Taylor Models

Christoph Traut and Fabian Immler

March 17, 2025

Abstract

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

Contents

1 Topology for Floating Point Numbers **1**

2 Horner Evaluation **6**

3 Splitting polynomials to reduce floating point precision **11**

4 Splitting polynomials by degree **13**

5 Multivariate Taylor Models **19**
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* ⇒ *float* ⇒ *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* ≤ *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* × *float*) *filter*
where *uniformity-float* = (*INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist x y* < *e*})

instance ..
end

lemma *float-dense-in-real*:
fixes *x* :: *real*
assumes *x* < *y*
shows ∃ *r*∈*float*. *x* < *r* ∧ *r* < *y*
proof −
from ‹*x* < *y*› **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* ≡ *2* ^ *nat* ⌊*bitlen q′*⌋
from *bitlen-bounds*[*of q′*] ‹*0* < *q′*› **have** *q′* < *q*
by (*auto simp*: *q-def*)
then have *inverse q* < *inverse q′*
using ‹*0* < *q′*›

```
    by (auto simp: divide-simps)
  with ‹q' < q› q' have q: inverse (real q) < y - x and 0 < q
    by (auto simp: split: if-splits)
  define p where p = ⌈y * real q⌉ - 1
  define r where r = of-int p / real q
  from q have x < y - inverse (real q)
    by simp
  also from ‹0 < q› have y - inverse (real q) ≤ 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 ∈ float
    by (simp add: r-def q-def)
  ultimately show ?thesis by blast
qed

lemma real-of-float-dense:
  fixes x y :: real
  assumes x < y
  shows ∃ q :: float. x < real-of-float q ∧ real-of-float q < y
  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 ⇒ bool where
  open-float S = (∀ x∈S. ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ S)

instance
proof (standard, intro ext iffI)
  fix U::float set
  assume generate-topology (range lessThan ∪ 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 ∈ a x ∈ b
      with Int(3,4) obtain e1 e2
        where dist (y) (x) < e1 ⟹ y ∈ a
          and dist (y) (x) < e2 ⟹ y ∈ b
          and 0 < e1 0 < e2
        for y
```

```
      by (auto dest!: bspec)
    then show ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ a ∩ 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 ∈ X and X: X ∈ K
   from UN[OF X] x obtain e where
    dist (y) (x) < e ⟹ y ∈ X e > 0 for y
    by auto
   then show ∃ e>0. ∀ y. dist (real-of-float y) (real-of-float x) < e ⟶ y ∈ ⋃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 ∃ e>0. ∀ y. dist (real-of-float y) (real-of-float x) < e ⟶ y ∈
{..<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 ∃ e>0. ∀ y. dist (real-of-float y) (real-of-float x) < e ⟶ y ∈
{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 ∈ U ⟹ e x > 0
  x ∈ U ⟹ dist ( y) ( x) < e x ⟹ y ∈ U for x y
  unfolding open-float-def uniformity-float-def
  by metis
 {
  fix x
  assume x: x ∈ U
  obtain e' where e': e' > 0 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*′ ⟹ *y* ∈ *U* **for** *y*
by (*intro e*[*OF x*]) *auto*
then have ∃ *e*′>*0*. ∀ *y*. *dist* (*y*) (*x*) < *real-of-float e*′ ⟶ *y* ∈ *U*
using *e*′
by *auto*
} then
obtain *e*′ **where** *e*′:
x ∈ *U* ⟹ *0* < *e*′ *x*
x ∈ *U* ⟹ *dist y x* < *real-of-float* (*e*′ *x*) ⟹ *y* ∈ *U* **for** *x y*
by *metis*
then have *U* = (⋃ *x*∈*U*. *greaterThan* (*x* − *e*′ *x*) ∩ *lessThan* (*x* + *e*′ *x*))
by (*auto simp*: *dist-float-def dist-commute dist-real-def*)
also have *generate-topology* (*range lessThan* ∪ *range greaterThan*) ...
by (*intro generate-topology-Union generate-topology.Int generate-topology.Basis*) *auto*
finally show *generate-topology* (*range lessThan* ∪ *range greaterThan*) *U* .
qed

end

instance *float* :: *metric-space*
proof *standard*
fix *U*::*float set*
show *open U* = (∀ *x*∈*U*. ∀ *F* (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ *U*)
unfolding *open-float-def open-dist uniformity-float-def uniformity-real-def*
proof *safe*
fix *x*
assume ∀ *x*∈*U*. ∃ *e*>*0*. ∀ *y*. *dist* (*real-of-float y*) (*real-of-float x*) < *e* ⟶ *y* ∈ *U x* ∈ *U*
then obtain *e* **where** *e* > *0 dist* (*y*) (*x*) < *e* ⟹ *y* ∈ *U* **for** *y*
by *auto*
then show ∀ *F* (*x*′, *y*) *in INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist x y* < *e*}. *x*′ = *x* ⟶ *y* ∈ *U*
by (*intro eventually-INF1*[**where** *i*=*e*])
(*auto simp*: *eventually-principal dist-commute dist-float-def*)
next
fix *u*
assume ∀ *x*∈*U*. ∀ *F* (*x*′, *y*) *in INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist x y* < *e*}. *x*′ = *x* ⟶ *y* ∈ *U*
u ∈ *U*
from *this* **obtain** *E* **where** *E*: *E* ⊆ {*0*<..} *finite E*
∀ (*x*′, *y*)∈⋂ *x*∈*E*. {(*y*′, *y*). *dist y*′ *y* < *x*}. *x*′ = *u* ⟶ *y* ∈ *U*
by (*subst* (*asm*) *eventually-INF*) (*auto simp*: *INF-principal-finite eventually-principal*)
then show ∃ *e*>*0*. ∀ *y*. *dist* (*real-of-float y*) (*real-of-float u*) < *e* ⟶ *y* ∈ *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-eval′ f x v 0 = v
| horner-eval′ f x v (Suc i) = horner-eval′ f x (f i + x * v) i

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

lemma horner-eval-cong:
  assumes ⋀i. i < n ⟹ f i = g i
  assumes x = y
  assumes n = m
  shows horner-eval f x n = horner-eval g y m
proof−
  {
    fix v have horner-eval′ f x v n = horner-eval′ g x v n
      using assms(1) 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-eval f x n = (∑ i<n. f i * x^i)
proof−
  {
    fix v have horner-eval′ f x v n = (∑ i<n. f i * x^i) + v * x^n
      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-eval f x (Suc n) = horner-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-eval f x (Suc n) = f 0 + x * (horner-eval (λi. f (Suc i)) x n)
proof−
  {
    fix v have horner-eval′ f x v (Suc n) = f 0 + x * horner-eval′ (λi. 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 ⋀*i*. *i* < *n* ⟹ *f i* ∈ *set-of* (*g i*)
assumes *x* ∈$_i$ *I v* ∈$_i$ *V*
shows *horner-eval′ f x v n* ∈$_i$ *horner-eval′ g I V n*
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 ⋀*i*. *i* < *n* ⟹ *f i* ∈ *set-of* (*g i*)
assumes *x* ∈ *set-of I*
shows *horner-eval f x n* ∈$_i$ *horner-eval g I n*
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*]: *real-of-float x = 0* ⟷ *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*⇒*float interval*]]

Apply float interval arguments to a float poly.

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

map-poly for homomorphisms

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

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

by *auto*

lemma *map-poly-homo-polyadd*:
map-poly f (*p1* $+_p$ *p2*) = *map-poly f p1* $+_p$ *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 ($\sim_p$ *p1*) = $\sim_p$ (*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* $-_p$ *p2*) = *map-poly f p1* $-_p$ *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* $*_p$ *p2*) = *map-poly f p1* $*_p$ *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* $\hat{}_p$ *n*) = *map-poly f p1* $\hat{}_p$ *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* ⇒ *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 +_p 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 (~_p p) = num-params p
  by (induction p rule: polyneg.induct) simp-all

lemma num-params-polymul:
  shows num-params (p1 *_p 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 ^_p 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-eqiv*:
  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-in*$_i$ *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 polynate only contain *poly.C* and *poly.CN* constructors.

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

```
  | split-by-prec prec (poly.Neg p) = (let (l, r) = split-by-prec prec p in (poly.Neg l,
poly.Neg r))
  | split-by-prec prec (poly.Pw p 0) = (poly.C 1, poly.C 0)
  | split-by-prec prec (poly.Pw p (Suc n)) = (let (l, r) = split-by-prec prec p in
(poly.Pw l n, poly.Sub (poly.Pw p (Suc n)) (poly.Pw l n)))
  | split-by-prec prec (poly.CN c n p) = (let (cl, cr) = split-by-prec prec c;
                                              (pl, pr) = split-by-prec prec p
                                          in (poly.CN cl n pl, poly.CN cr n pr))
```

TODO: Prove precision constraint on l.

```
lemma split-by-prec-correct:
  fixes args :: real list
  assumes (l, r) = split-by-prec prec p
  shows Ipoly args p = Ipoly args l + Ipoly args r (is ?P1)
    and   num-params l ≤ num-params p (is ?P2)
    and   num-params r ≤ num-params p (is ?P3)
  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* ⇒ *nat*
  where *maxdegree* (*poly.C c*) = *0*
  | *maxdegree* (*poly.Bound n*) = *1*
  | *maxdegree* (*poly.Add l r*) = *max* (*maxdegree l*) (*maxdegree r*)
  | *maxdegree* (*poly.Sub l r*) = *max* (*maxdegree l*) (*maxdegree r*)
  | *maxdegree* (*poly.Mul l r*) = *maxdegree l* + *maxdegree r*
  | *maxdegree* (*poly.Neg p*) = *maxdegree p*
  | *maxdegree* (*poly.Pw p n*) = *n* ∗ *maxdegree p*
  | *maxdegree* (*poly.CN c n p*) = *max* (*maxdegree c*) (*1* + *maxdegree p*)

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

lemma *split-by-degree-correct*:
  fixes *x* :: *real list* **and** *p* :: *float poly*
  assumes (*l*, *r*) = *split-by-degree ord p*
  shows *maxdegree l* ≤ *ord* (**is** *?P1*)
    and *Ipoly x p* = *Ipoly x l* + *Ipoly x r* (**is** *?P2*)
    and *num-params l* ≤ *num-params p* (**is** *?P3*)
    and *num-params r* ≤ *num-params p* (**is** *?P4*)
  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 = [] | 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 ∧ 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 $n < length\ a$
assumes $n < length\ b$
shows $(map2\ f\ a\ b)!n = f\ (a!n)\ (b!n)$
using *assms*
proof(*induction n arbitrary*: *a b*)
case (*0 a b*)
have $0 < length\ a$ **and** $0 < length\ b$
using *0*
by *simp-all*
thus *?case*
using *0*
by *simp*
next
case (*Suc n a b*)
from *Suc.prems* **have** $0 < length\ a\ 0 < length\ b\ n < length\ (tl\ a)\ n < length\ (tl\ b)$
using *Suc.prems* **by** *auto*
have $map2\ f\ a\ b = map2\ f\ (hd\ a\ \#\ tl\ a)\ (hd\ b\ \#\ tl\ b)$
using ‹$0 < length\ a$› ‹$0 < length\ b$›
by *simp*
also have $\ldots\ !\ Suc\ n = map2\ f\ (tl\ a)\ (tl\ b)\ !\ n$
by *simp*
also have $\ldots = f\ (tl\ a\ !\ n)\ (tl\ b\ !\ n)$
using ‹$n < length\ (tl\ a)$› ‹$n < length\ (tl\ b)$› **by** (*rule Suc.IH*)
also have $tl\ a\ !\ n = (hd\ a\ \#\ tl\ a)\ !\ Suc\ n$ **by** *simp*
also have $(hd\ a\ \#\ tl\ a) = a$ **using** ‹$0 < length\ a$› **by** *simp*
also have $tl\ b\ !\ n = (hd\ b\ \#\ tl\ b)\ !\ Suc\ n$ **by** *simp*
also have $(hd\ b\ \#\ tl\ b) = b$ **using** ‹$0 < length\ b$› **by** *simp*
finally show *?case* **.**
qed

Translating a polynomial by a vector.

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

Translating a polynomial is equivalent to translating its argument.

lemma *poly-translate-correct*:
assumes *num-params p* ≤ *length x*

assumes *length x = length v*
shows *Ipoly x (poly-translate v p) = Ipoly (map2 (+) x v) p*
using *assms*
by (*induction p*, *simp-all*)

lemma *real-poly-translate*:
assumes *num-params p ≤ length v*
shows *Ipoly x (map-poly real-of-float (poly-translate v p)) = Ipoly x (poly-translate v (map-poly real-of-float p))*
using *assms*
by (*induction p*, *simp-all*)

lemma *num-params-poly-translate*[*simp*]:
shows *num-params (poly-translate v p) = 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*: (*of-nat n* :: *real*) = (*of-nat n* :: *float*)
by (*induction n*, *simp-all add*: *of-nat-def*)

lemma *fact-real-float-equiv*: (*fact n* :: *float*) = (*fact n* :: *real*)
by (*induction n*) *simp-all*

lemma *Some-those-length*:
those ys = Some xs ⟹ length xs = length ys
by (*induction ys arbitrary*: *xs*) (*auto split*: *option.splits*)

lemma *those-eq-None-iff*: *those ys = None ⟷ None ∈ set ys*
by (*induction ys*) (*auto simp*: *split*: *option.splits*)

lemma *those-eq-Some-iff*: *those ys = (Some xs) ⟷ (ys = map Some xs)*
by (*induction ys arbitrary*: *xs*) (*auto simp*: *split*: *option.splits*)

lemma *Some-those-nth*:
assumes *those ys = Some xs*
assumes *i < length xs*
shows *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*: $f\hat{}n = (\lambda x.\ (f\ x)\hat{}n)$
by (*induction n*, *simp-all*)

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

Translate floatarith expressions by a vector of floats.

fun *fa-translate* :: *float list* ⇒ *floatarith* ⇒ *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* ≤ *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*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Mult a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Inverse a*) = *vars-floatarith a*
| *vars-floatarith* (*Minus a*) = *vars-floatarith a*
| *vars-floatarith* (*Num a*) = {}
| *vars-floatarith* (*Var i*) = {*i*}
| *vars-floatarith* (*Cos a*) = *vars-floatarith a*
| *vars-floatarith* (*Arctan a*) = *vars-floatarith a*
| *vars-floatarith* (*Abs a*) = *vars-floatarith a*
| *vars-floatarith* (*Max a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Min a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Pi*) = {}
| *vars-floatarith* (*Sqrt a*) = *vars-floatarith a*

| *vars-floatarith* (*Exp a*) = *vars-floatarith a*
| *vars-floatarith* (*Powr a b*) = (*vars-floatarith a*) ∪ (*vars-floatarith b*)
| *vars-floatarith* (*Ln a*) = *vars-floatarith a*
| *vars-floatarith* (*Power a n*) = *vars-floatarith a*
| *vars-floatarith* (*Floor a*) = *vars-floatarith a*

lemma *finite-vars-floatarith*[*simp*]: *finite* (*vars-floatarith x*)
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 inteval and use real poly/real interval and data refinement?

5 Multivariate Taylor Models

5.1 Computing interval bounds on arithmetic expressions

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

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

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

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

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

lemma *Ipoly-num-params-cong*: *Ipoly xs p* = *Ipoly ys p*
if ⋀*i*. *i* < *num-params p* ⟹ *xs* ! *i* = *ys* ! *i*
using *that*
by (*induction p*; *auto*)

lemma *insertion-num-params-cong*: *insertion e p* = *insertion f p*
if ⋀*i*. *i* < *num-params p* ⟹ *e i* = *f i*
using *that*
by (*induction p*; *auto*)

lemma *insertion-eq-IpolyI*: *insertion xs p* = *Ipoly ys p*
if ⋀*i*. *i* < *num-params p* ⟹ *xs i* = *ys* ! *i*
using *that*
by (*induction p*; *auto*)

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

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

lemma *range-tmI*:
x ∈$_i$ *range-tm e tm*
if *x*: *x* ∈$_i$ *interval-of* (*insertion e* ((*tm-poly tm*))) + *real-interval* (*tm-bound tm*)
for *e*::*nat*⇒*real*
by (*auto simp*: *range-tm-def x*)

lemma *range-tmD*:
x ∈$_i$ *interval-of* (*insertion e* (*tm-poly tm*)) + *real-interval* (*tm-bound tm*)
if *x* ∈$_i$ *range-tm e tm*
for *e*::*nat*⇒*real*
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* ⇒ *float interval poly* ⇒ (*float interval list*) ⇒ (*float interval list*) ⇒ *float interval* **where**
compute-bound-poly prec (*poly.C f*) *I a* = *f*
| *compute-bound-poly prec* (*poly.Bound n*) *I a* = *round-interval prec* (*I* ! *n* − (*a* ! *n*))
| *compute-bound-poly prec* (*poly.Add p q*) *I a* =
round-interval prec (*compute-bound-poly prec p I a* + *compute-bound-poly prec q I a*)
| *compute-bound-poly prec* (*poly.Sub p q*) *I a* =
round-interval prec (*compute-bound-poly prec p I a* − *compute-bound-poly prec q I a*)
| *compute-bound-poly prec* (*poly.Mul p q*) *I a* =
mult-float-interval prec (*compute-bound-poly prec p I a*) (*compute-bound-poly prec q I a*)
| *compute-bound-poly prec* (*poly.Neg p*) *I a* = −*compute-bound-poly prec p I a*
| *compute-bound-poly prec* (*poly.Pw p n*) *I a* = *power-float-interval prec n* (*compute-bound-poly prec p I a*)
| *compute-bound-poly prec* (*poly.CN p n q*) *I a* =
round-interval prec (*compute-bound-poly prec p I a* +
mult-float-interval prec (*round-interval prec* (*I* ! *n* − (*a* ! *n*))) (*compute-bound-poly prec q I a*))

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

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

lemma *compute-bound-tm-def*:
compute-bound-tm prec I a tm = *compute-bound-poly prec* (*tm-poly tm*) *I a* + (*tm-bound tm*)
by (*cases tm*) *auto*

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

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

lemma *real-interval-plus*: *real-interval* (*a* + *b*) = *real-interval a* + *real-interval b*
by (*simp add*: *interval-eqI*)

lemma *real-interval-uminus*: *real-interval* (− *b*) = − *real-interval b*
by (*simp add*: *interval-eqI*)

lemma *real-interval-of*: *real-interval* (*interval-of b*) = *interval-of b*
by (*simp add*: *interval-eqI*)

lemma *real-interval-minus*: *real-interval* (*a* − *b*) = *real-interval a* − *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* **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* **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* **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*: *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* $(a \hat{} b) =$ *real-interval* $a \hat{} b$
by (*induction b arbitrary*: *a*)
(*auto simp*: *real-interval-times real-interval-one*)

lemma *in-set-real-power-interval*[*intro*, *simp*]:
$x \hat{} n \in_r X \hat{} n$ **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*]:
$x \hat{} n \in_r$ *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*]:
$x * y \in_r$ *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* ⇒ *real*) ⇒ *float interval list* ⇒ *float interval list* ⇒ *bool*
where *develops-at-within e a I* ⟷ (*a all-subset I*) ∧ (∀ *i* < *length I*. *e i* ∈$_r$ *I* ! *i* − *a* ! *i*)

lemma *develops-at-withinI*:
assumes *all-in*: *a all-subset I*
assumes *e*: ⋀*i*. *i* < *length I* ⟹ *e i* ∈$_r$ *I* ! *i* − *a* ! *i*
shows *develops-at-within e a I*
using *assms* **by** (*auto simp*: *develops-at-within-def*)

lemma *develops-at-withinD*:
assumes *develops-at-within e a I*
shows *a all-subset I*
⋀*i*. *i* < *length I* ⟹ *e i* ∈$_r$ *I* ! *i* − *a* ! *i*
using *assms* **by** (*auto simp*: *develops-at-within-def*)

lemma *compute-bound-poly-correct*:
fixes *p*::*float poly*
assumes *num-params p* ≤ *length I*
assumes *dev*: *develops-at-within e a I*

shows *insertion e* (*p*::*real poly*) $\in_r$ *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* :: *float interval list* **and** *f* :: *real list* $\Rightarrow$ *real*
assumes *n*: *num-params* (*tm-poly t*) $\leq$ *length I*
assumes *dev*: *develops-at-within e a I*
assumes *x0*: *x0* $\in_i$ *range-tm e t*
shows *x0* $\in_r$ *compute-bound-tm prec I a t*
proof −
let *?I* = *insertion e* (*tm-poly t*)
have *x0* = *?I* + (*x0* − *?I*) **by** *simp*
also have ... $\in_r$ *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 ∈ᵣ compute-bound-tm prec I a t .
qed

lemma compute-bound-tm-correct-subset:
  fixes I :: float interval list and f :: real list ⇒ real
  assumes n: num-params (tm-poly t) ≤ length I
  assumes dev: develops-at-within e a I
  shows set-of (range-tm e t) ⊆ set-of (real-interval (compute-bound-tm prec I a
t))
  using assms
  by (auto intro!: compute-bound-tm-correct)

lemma compute-bound-poly-mono:
  assumes num-params p ≤ length I
  assumes mem: I all-subset J a all-subset I
  shows set-of (compute-bound-poly prec p I a) ⊆ set-of (compute-bound-poly prec
p J a)
  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* ⇒ *real*
assumes *num-params* (*tm-poly t*) ≤ *length I*
assumes *I all-subset J*
assumes *a all-subset I*
shows *set-of* (*compute-bound-tm prec I a t*) ⊆ *set-of* (*compute-bound-tm prec J a t*)
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* ⇒ *taylor-model*
where *tm-const c* = *TaylorModel* (*poly.C c*) *0*

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

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

lemma *zero-real-interval*[*intro*,*simp*]: *0* ∈$_r$ *0*
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* ∈$_i$ *range-tm e* (*tm-pi prec*)
proof−
have ⋀*prec*. *real-of-float* (*lb-pi prec*) ≤ *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* ⇒ *floatarith* ⇒ *nat* ⇒ *floatarith*
where *deriv v f 0* = *f*
| *deriv v f* (*Suc n*) = *DERIV-floatarith v* (*deriv v f n*)

lemma *isDERIV-DERIV-floatarith*:
assumes *isDERIV v f vs*
shows *isDERIV v* (*DERIV-floatarith v f*) *vs*
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 ((λ*x*. *interpret-floatarith* (*deriv i f n*) (*xs*[*i*:=*x*])) *has-real-derivative interpret-floatarith* (*deriv i f* (*Suc n*)) (*xs*[*i*:=*t*]))
(*at t within S*)
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* ⇒ *nat* ⇒ *floatarith*
where *deriv-rec* (*Exp* (*Var 0*)) - = *Exp* (*Var 0*)

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

lemma deriv-rec-correct:
  assumes isDERIV 0 f (xs[0:=t]) 0 < length xs
   shows ((λx. 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 = Cos (Var 0)

  have n-mod-4-cases: n mod 4 = 0 | n mod 4 = 1 | n mod 4 = 2 | n mod 4 = 3
    by auto
  have Sin-sin: (λxs. interpret-floatarith (Sin (Var 0)) xs) = (λxs. sin (xs!0))
    by simp
 show ((λx. interpret-floatarith (deriv-rec (Cos (Var 0)) n) (xs[0:=x])) has-real-derivative
        interpret-floatarith (deriv-rec (Cos (Var 0)) (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 = Inverse (Var 0) and isDERIV 0 f (xs[0:=t])
  hence t ≠ 0 using assms
    by simp
  {
    fix n::nat and x::real
    assume x ≠ 0
    moreover have (n mod 2 = 0 ∧ Suc n mod 2 = 1) ∨ (n mod 2 = 1 ∧ Suc n
mod 2 = 0)
      by (cases n rule: parity-cases) auto
     ultimately have interpret-floatarith (deriv-rec f n) (xs[0:=x]) = fact n *
(−1::real)^n / (x^Suc n)
      using assms by (auto simp add: f-def field-simps fact-real-float-equiv)
  }
  note closed-formula = this

  have ((λx. inverse (x ^ Suc n)) has-real-derivative −real (Suc n) * inverse (t ^
Suc (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 ‹$t \neq 0$›
by (*simp-all add*: *divide-simps*)
hence ((λ*x*. *fact n* ∗ (−*1*::*real*)^*n* ∗ *inverse* (*x* ^ *Suc n*)) *has-real-derivative fact* (*Suc n*) ∗ (− *1*) ^ *Suc n* / *t* ^ *Suc* (*Suc n*)) (*at t*)
apply(*rule iffD1*[*OF DERIV-cong-ev*, *OF refl - - DERIV-cmult*[**where** *c*=*fact n* ∗ (−*1*::*real*)^*n*], *rotated 2*])
using ‹$t \neq 0$›
by (*simp-all add*: *field-simps distrib-left*)
then show ((λ*x*. *interpret-floatarith* (*deriv-rec* (*Inverse* (*Var 0*)) *n*) (*xs*[*0*:=*x*])) *has-real-derivative*
interpret-floatarith (*deriv-rec* (*Inverse* (*Var 0*)) (*Suc n*)) (*xs*[*0*:=*t*]))
(*at t within S*)
apply −
apply (*rule has-field-derivative-at-within*)
apply(*rule iffD1*[*OF DERIV-cong-ev*[*OF refl - closed-formula*[*OF* ‹$t \neq 0$›, *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** ‹*simp-all add*: *has-field-derivative-subset*[*OF DERIV-exp subset-UNIV*]›)

lemma *deriv-rec-0-idem*[*simp*]:
shows *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* :: *nat* ⇒ *float interval list* ⇒ *floatarith* ⇒ *nat* ⇒ *float interval option*
where *tmf-c prec I f i* = *compute-bound-fa prec* (*Mult* (*deriv-rec f i*) (*Inverse* (*Num* (*fact i*)))) *I*
— The interval coefficients of the Taylor polynomial, i.e. the real coefficients approximated by a float interval.

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

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

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

term *compute-bound-poly*
lemma *tmf-c-correct*:
fixes *A*::*float interval list* **and** *I*::*float interval* **and** *f*::*floatarith* **and** *a*::*real list*
assumes *a all-in A*
assumes *tmf-c prec A f i* = *Some I*
shows *interpret-floatarith* (*deriv-rec f i*) *a* / *fact i* ∈$_r$ *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 ⋀*i*. *i* < *ord* ⟹ *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* ‹*i* < *ord*›
by *simp-all*
then show *tmf-c prec a f i* = *Some* (*cs*!*i*)
using ‹*i* < *ord*›

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*:
Ipoly xs (*fst* (*tmf-polys z*)) = (∑ *i*<*length z*. *xs* ! *0* ^ *i* ∗ (*mid* (*z* ! *i*)))
for *xs*::*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*:
insertion e (*fst* (*tmf-polys z*)) = (∑ *i*<*length z*. *e 0* ^ *i* ∗ (*mid* (*z* ! *i*)))
for *e*::*nat* ⇒ *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*:
set-of (*horner-eval* (*real-interval o centered o nth z*) *x* (*length z*)) ⊆ *set-of* (*Ipoly* [*x*] (*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* ∈*i* *0*
by (*simp add*: *set-of-eq*)

lemma *sum-in-intervalI*: *sum f X* ∈*i* *sum g X* **if** ⋀*x*. *x* ∈ *X* ⟹ *f x* ∈*i* *g x*
for *f* :: - ⇒ *′a* :: *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*: *set-of* (*sum f X*) ⊆ *set-of* (*sum g X*)
if ⋀*x*. *x* ∈ *X* ⟹ *set-of* (*f x*) ⊆ *set-of* (*g x*)
for *f* :: -⇒′*a*::*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*: *interval-of* (*a* + *b*) = *interval-of a* + *interval-of b*
by (*simp add*: *interval-eqI*)

lemma *interval-of-uminus*: *interval-of* (− *a*) = − *interval-of a*
by (*simp add*: *interval-eqI*)

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

lemma *interval-of-sum*: *interval-of* (*sum f X*) = *sum* (λ*x*. *interval-of* (*f x*)) *X*
by (*induction X rule*: *infinite-finite-induct*) (*auto simp*: *interval-of-plus interval-of-zero*)

lemma *interval-of-prod*: *interval-of* (*a* ∗ *b*) = *interval-of a* ∗ *interval-of b*
by (*simp add*: *lower-times upper-times interval-eqI*)

lemma *in-set-of-interval-of*[*simp*]: *x* ∈*ᵢ* (*interval-of y*) ⟷ *x* = *y* **for** *x y*::′*a*::*order*
by (*auto simp*: *set-of-eq*)

lemma *real-interval-Ipoly*: *real-interval* (*Ipoly xs p*) = *Ipoly* (*map real-interval xs*) (*map-poly real-interval p*)
if *num-params p* ≤ *length xs*
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*: *num-params* (*fst* (*tmf-polys z*)) ≤ *Suc 0*
by (*induction z*) (*auto simp*: *split-beta′ Let-def*)

lemma *num-params-tmf-polys2*: *num-params* (*snd* (*tmf-polys z*)) ≤ *Suc 0*
by (*induction z*) (*auto simp*: *split-beta′ Let-def*)

lemma *set-of-real-interval-subset*: *set-of* (*real-interval x*) ⊆ *set-of* (*real-interval y*)
if *set-of x* ⊆ *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* ∈*ᵣ I* ! *0*

assumes *xs-ne*: $xs \neq []$
assumes *deriv*: $\bigwedge x.\ x \in_r I\ !\ 0 \Longrightarrow isDERIV\ 0\ f\ (xs[0 := x])$
assumes $\bigwedge i.\ 0 < i \Longrightarrow i < length\ xs \Longrightarrow e\ i = real\text{-}of\text{-}float\ (xs\ !\ i)$
assumes *diff-e*: $(x - real\text{-}of\text{-}float\ (xs\ !\ 0)) = e\ 0$
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*: $i < ord \Longrightarrow$ *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** ‹$xs \neq []$›
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*: $?a \leq ?c\ ?c \leq ?b\quad ?a \leq ?x\ ?x \leq ?b$
using *a xs-ne x*
by (*force simp*: *set-of-eq*)+

define *cr* **where** *cr* ≡ $\lambda s\ m.$ *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* ≡ $\lambda i.$ *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*: $(\sum m{<}ord.\ cr\ s\ m * (x - (xs\ !\ 0))\ \hat{}\ m) + cr\ s\ ord * (x - (xs\ !\ 0))\ \hat{}\ ord$

```
      ∈r round-interval prec (Ipoly (List.map2 (−) I (map interval-of xs)) (snd
(tmf-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 ∘ (!) 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 (tmf-polys z))))
      (is - ⊆ set-of ?x)
      by (rule Ipoly-snd-tmf-polys)
    also have ... = set-of (real-interval (Ipoly [(I ! 0 − xs ! 0)] (snd (tmf-polys
z))))
      by (auto simp: real-interval-Ipoly num-params-tmf-polys2)
    also have ... ⊆ set-of (real-interval (round-interval prec (Ipoly [(I ! 0 − xs !
0)] (snd (tmf-polys z)))))
      by (rule set-of-real-interval-subset) (rule round-ivl-correct)
    also
    have Ipoly [I ! 0 − interval-of (xs ! 0)] (snd (tmf-polys z)) = Ipoly (List.map2
(−) I (map interval-of xs)) (snd (tmf-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-tmf-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* *0*$_p$
by *simp*
finally have *fst* (*tmf-polys z*) = *CN* (*mid* (*z* ! *0*))$_p$ *0* *0*$_p$.
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*: {..<*length z*} = *insert 0* {*1*..<*length z*}
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* ⇒ *taylor-model*
where *tm-norm-poly* (*TaylorModel p e*) = *TaylorModel* (*polynate p*) *e*
— Normalizes the Taylor model by transforming its polynomial into horner form.

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

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

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

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

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

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

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

fun *tm-pow* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float interval list* ⇒ *taylor-model* ⇒ *nat* ⇒ *taylor-model*
  where *tm-pow prec ord I a t 0* = *tm-const 1*
  | *tm-pow prec ord I a t* (*Suc n*) = (
      if odd (*Suc n*)
      then tm-mul prec ord I a t (*tm-pow prec ord I a t n*)
      else let t′ = *tm-pow prec ord I a t* ((*Suc n*) *div 2*)
        in tm-mul prec ord I a t′ t′
)

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

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

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

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

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

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

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

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

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

Rangeity of is preserved by our operations on Taylor models.

lemma *insertion-polyadd*[*simp*]: *insertion e* (*a* +$_p$ *b*) = *insertion e a* + *insertion e b*
for *a b*::′*a*::*ring-1 poly*
apply (*induction a b rule*: *polyadd.induct*)
apply (*auto simp*: *algebra-simps Let-def*)
by (*metis* (*no-types*) *mult-zero-right ring-class.ring-distribs*(*1*))

lemma *insertion-polyneg*[*simp*]: *insertion e* (~$_p$ *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..<num-params p]
```

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

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

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

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

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

have *range-tm e t* =
interval-of (*insertion e* (*map-poly real-of-float pl*)) +
(*interval-of* (*insertion e* (*map-poly real-of-float pr*)) + *real-interval err*)
by (*auto simp*: *t-decomp range-tm-def ins ac-simps interval-of-plus*) **term** *round-interval*
also have *set-of* ... $\subseteq$ *set-of* (*interval-of* (*insertion e pl*)) +
set-of (*real-interval* (*round-interval prec* (*err* + *compute-bound-poly 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 ... = *set-of* (*range-tm e* (*tm-lower-order-of-normed prec ord I a t*))
by (*simp add*: *t-decomp split-beta′ Let-def p-split range-tm-def set-of-plus*)
finally show *?thesis* **using** *assms* **by** *auto*
qed

lemma *num-params-tm-norm-poly-le*: *num-params* (*tm-poly* (*tm-norm-poly t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
using *that*
by (*cases t*) (*auto simp*: *intro*!: *num-params-polynate*[*THEN order-trans*])

lemma *tm-lower-order-range*:
assumes *x* ∈$_i$ *range-tm e t*
assumes *dev*: *develops-at-within e a I*
assumes *num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-lower-order prec ord I a t*)
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* ∈$_i$ *range-tm e t*
assumes *dev*: *develops-at-within e a I*
assumes *num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-round-floats-of-normed prec I a t*)
— TODO: this is a clone of ⟦*?x* ∈$_i$ *range-tm ?e ?t*; *develops-at-within ?e ?a ?I*; *num-params* (*tm-poly ?t*) ≤ *length ?I*⟧ ⟹ *?x* ∈$_i$ *range-tm ?e* (*tm-lower-order-of-normed ?prec ?ord ?I ?a ?t*) -> general sweeping method!
proof−
obtain *p err* **where** *t-decomp*: *t* = *TaylorModel p err*
by (*cases t*) *simp*
obtain *pl pr* **where** *p-prec*: *split-by-prec prec p* = (*pl*, *pr*)
by (*cases split-by-prec prec p*, *simp*)

from *split-by-prec-correct-insertion*[*OF p-prec*]
have *params*: *num-params pl* ≤ *num-params p num-params pr* ≤ *num-params p*
and *ins*: *insertion e* (*map-poly real-of-float p*) =
insertion e (*map-poly real-of-float pl*) + *insertion e* (*map-poly real-of-float pr*)
by *auto*
from *assms params* **have** *params-pr*: *num-params pr* ≤ *length I*
by (*auto simp*: *t-decomp*)

have *range-tm e t* =
interval-of (*insertion e* (*map-poly real-of-float pl*)) +
(*interval-of* (*insertion e* (*map-poly real-of-float pr*)) + *real-interval err*)
by (*auto simp*: *t-decomp range-tm-def ins ac-simps interval-of-plus*)
also have *set-of* ... ⊆ *set-of* (*interval-of* (*insertion e pl*)) +
set-of (*real-interval* (*round-interval prec* (*err* + *compute-bound-poly 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 ... = *set-of* (*range-tm e* (*tm-round-floats-of-normed prec I 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*: *num-params* (*fst* (*split-by-degree ord x*)) ≤ *K*
num-params (*snd* (*split-by-degree ord x*)) ≤ *K*
if *num-params x* ≤ *K* **for** *x*::*float poly*
using *split-by-degree-correct-insertion*(*3*,*4*)[*of ord x*, *OF surjective-pairing*] *that*
by *auto*

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

lemma *num-params-tm-norm'-le*:
num-params (*tm-poly* (*tm-round-floats-of-normed prec I a t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
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* ∈$_i$ *range-tm e t develops-at-within e a I num-params* (*tm-poly t*) ≤ *length I*
shows *x* ∈$_i$ *range-tm e* (*tm-round-floats prec I a t*)
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*: *num-params* (*tm-poly* (*tm-lower-order-of-normed prec ord I a t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
using *that*
apply (*cases t*)
apply (*auto simp*: *split-beta' Let-def intro*!: *num-params-polynate*[*THEN order-trans*])
apply (*rule order-trans*[*OF split-by-degree-correct*(*3*)])
by (*auto simp*: *prod-eq-iff*)

lemma *tm-norm'-range*:

assumes $x \in_i$ *range-tm e t develops-at-within e a I num-params* (*tm-poly t*) $\leq$ *length I*
shows $x \in_i$ *range-tm e* (*tm-norm′ prec ord I a 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′*:
num-params (*tm-poly* (*tm-norm′ prec ord I a t*)) $\leq X$
if *num-params* (*tm-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 $x \in_i$ *range-tm e t develops-at-within e a I num-params* (*tm-poly t*) $\leq$ *length I*
shows $x \in_i$ *range-tm e* (*tm-norm prec ord I a 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 $x \in_i$ *range-tm e t*
shows $-x \in_i$ *range-tm e* (*tm-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-bound* (*tm-add t1 t2*) = *tm-bound t1* + *tm-bound t2*
by (*cases t1*; *cases t2*) *auto*

lemma *interval-of-add*: *interval-of* (*a* + *b*) = *interval-of a* + *interval-of b*
by (*auto intro*!: *interval-eqI*)

lemma *tm-add-range*:
$x + y \in_i$ *range-tm e* (*tm-add t1 t2*)
if $x \in_i$ *range-tm e t1*
$y \in_i$ *range-tm e t2*
proof −
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 $x \in_i$ *range-tm e t1*
assumes $y \in_i$ *range-tm e t2*
shows $x - y \in_i$ *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*: *set-of* (*interval-of y*) $\subseteq$ *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*: *set-of* (*interval-of y*) $\subseteq$ *set-of* (*real-interval Y*) **if** *y* $\in_r$ *Y*
using *that* **by** (*auto simp*: *set-of-eq*)

lemma *tm-mul-range*:
assumes $x \in_i$ *range-tm e t1*
assumes $y \in_i$ *range-tm e t2*
assumes *dev*: *develops-at-within e a I*
assumes *params*: *num-params* (*tm-poly t1*) $\leq$ *length I num-params* (*tm-poly t2*) $\leq$ *length I*
shows $x * y \in_i$ *range-tm e* (*tm-mul prec 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* = *TaylorModel p1 e1* **and** *t2-def*: *t2* = *TaylorModel p2 e2*
by (*auto simp*: *p1-def e1-def p2-def e2-def*)
from *params* **have** *params*: *num-params p1* $\leq$ *length I num-params p2* $\leq$ *length I*
by (*auto simp*: *p1-def p2-def*)
from *range-tmD*[*OF assms*(*1*)]
obtain *xe* **where** *x*: *x* = *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*: *y* = *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* = *insertion e* (*p1* $*_p$ *p2*) + (*xe* * *?y'* + *?x'* * *ye* + *xe* * *ye*)
by (*simp add*: *algebra-simps x y map-poly-homo-polymul*)
also have ... $\in_i$ *range-tm e* (*tm-mul prec 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*)) ≤ *x*
if *num-params* (*tm-poly t*) ≤ *x num-params p* ≤ *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* ≤ *1*
assumes *tg-def*: *e′ 0* ∈$_i$ *range-tm e tg*
assumes *dev*: *develops-at-within e a I* **and** *params*: *num-params* (*tm-poly tg*) ≤ *length I*
shows *insertion e′ p* ∈$_i$ *range-tm e* (*eval-poly-at-tm prec ord I a p tg*)
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* ∈$_i$ *range-tm e* (*tm-inc-err i t*)
if *x* ∈$_i$ *range-tm e t* + *real-interval i*
using *that*
by (*cases t*) (*auto simp*: *range-tm-def real-interval-plus ac-simps*)

lemma *num-params-tm-inc-err*: *num-params* (*tm-poly* (*tm-inc-err i t*)) ≤ *X*
if *num-params* (*tm-poly t*) ≤ *X*
using *that*
by (*cases t*) *auto*

lemma *num-params-tm-comp-le*: *num-params* (*tm-poly* (*tm-comp prec ord I a ga tf tg*)) ≤ *X*
if *num-params* (*tm-poly tf*) ≤ *max 1 X num-params* (*tm-poly tg*) ≤ *X*
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* ∈$_i$ *range-tm e′ tf*
assumes *tg-def*: *e′ 0* ∈$_i$ *range-tm e* (*tm-sub tg* (*tm-const ga*))
assumes *params*: *num-params* (*tm-poly tf*) ≤ *1 num-params* (*tm-poly tg*) ≤ *length I*
assumes *dev*: *develops-at-within e a I*
shows *x* ∈$_i$ *range-tm e* (*tm-comp prec ord I a ga tf tg*)
proof−
obtain *pf ef* **where** *tf-decomp*: *tf* = *TaylorModel pf ef* **using** *taylor-model.exhaust*
by *auto*
obtain *pg eg* **where** *tg-decomp*: *tg* = *TaylorModel pg eg* **using** *taylor-model.exhaust*
by *auto*

from *params* **have** *params*: *num-params pf* ≤ *Suc 0 num-params pg* ≤ *length I*
by (*auto simp*: *tf-decomp tg-decomp*)
from *tf-def* **obtain** *xe* **where** *x-def*: *x* = *insertion e′ pf* + *xe xe* ∈$_r$ *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*:
interval-of (*real-of-float* (*mid abs-bound*)) + *real-interval* (*centered abs-bound*) =
real-interval abs-bound
by (*auto simp*: *centered-def interval-eq-iff*)

lemmas [*simp del*] = *tm-abs.simps*
lemma *tm-abs-range*:
assumes *x*: *x* ∈$_i$ *range-tm e t*
assumes *n*: *num-params* (*tm-poly t*) ≤ *length I* **and** *d*: *develops-at-within e a I*
shows *abs x* ∈$_i$ *range-tm e* (*tm-abs prec I a t*)
proof−
obtain *p e* **where** *t-def*[*simp*]: *t* = *TaylorModel p e* **using** *taylor-model.exhaust*
by *auto*
define *bound* **where** *bound* = *compute-bound-tm 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* ≡ *Ivl 0* (*max* |*lower bound*| |*upper bound*|)
have *abs-bound*: |*x*| ∈$_r$ *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*)) ≤ *X* **if** *num-params* (*tm-poly t*) ≤ *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* ∈$_i$ *a* ⟹ *x* ∈$_i$ *sup a b*
and *in-interval-supI2*: *x* ∈$_i$ *b* ⟹ *x* ∈$_i$ *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* ∈$_i$ *range-tm e t1*
num-params (*tm-poly t1*) ≤ *length I develops-at-within e a I*
shows *x* ∈$_i$ *range-tm e* (*tm-union prec I a t1 t2*)
proof−
define *b1* **where** *b1* ≡ *compute-bound-tm prec I a t1*
define *b2* **where** *b2* ≡ *compute-bound-tm prec I a t2*
define *b-combined* **where** *b-combined* ≡ *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-def*: *p* = (*mid b-combined*)$_p$
and *e-def*: *e* = *centered b-combined*
by (*auto simp*: *Let-def b1-def b2-def b-combined-def interval-eq-iff*)
have *x* ∈$_r$ *b1*
by (*auto simp* : *b1-def intro*!: *compute-bound-tm-correct assms*)
then have *x* ∈$_r$ *b-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 $x \in_i$ *range-tm e t2*
num-params (*tm-poly t2*) $\leq$ *length I develops-at-within e a I*
shows $x \in_i$ *range-tm e* (*tm-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*:
num-params (*tm-poly* (*tm-union prec I a t1 t2*)) $\leq X$
if *num-params* (*tm-poly t1*) $\leq X$ *num-params* (*tm-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 $x \in_i$ *range-tm e t1*
assumes $y \in_i$ *range-tm e t2*
num-params (*tm-poly t1*) $\leq$ *length I*
num-params (*tm-poly t2*) $\leq$ *length I*
develops-at-within e a I
shows *min x y* $\in_i$ *range-tm e* (*tm-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 $x \in_i$ *range-tm e t1*
assumes $y \in_i$ *range-tm e t2*
num-params (*tm-poly t1*) $\leq$ *length I*
num-params (*tm-poly t2*) $\leq$ *length I*
develops-at-within e a I
shows *max x y* $\in_i$ *range-tm e* (*tm-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* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *float interval list* $\Rightarrow$ *float interval list* $\Rightarrow$ *floatarith* $\Rightarrow$ *taylor-model option* $\Rightarrow$ (*float interval* $\Rightarrow$ *bool*) $\Rightarrow$ *taylor-model option*
where *compute-tm-by-comp prec ord I a f g f-exists-on* = (

```
        case g
        of Some tg ⇒ (
          let gI = compute-bound-tm prec I a tg;
              ga = mid (compute-bound-tm prec a a tg)
          in if f-exists-on gI
            then map-option (λtf. tm-comp prec ord I a ga tf tg ) (tm-floatarith prec
ord [gI] [ga] f)
             else None)
        | - ⇒ None
      )
```

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

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

```
  | approx-tm prec ord I a (Sqrt f) env
        = compute-tm-by-comp prec ord I a (Sqrt (Var 0)) (approx-tm prec ord I a
f env) (λx. 0 < lower x)
  | approx-tm prec ord I a Pi env = Some (tm-pi prec)
  | approx-tm prec ord I a (Abs f) env
        = map-option (tm-abs prec I a) (approx-tm prec ord I a f env)
  | approx-tm prec ord I a (Min l r) env = (
        case (approx-tm prec ord I a l env, approx-tm prec ord I a r env)
        of (Some t1, Some t2) ⇒ Some (tm-min prec I a t1 t2)
         | - ⇒ None)
  | approx-tm prec ord I a (Max l r) env = (
        case (approx-tm prec ord I a l env, approx-tm prec ord I a r env)
        of (Some t1, Some t2) ⇒ Some (tm-max prec I a t1 t2)
         | - ⇒ None)
  | approx-tm prec ord I a (Powr l r) env = None — TODO
  | approx-tm prec ord I a (Floor l) env = None — TODO

lemma mid-in-real-interval: mid i ∈r i
  using lower-le-upper[of i]
  by (auto simp: mid-def set-of-eq powr-minus)

lemma set-of-real-interval-mono:set-of (real-interval x) ⊆ set-of (real-interval y)
  if set-of x ⊆ 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:
  tm-floatarith prec ord a b f = Some tf ⟹ num-params (tm-poly tf) ≤ 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 max-Var-floatarith f ≤ 1
  assumes a: a all-subset I
  assumes tx-range: x ∈i range-tm e tg
  assumes t-def: compute-tm-by-comp prec ord I a f (Some tg) c = Some t
  assumes f-deriv:
    ⋀x. x ∈r compute-bound-tm prec I a tg ⟹ c (compute-bound-tm prec I a tg)
⟹ isDERIV 0 f [x]
  assumes params: num-params (tm-poly tg) ≤ length I
    and dev: develops-at-within e a I
  shows interpret-floatarith f [x] ∈i range-tm e t
proof−
  from t-def[simplified, simplified Let-def]
  obtain 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]: ⋀x. 0 ≤ length x ⟹ (λx. interpret-floatarith f [x ! 0]) x = inter-
pret-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 ≤ X num-params (tm-poly t0) ≤ X
  shows num-params (tm-poly t) ≤ 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
= Some t ⟷
  (∃ z x2. t0 = Some x2 ∧
    tm-floatarith prec ord [compute-bound-tm prec I a x2]
      [mid (compute-bound-tm prec a a x2)] f =
      Some z
   ∧ tm-comp prec ord I a
      (mid (compute-bound-tm prec a a x2)) z x2 = t
   ∧ i (compute-bound-tm prec I a x2))
  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 ⋀tm. tm ∈ set env ⟹ num-params (tm-poly tm) ≤ length I
  shows num-params (tm-poly t) ≤ 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*$_i$ *range-tms e env*
assumes *devs*: *develops-at-within e a I*
assumes *env*: $\bigwedge$*tm. tm* ∈ *set env* ⟹ *num-params* (*tm-poly tm*) ≤ *length I*
shows *interpret-floatarith f xs* ∈$_i$ *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*: $t = tm\text{-}add\ t1\ t2$
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.prems*(1) *approx-tm.simps*(4) *map-option-eq-Some*)
have *t-def*: $t = tm\text{-}neg\ t1$
by (*metis Minus.prems*(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 option.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*: $t = tm\text{-}mul\ prec\ ord\ I\ a\ t1\ t2$
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*: $t = tm\text{-}pow\ prec\ ord\ I\ a\ tm\text{-}f\ k$
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: ⋀x. x ∈r (compute-bound-tm prec I a tf) ⟹
        0 < lower (compute-bound-tm prec I a tf) ∨ upper (compute-bound-tm prec
I a tf) < 0 ⟹
         isDERIV 0 (Inverse (Var 0)) [x]
    by (simp add: set-of-eq , safe, simp-all)
  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
      Inverse(1)[OF tf-def]
      Inverse(2)[unfolded approx-tm.simps tf-def]
      safe np devs]
  show ?case by simp
next
  case hyps: (Cos 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: (Arctan 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: (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: ⋀x. x ∈r compute-bound-tm prec I a tf ⟹
      0 < lower (compute-bound-tm prec I a tf) ⟹ isDERIV 0 (Ln (Var 0)) [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: ⋀x. x ∈r compute-bound-tm prec I a tf ⟹
      0 < lower (compute-bound-tm prec I a tf) ⟹ isDERIV 0 (Sqrt (Var 0))
[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*) = (λ*vs*. *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*) *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*) = (λ*vs*. *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* (λ(*i*, *ivl*). *tm-of-ivl-par i ivl*) (*zip* [*0*..<*length ivls*] *ivls*)

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

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

approx-slp′ p ord I a eas (r#xs)
}

lemma *mem-range-tms-Cons-iff*[*simp*]: *x#xs all-in$_i$ range-tms e (X#XS) ⟷ x ∈$_i$ range-tm e X ∧ xs all-in$_i$ range-tms e XS*
by (*auto simp*: *range-tms-def all-in-i-def nth-Cons split*: *nat.splits*)

lemma *approx-slp′-range*:
assumes *i*: *i all-subset I*
assumes *dev*: *develops-at-within e i I*
assumes *vs*: *vs all-in$_i$ range-tms e VS* (⋀*tm*. *tm ∈ set VS ⟹ num-params* (*tm-poly tm*) ≤ *length I*)
assumes *appr*: *approx-slp′ p ord I i ra VS = Some X*
shows *interpret-slp ra vs all-in$_i$ range-tms e X*
using *appr vs*
proof (*induction ra arbitrary*: *X vs VS*)
case (*Cons ra ras*)
from *Cons.prems*
obtain *a* **where** *a*: *approx-tm p ord I i ra VS = Some a*
and *r*: *approx-slp′ p ord I i ras* (*a # VS*) = *Some X*
by (*auto simp*: *bind-eq-Some-conv*)
from *approx-tm-range*[*OF i a Cons.prems*(*2*) *dev Cons.prems*(*3*)]
have *interpret-floatarith ra vs ∈$_i$ range-tm e a*
by *auto*
then have *1*: *interpret-floatarith ra vs#vs all-in$_i$ range-tms e* (*a#VS*)
using *Cons.prems*(*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.prems*)
qed *auto*

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

lemma *length-range-tms*[*simp*]: *length* (*range-tms e VS*) = *length VS*
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-in*$_i$ *range-tms e VS*
assumes *lens*: ⋀*tm*. *tm* ∈ *set VS* ⟹ *num-params* (*tm-poly tm*) ≤ *length vs*
shows *interpret-floatariths fas vs all-in*$_i$ *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-in*$_i$ *range-tms e XS* **by** (*auto simp*: *I-def*)
then have *take d* (*interpret-slp slp vs*) *all-in*$_i$ *range-tms e* (*take d XS*)
by (*auto simp*: *all-in-i-def range-tms-def*)
also note ‹*take d XS* = *X*›
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 ⇒ *'a interval* ⇒ *char list* ⇒ *char list*
is λ*p ivl s*. (*shows-string ''Interval'' o shows ivl*) *s* **.**

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

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

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

((*map-option* (*Inf-tm p params*)) *R*) *o shows* ′′)′′) ′′⏎′′)) *else* () *in case split-largest-interval X of* (*a*, *b*) ⇒
prove-pos prnt i p ord F (*a*#*b*#*XS*))

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

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

experiment begin

unbundle *floatarith-syntax*

abbreviation *schwefel* ≡
(*5.8806* / *10* ^ *10*) + (*Var 0* − (*Var 1*)^*e2*)^*e2* + (*Var 1* − *1*)^*e2* + (*Var 0* − (*Var 2*)^*e2*)^*e2* + (*Var 2* − *1*)^*e2*

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*)^*e2* ∗ *Var 3* +
4 ∗ *Var 0* ∗ *Var 2* ∗ (*Var 3*)^*e2* + *2* ∗ *Var 1* ∗ (*Var 3*)^*e3* + *4* ∗ *Var 0* ∗ *Var 2* + *4* ∗ (*Var 2*)^*e2* − *10* ∗ *Var 1* ∗ *Var 3* +
−*10* ∗ (*Var 3*)^*e2* + *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*)^*e2* + *2* ∗ (*Var 1*)^*e2* + *2* ∗ (*Var 2*)^*e2* + *2* ∗ (*Var 3*)^*e2* + *2* ∗ (*Var 4*)^*e2* + *2* ∗ (*Var 5*)^*e2* +
2 ∗ (*Var 6*)^*e2* − *Var 0*

end

end