floatarith.Min l r) = (λvs. min (interpret-floatarith
l vs) (interpret-floatarith r vs))
  by auto
have np1: num-params (tm-poly t1) ≤ length I
  using t1-def[symmetric]
  apply (rule num-params-approx-tm)
  using assms by auto
have np2: num-params (tm-poly t2) ≤ length I
  using t2-def
  apply (rule num-params-approx-tm)
  using assms by auto
show ?case
  unfolding t-decomp(1)
  apply(simp del: tm-min.simps)
  using t1-range t2-range np1 np2
  by (auto intro!: tm-max-range devs)
qed simp-all

```

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* ( $\lambda(i, ivl). \text{tm-of-ivl-par } i \text{ ivl}$ ) (*zip* [0..*length ivls*] *ivls*)

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

**primrec** *approx-slp'*::*nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *float interval list*  $\Rightarrow$  *slp*  $\Rightarrow$  *taylor-model list*  $\Rightarrow$  *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]*:  $x\#xs \text{ all-in}_i \text{ range-tms } e \ (X\#XS) \longleftrightarrow x \in_i \text{ range-tm } e \ X \wedge xs \text{ all-in}_i \text{ range-tms } e \ XS$   
**by** (auto simp: range-tms-def all-in-i-def nth-Cons split: nat.splits)

**lemma** *approx-slp'-range*:  
**assumes** *i*:  $i \text{ all-subset } I$   
**assumes** *dev*:  $\text{develops-at-within } e \ i \ I$   
**assumes** *vs*:  $vs \text{ all-in}_i \text{ range-tms } e \ VS \ (\bigwedge tm. tm \in \text{set } VS \implies \text{num-params } (tm\text{-poly } tm) \leq \text{length } I)$   
**assumes** *appr*:  $\text{approx-slp}' p \text{ ord } I \ i \ ra \ VS = \text{Some } X$   
**shows**  $\text{interpret-slp } ra \ vs \text{ all-in}_i \text{ range-tms } e \ X$   
**using** *appr vs*  
**proof** (induction *ra* arbitrary:  $X \ vs \ VS$ )  
**case** (Cons *ra ras*)  
**from** *Cons.prem*s  
**obtain** *a* **where** *a*:  $\text{approx-tm } p \text{ ord } I \ i \ ra \ VS = \text{Some } a$   
**and** *r*:  $\text{approx-slp}' p \text{ ord } I \ i \ ras \ (a \# VS) = \text{Some } X$   
**by** (auto simp: bind-eq-Some-conv)  
**from** *approx-tm-range*[OF *i a Cons.prem*s(2) *dev Cons.prem*s(3)]  
**have**  $\text{interpret-floatarith } ra \ vs \in_i \text{ range-tm } e \ a$   
**by** auto  
**then have** 1:  $\text{interpret-floatarith } ra \ vs\#vs \text{ all-in}_i \text{ range-tms } e \ (a\#VS)$   
**using** *Cons.prem*s(2)  
**by** auto  
**show** ?case  
**apply** auto  
**apply** (rule *Cons.IH*)  
**apply** (rule *r*)  
**apply** (rule 1)  
**apply** auto  
**apply** (rule *num-params-approx-tm*)  
**apply** (rule *a*)  
**by** (auto intro!: *Cons.prem*s)  
**qed** auto

**definition** *approx-slp::nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  slp  $\Rightarrow$  taylor-model list  $\Rightarrow$  taylor-model list option*  
**where**  
 $\text{approx-slp } p \text{ ord } d \text{ slp } tms =$   
 $\text{map-option } (\text{take } d)$   
 $(\text{approx-slp}' p \text{ ord } (\text{replicate } (\text{length } tms) \ (Ivl \ (-1) \ 1))) \ (\text{replicate } (\text{length } tms) \ 0) \ \text{slp } tms)$

**lemma** *length-range-tms[simp]*:  $\text{length } (\text{range-tms } e \ VS) = \text{length } VS$   
**by** (auto simp: range-tms-def)

```

lemma set-of-Ivl: set-of (Ivl a b) = {a .. b} if a ≤ b
  by (auto simp: set-of-eq that min-def)

lemma set-of-zero[simp]: set-of 0 = {0::'a::ordered-comm-monoid-add}
  by (auto simp: set-of-eq)

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-ini range-tms e VS
  assumes lens:  $\bigwedge tm. tm \in \text{set } VS \implies \text{num-params } (tm\text{-poly } tm) \leq \text{length } vs$ 
  shows interpret-floatariths fas vs all-ini range-tms e X
proof −
  have interpret-floatariths fas vs = take d (interpret-slp slp vs)
    by (simp add: slp-of-fas slp-def d-def)
  also
  have lvs: length vs = length VS
    using assms by (auto simp: all-in-i-def)
  define i where i = replicate (length vs) (0::float interval)
  define I where I = replicate (length vs) (Ivl (−1) 1::float interval)
  from assms obtain XS where
    XS: approx-slp' p ord I i slp VS = Some XS
    and X: take d XS = X
    by (auto simp: approx-slp-def lvs i-def I-def)
  have iI: i all-subset I
    by (auto simp: i-def I-def set-of-Ivl)
  have dev: develops-at-within e i I
    using e
    by (auto simp: develops-at-within-def i-def I-def set-of-Ivl real-interval-Ivl
      real-interval-minus real-interval-zero set-of-eq Pi-iff min-def)
  from approx-slp'-range[OF iI dev vs - XS] lens
  have interpret-slp slp vs all-ini range-tms e XS by (auto simp: I-def)
  then have take d (interpret-slp slp vs) all-ini range-tms e (take d XS)
    by (auto simp: all-in-i-def range-tms-def)
  also note  $\langle \text{take } d \text{ } XS = X \rangle$ 
  finally show ?thesis .
qed

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*  $\Rightarrow$  '*a interval*  $\Rightarrow$  *char list*  $\Rightarrow$  *char list*  
**is**  $\lambda p$  *ivl s. (shows-string "Interval" o shows ivl) s .*

**lift-definition** *shows-list-interval::*  
'*a interval list*  $\Rightarrow$  *char list*  $\Rightarrow$  *char list*  
**is**  $\lambda ivls$  *s. shows-list ivls s .*

**instance**  
**apply** *standard*  
**subgoal by** *transfer (auto simp: show-law-simps)*  
**subgoal by** *transfer (auto simp: show-law-simps)*  
**done**  
**end**  
**end**

**definition** *split-largest-interval :: float interval list  $\Rightarrow$  float interval list  $\times$  float interval list* **where**  
*split-largest-interval xs = (case sort-key (uminus o snd) (zip [0..*length xs*] (map ( $\lambda x$ . upper *x* - lower *x*) *xs*))) of Nil  $\Rightarrow$  ([], [])*  
 $| (i, -)\# \Rightarrow$  *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  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$*   
 $(nat \Rightarrow nat \Rightarrow$  *taylor-model list*  $\Rightarrow$  *taylor-model option*)  $\Rightarrow$  *float interval list list*  
 $\Rightarrow$  *bool* **where**  
*prove-pos prnt 0 p ord F X = (let - = if prnt then print (STR "# depth limit exceeded"  $\boxed{\leftarrow}$ ) else () in False)*  
 $|$  *prove-pos prnt (Suc i) p ord F XXS =*  
 $(case XXS of [] \Rightarrow True | (X\#XS) \Rightarrow$   
 $let$   
 $params = length X;$   
 $R = F p ord (tms-of-ivls X);$   
 $- = if prnt then print (String.implode ((shows "# " o shows (map ( $\lambda ivl$ . (lower *ivl*, upper *ivl*)) *X*)) "  $\boxed{\leftarrow}$ ")) else ()$   
 $in$   
 $if R \neq None \wedge 0 < Inf-tm p params (the R)$   
 $then let - = if prnt then print (STR "# Success"  $\boxed{\leftarrow}$ ) 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 (λp ord xs.  
case approx-slp prec ord 1 slp xs of None ⇒ None | Some [x] ⇒ Some x | Some  
- ⇒ None) xs)

**experiment begin**

**unbundle** floatarith-notation

**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)]  
**by** eval

**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)))]  
**by** eval

**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)))]  
**by** eval

**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