Math.Comput.Sci. (2011) 5:437–467
DOI 10.1007/s11786-011-0102-5

Mathematics in Computer Science

A Validated Real Function Calculus

Pieter Collins · Milad Niqui · Nathalie Revol

Received: 15 November 2010 / Revised: 11 May 2011 / Accepted: 13 September 2011 / Published online: 15 November 2011

Abstract We present a framework for validated numerical computations with real functions. The framework is based on a formalisation of abstract data types for basic floating-point arithmetic, interval arithmetic and function models based on Banach algebra. As a concrete instantiation, we develop an elementary smooth function calculus approximated by sparse polynomial models. We demonstrate formal verification applied to validated calculus by a formalisation of basic arithmetic operations in a theorem prover. The ultimate aim is to develop a formalism powerful enough for reachability analysis of nonlinear hybrid systems.

Keywords Interval arithmetic · Floating point · Function calculus

Mathematics Subject Classification (2010) 65G20

1 Introduction

In this paper, we present a framework for developing and verifying validated numerical calculations with real functions. By a *validated* computation, we mean that the result is presented as a set of possible solutions which is guaranteed to contain the exact mathematical answer. By *verification*, we mean that the algorithms and implementation themselves are proven to be correct.

The scope of this paper is to consider a function calculus. We assume the existence of a floating-point number type supporting rounded arithmetical operations and develop algorithms for the basic operations on functions, including addition, multiplication, composition and comparison. We first give a mathematical definition of the algorithms

P. Collins (✉)
Maastricht University, Maastricht, The Netherlands
e-mail: pieter.collins@maastrichtuniversity.nl

M. Niqui (✉)
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
e-mail: M.Niqui@cwi.nl

N. Revol (✉)
INRIA, LIP (UMR 5668 CNRS, ENS de Lyon, INRIA, UCBL), Université de Lyon, Lyon, France
e-mail: Nathalie.Revol@ens-lyon.fr

used and prove their correctness, and then formalise the algorithms within the framework of the theorem-proving tool Coq [16] and use Coq to prove correctness.

In order to facilitate the verification procedure, we split the design into layers. In each layer, the concrete implementations are shown to satisfy abstract axioms, which are then used in the verification of the higher layers. Not only does this approach simplify the structure of the proofs, it also means that different implementations of the same abstract data type can easily be supported; one need only show that the new data type satisfies the required axioms, and it can then be used in subsequent computations without having to re-prove correctness of dependent algorithms. The lowest layer is that of floating-point arithmetic, for which we assume the existence of an implementation. The next layer is that of function algebras, for which we axiomatise abstract operations and show that our polynomial models satisfy the axioms. We also give examples of operations at a higher layer, in which we show how to solve algebraic and differential operations using purely the abstract operations at the numeric and function layers.

To place our goals in a consistent framework, we give an axiom-schema for validated computing, which is simply a formalisation of the requirement that the result of any computation is a set which is guaranteed to contain the mathematically exact answer.

The original motivation for this paper was to support the development and verification of ARIADNE, a tool for the analysis of nonlinear hybrid systems. Since hybrid systems are frequently used in modelling safety-critical embedded-systems, it is vital to be certain that the results of the numerical calculations are correct, notably that round-off and truncation errors are properly accounted-for. The tool needs to evaluate arithmetic, algebraic and transcendental functions, find the solution of ordinary differential equations and solve algebraic equations, and each of these operations needs to be performed over a range of different parameters and states. The results of this paper are the first part of such a verification, in which we analyse the algorithms for the functional operations needed.

The paper is organised as follows. In Sect. 2 we give an axiom-schema for validated computation. In Sect. 3 we apply this axiom-schema to floating-point and interval arithmetic, and in Sect. 4 to various abstract algebras, including function algebras. In Sect. 5 we develop a concrete representation of a real function algebra based on polynomial models. In Sect. 6 we show how to compute solutions to differential and algebraic equations using only the abstract operations on function algebras. In Sect. 7, we show how the correctness of the operations on polynomial models can be proved using the theorem-prover Coq. Finally, in the appendix we give real C++ and Coq code used, respectively, to implement and validate the polynomial models.

2 Axiom Schema for Validated and Approximate Computation

Without attempting to present a complete list, we distinguish five distinct paradigms in designing numeric libraries.

Axiomatic The formal mathematical notion of the correct answer, not necessarily computable.
Symbolic The exact answer x is given with a finite symbolic description.
Effective Given an accuracy n, an approximation a to the result can be output with error 2^{-n}. However, to specify the answer completely requires an infinite amount of data.
Validated A set $\hat{x}$ of possible exact answers. May be specified by a pair (a, e), where a is an approximation to the exact answer x with error e, or an over-approximation u satisfying $u > x$.
Approximate An approximation a is given to the exact answer.

In general, the lower an item is on the list above, the less information is provided about the solution, but the faster the computations can be performed (in general). Most numerical methods in use today use the *approximate* computational paradigm; no guarantees are given about the accuracy of the result, and the user must *trust* the results of the computations. The *symbolic* paradigm is implemented in packages for symbolic algebra. The *effective* and *validated* paradigms are closely related; in both cases a set of values is given which is guaranteed to contain the exact answer; in the validated paradigm, a single approximation is given, whereas in the effective paradigm, the result

can be given to an arbitrary specified accuracy. The effective paradigm is usually more appropriate for functional languages, and the validated paradigm for imperative languages.

In many cases, it is useful to mix these paradigms; for example, symbolic computation may be used to simplify the problem before reverting to approximate computation, or the results of an approximate computation may be used to precondition or hot-start a validated computation.

Most numerical computation is currently performed using approximate operations. Given an uncountable set X, we represent selected elements of X by elements of a countable set $\widetilde{X}$. We write $\tilde{x} \vdash x$ if $\tilde{x} \in \widetilde{X}$ denotes $x \in X$, and set $\imath(\tilde{x}) = x$. Frequently, $\widetilde{X}$ will be a subset of X, and $\imath(\cdot)$ an injection, but there are important examples where this is not the case, e.g. IEEE floating-point formats have separate elements $+0.0$ and -0.0 denoting the real number 0. By a slight abuse of notation, we henceforth write $\tilde{x} = x$ rather than $\tilde{x} \vdash x$.

An *approximate* version of an operator $\mathrm{op} : X_1 \times \cdots \times X_n \to X_0$ is a function $\widetilde{\mathrm{op}} : \widetilde{X}_1 \times \cdots \times \widetilde{X}_n \to \widetilde{X}_0$ such that

$$\imath(\tilde{x}_i) \approx x_i \quad \text{for } i = 1, \ldots, n \text{ implies } \imath(\widetilde{\mathrm{op}}(\tilde{x}_1, \ldots, \tilde{x}_n)) \sim \mathrm{op}(x_1, \ldots, x_n). \tag{2.1}$$

We write $\tilde{x} \approx x$ to suggest $\imath(\tilde{x})$ is 'very close' to x, and $\tilde{x} \sim x$ if $\imath(\tilde{x})$ is merely 'close' to x; formally however there are no restrictions. In other words, there is the argument is close to the exact argument, then the result 'should' be close to the exact result. Formally, there are no guarantees on the result, which may in some cases be so inaccurate as to be useless or even misleading.

When performing validated computations on an uncountable set X, we aim to compute a subset of X which is guaranteed to contain the exact result. Hence the result of a validated computation will be an element $\hat{x}$ of a countable subset $\widehat{X}$ of $\mathcal{P}(X)$, and must satisfy $\hat{x} \ni x$, where x is the exact result. Alternatively, we can take $\widehat{X}$ to be a general countable set, and define a *modelling* or *realisation* relation $\models$ on $\widehat{X} \times X$, writing $\hat{x} \models x$ rather than $\hat{x} \ni x$.

Definition 2.1 (*Modelling relations*) Let X be a set, $\widehat{X}$ a countable set, and $\models$ be a binary relation (called 'realises' or 'models') between $\widehat{X}$ and X such that for any $x \in X$, there exists $\hat{x} \in \widehat{X}$ such that $\hat{x} \models x$. Frequently, $\widehat{X}$ will consist of subsets of X, with $\hat{x} \models x \iff \hat{x} \ni x$; in this case we use $\ni$ and $\models$ interchangeably.

We impose the following *axiom-schema* for validated computation:

Definition 2.2 (*Axiom-schema for validated computation*) Let X_i be a class of mathematical objects, and $\widehat{X}_i$ be a countable set, and a $\models_i$ a binary relation between $\widehat{X}_i$ and X_i. Then an operator $\widehat{\mathrm{op}} : \widehat{X}_1 \times \cdots \times \widehat{X}_n \to \widehat{X}_0$ is a *validated* version of $\mathrm{op} : X_1 \times \cdots \times X_n \to X_0$ if

$$\hat{x}_i \models_i x_i \quad \text{for } i = 1, \ldots, n \text{ implies } \widehat{\mathrm{op}}(\hat{x}_1, \ldots, \hat{x}_n) \models_0 \mathrm{op}(x_1, \ldots, x_n). \tag{2.2}$$

Note that the relations $\hat{x}_i \models_i x_i$ are merely used to define the axioms, and are *not* required to be computable.

An important property is when one model gives more information than another.

Definition 2.3 (*Refinement*) Let $\models$ be a modelling between $\widehat{X}$ and X, and $\hat{x}_1, \hat{x}_2 \in \widehat{X}$. We say $\hat{x}_1$ *refines* $\hat{x}_2$, denoted $\hat{x}_1 \prec \hat{x}_2$ if $\hat{x}_1 \models x \implies \hat{x}_2 \models x$.

Notation When dealing with *mathematical* statements, given an object x of a set X, we will use $\tilde{x}$ to denote an approximation to x in $\widetilde{X}$, and $\hat{x}$ to denote a validated model of x in $\widehat{X}$. When dealing with *implementations*, we will use $\mathtt{x}$ to denote a concrete object of class $\mathtt{X}$, and $\imath(\mathtt{x})$ or $\overline{\mathtt{x}}$ to denote the mathematical object it represents.

Given some guarantees on the result, approximate computation on countable sets can be used to implement validated computations. Consider countable sets $\widetilde{X}_i$ such that each $x_i \in \widetilde{X}_i$ represents an element $\imath(x_i)$ of X_i. Given a function $f : X_1 \times \cdots \times X_n \to X_0$, a *version* of f is a function $f_\star : \widetilde{X}_1 \times \cdots \times \widetilde{X}_n \to \widetilde{X}_0$ satisfying some axiom-schema giving conditions on the result. We are interested in versions satisfying following axiom schema:

Exact $f_e(\tilde{x}_1, \ldots, \tilde{x}_n) = f(x_1, \ldots, x_n)$.

Upward/downward If X_0 is a partially ordered set, $f_d(\tilde{x}_1, \ldots, \tilde{x}_n) \leq f(x_1, \ldots, x_n) \leq f_u(\tilde{x}_1, \ldots, \tilde{x}_n)$.
Nearest If X_0 is a metric space and $\widetilde{X}_0$ is compact, $\forall \tilde{x}_0 \in \widetilde{X}_0$, $d(f_n(\tilde{x}_1, \ldots, \tilde{x}_n), f(x_1, \ldots, x_n)) \leq d(\tilde{x}_0, f(x_1, \ldots, x_n))$.
Approximate $f_a(\tilde{x}_1, \ldots, \tilde{x}_n) \approx f(x_1, \ldots, x_n)$.

Note that in an approximate version f_a of a function f, no guarantees at all are given on the error.

3 Floating-Point and Interval Arithmetic

In this section we describe axioms for floating-point and interval arithmetic, which describe approximate and validated real numbers.

3.1 Floating-Point Data Type

We describe our axiomatisation of an abstract data type for the floating-point numbers: we introduce a type $\mathbb{F}$ (represented in code by the class `Float`) together with some of the basic IEEE-754 operations. Let us point out the fact that operations with various rounding mode are directly added to the signature of our abstract data type. For instance there are three addition operations $+_u$, $+_d$ and $+_n$ for summing up two floating-point numbers using, respectively, upwards, downwards and to-nearest rounding. In future work, we plan to extend our axiomatisation to a library compatible with IEEE-754 specification of the basic operations $(+, \times, -, \div)$ and the recommended elementary functions. At the moment we only handle operations that are necessary in formalising the proofs for arithmetic on function approximations.

Our axiomatisation includes a type $\mathbb{F}$ together with the binary operations $+_u, +_d, +_n, \times_u, \times_d, \times_n$, the exact unary operation $-_e$ and the exact constants 0_e and 1_e. We additionally include upward division by natural numbers $\div_u$, an exact absolute value function $|\cdot|_e$ and an exact less-than-or-equal-to comparison operator $\leq_e$. Additional operations could of course be axiomatised in a similar way, but these are the only operations we need to develop the calculus of polynomial models. Note that currently $\mathbb{F}$ is not handled as a bounded set and that there is no reciprocal and no symbols for `NaN` and $\pm$`Inf`. (These will be added in the future.)

The axiomatisation is based on the existence of a function $\iota : \mathbb{F} \longrightarrow \mathbb{R}$. The rest of the axioms will govern the arithmetic operations in $\mathbb{F}$ and are stated in terms of the injection ι.

The complete axioms are given in Sect. 7.1, and follow the schema shown below, with $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ taken to be elements of $\mathbb{F}$.

Constants $\iota(0_e) = 0$ and $\iota(1_e) = 1$.
Exact $\forall \mathtt{x},\ \iota(-_e \mathtt{x}) = -\iota(\mathtt{x})$ and $\iota(|\mathtt{x}|_e) = |\iota(\mathtt{x})|$.
Downward $\forall \mathtt{x}, \mathtt{y},\ \iota(\mathtt{x} \star_d \mathtt{y}) \leq \iota(\mathtt{x}) \star \iota(\mathtt{y})$.
Upward $\forall \mathtt{x}, \mathtt{y},\ \iota(\mathtt{x} \star_u \mathtt{y}) \geq \iota(\mathtt{x}) \star \iota(\mathtt{y})$.
Nearest $\forall \mathtt{x}, \mathtt{y}, \mathtt{z},\ d(\iota(\mathtt{x} \star_n \mathtt{y}), \iota(\mathtt{x}) \star \iota(\mathtt{y})) \leq d(\iota(\mathtt{z}), \iota(\mathtt{x}) \star \iota(\mathtt{y}))$.
Division $\forall \mathtt{x}, m\ m > 0 \implies \iota(\mathtt{x} \div_u m) \leq \iota(\mathtt{x}) \div 2$.
Comparison $\forall \mathtt{x}, \mathtt{y},\ \mathtt{x} \leq_e \mathtt{y} \iff \iota(\mathtt{x}) \leq \iota(\mathtt{y})$.

Note that here the distance function d is evaluated in $\mathbb{R}$, and has no computational meaning. Subtraction $-_{u/d/n}$ can be defined in terms of the primitives $+_{u/d/n}$ and $-_e$. The IEEE-754 specifies two representations $\pm\mathtt{0}$ of the mathematical 0; either can be used as the constant 0_e in our axiomatisation.

We will later need the following result:

Lemma 3.1 *For any binary operation $\star$ on $\mathbb{R}$, we have*

$$2d(\iota(x \star_n y), \iota(x) \star \iota(y)) \leq \iota((x \star_u y) -_u (x \star_d y))$$

and

$$d(\iota(x \star_n y), \iota(x) \star \iota(y)) \leq \iota(((x \star_u y) -_u (x \star_d y)) \div_u 2).$$

Proof By the axiom-schema for nearest rounding, we have $d(x \star_n y, x \star y) \leq d(x \star_{u/d} y, x \star y)$. Taking the definition $d(x, y) = |x - y|$, and the axioms $|x| = x$ for $x \geq 0$ and $|x| = -x$ for $x \leq 0$, combined with the axiom-schema for upward/downward rounding, we have $d(x \star_u y, x \star y) = |x \star_u y - x \star y| = x \star_u y - x \star y$ and $d(x \star_d y, x \star y) = |x \star_d y - x \star y| = -(x \star_d y - x \star y) = x \star y - x \star_d y$. Hence $2d(x \star_n y, x \star y) \leq d(x \star_u y, x \star y) + d(x \star_d y, x \star y) = (x \star_u y - x \star y) + (x \star y - x \star_d y) = (x \star_u y) - (x \star_d y) \leq (x \star_u y) -_u (x \star_d y)$. □

In fact, the above result holds for any operation defined on a partially ordered metric space which has approximate versions that satisfy the upward/downward and nearest axiom schemes.

The upper reals The following statement asserts that the positive floating-point numbers with upper-rounded addition and multiplication $(\mathbb{F}^+, 0_e, 1_e, +_u, \times_u)$ form a validated implementation of $(\mathbb{R}^+, 0, 1, +, \times)$ under the modelling relationship $\mathtt{x} \models_{\geq} x \iff \iota(\mathtt{x}) \geq x$.

Theorem 3.2 *Consider the relationship $\mathbb{F} \models_{\geqslant} \mathbb{R}^+$ defined by $a \models_{\geqslant} x \iff \iota(a) \geq x$. Then $+_u$ and $\times_u$ are validated versions of $+$ and $\times$.*

Proof Since $+$ and $\times$ are each monotonic in both variables, we have $\iota(\mathtt{x}) \geq x$ and $\iota(\mathtt{y}) \geq y$ implies $\iota(\mathtt{x} +_u \mathtt{y}) \geq \iota(\mathtt{x}) + \iota(\mathtt{y}) \geq x + y$ and $\iota(\mathtt{x} \times_u \mathtt{y}) \geq \iota(\mathtt{x}) \times \iota(\mathtt{y}) \geq x \times y$. Hence $\mathtt{x} +_u \mathtt{y} \models_{\geqslant} x + y$ and $\mathtt{x} \times_u \mathtt{y} \models_{\geqslant} x \times y$ as required. □

3.2 Interval Arithmetic

Let $\mathbb{X}$ be a countable subtype of $\mathbb{R}$. Then an *interval* in $\mathbb{R}$ with endpoints in $\mathbb{X}$ is represented by a pair $\langle l, u \rangle$ with $l, u \in \mathbb{X}$. For $I \in \mathbb{I}$ and $x \in \mathbb{R}$ we take $I \ni x$ to mean $\iota(l) \leq x \leq \iota(u)$, where $\iota(\cdot)$ denotes the mapping of $\mathbb{X}$ into $\mathbb{R}$. In Ariadne, the type of intervals of real numbers is implemented a class `Interval` whose endpoints are elements of the class `Float`.[1]

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, an *interval extension* $[f]$ or $\hat{f}$ of f is simply a validated version of f. In other words:

Axiom 3.3 (*Interval extension*) Let $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a continuous function. Then a function $[f] : \mathbb{I}^n \longrightarrow \mathbb{I}$ is an interval extension of f if whenever $(x_1, \ldots, x_n) \in \mathbb{R}^n$ are such that $x_i \in I_i$ for all $i = 1, \ldots, n$, then

$$f(x_1, \ldots, x_n) \in [f](I_1, \ldots, I_n).$$

It is well-known that the rounded floating-point operations can be used to implement interval arithmetic. Denoting the interval extension of $\star$ by $\widehat{\star}$, we have:

Negation $\widehat{-}\,[l, u] = [-_e u, -_e l]$

Addition $[l_1, u_1] \mathbin{\widehat{+}} [l_2, u_2] = [l_1 +_d l_2, u_1 +_u u_2]$

Subtraction $[l_1, u_1] \mathbin{\widehat{-}} [l_2, u_2] = [l_1 -_d u_2, u_1 -_u l_2]$

Multiplication
$$[l_1, u_1] \mathbin{\widehat{\times}} [l_2, u_2] = [\min_e\{l_1 \times_d l_2, l_1 \times_d u_2, u_1 \times_d l_2, u_1 \times_d u_2\}, \max_e\{l_1 \times_u l_2, l_1 \times_u u_2, u_1 \times_u l_2, u_1 \times_u u_2\}]$$

Reciprocal $\widehat{1/}\,[l, u] = [1_e \div_d u, 1_e \div_u l]$ if $0 \notin [l, u]$, otherwise $[-\infty, +\infty]$.

Multiplication can be implemented more efficiently by first testing the signs of the l_i, u_i.

[1] More generally, it would be possible to use a template `Interval<X>` to denote intervals with endpoints in `X`. The parameter `X` can be instantiated by instances of `Float` or `Rational`. Instantiating by floats leads to a more *efficient* numerics library while instantiating by rational numbers results in a better *accuracy*.

4 Validated Function Algebras

In this section we discuss the various operations needed for a validated function calculus, and introduce the algebraic structures which formalise these operations, together with their validated counterparts. In an implementation, the algebraic structures themselves provide abstract interfaces which can be implemented in various ways by concrete data types for validated and approximate computations.

4.1 (Unital) Banach Algebra

An *algebra* over $\mathbb{R}$ is a set $\mathbb{A}$ with operations scalar multiplication $\cdot : \mathbb{R} \times \mathbb{A} \to \mathbb{A}$, addition $+ : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ and multiplication $\times : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ satisfying standard axioms. Note that negation can be defined using scalar multiplication as $-a = (-1) \cdot a$. An algebra is *associative* if multiplication is associative, and *commutative* if additionally multiplication is commutative. In a validated algebra $\widehat{\mathbb{A}}$, the scalar multiplication operator can be taken to be a function $\mathbb{I} \times \widehat{\mathbb{A}} \to \widehat{\mathbb{A}}$, i.e. we use $\mathbb{I}$ for $\hat{\mathbb{R}}$.

A *unital algebra* is an algebra with a unit $\mathbf{1}$ satisfying $\mathbf{1} \times a = a \times \mathbf{1} = a$ for all $a \in \mathbb{A}$. For a function algebra with pointwise operations, the unit is given by the constant function with value 1. The real numbers embed in a unital algebra by taking $c \mapsto c \cdot \mathbf{1}$. In a validated unital algebra, the embedding $\mathbb{R} \to \mathbb{A}$ should be such that $\hat{\mathbf{1}} \models \mathbf{1}$.

In a *normed algebra*, every element a has a *norm*, denoted $||a||$ or $\mathrm{nrm}(a)$ satisfying $||s \cdot a|| = |s|||a||$, $||a_1 + a_2|| \leq ||a_1|| + ||a_2||$ and $||a_1 \times a_2|| \leq ||a_1||\,||a_2||$. In a unital normed algebra, we additionally have $||\mathbf{1}|| = 1$. In applications, it is usually only necessary to compute an *upper* bound for the norm, hence for an *effective* normed algebra, the norm should be a computable function $A \to \mathbb{R}^+_>$, where $\mathbb{R}^+_>$ denotes the positive real numbers with the upper-topology/representation. In a validated normed algebra, we need only define a function $||\cdot||_u$ or $\mathrm{nrm}_u : \mathbb{A} \to \mathbb{F}$ such that $\mathrm{nrm}_u(x) \geq ||a||$ for any a.

Finally, as well as the unital element, we need to be able to define the unit ball as an element of the validated version of the algebra. This means, we require an element $\hat{E} \in \widehat{\mathbb{A}}$ such that $||a|| \leq 1 \implies a \in \hat{E}$, or equivalently, $\hat{E} \supset E$ where $E = \{a \in \mathbb{A} \mid ||a|| \leq 1\}$.

To summarise, in a validated normed unital Banach algebra $\widehat{\mathbb{A}}$, we have operations:

Definition 4.1 (*Validated Unital Banach algebra*)

Unit $\hat{I} \in \widehat{\mathbb{A}}$ such that $\mathbf{1} \in \hat{I}$.
Ball $\hat{E} \in \widehat{\mathbb{A}}$ such that $B(\mathbf{0}, 1) \subset \hat{E}$.
Scale $\widehat{\cdot} : \mathbb{I} \times \widehat{\mathbb{A}} \to \widehat{\mathbb{A}}$.
Addition $\widehat{+} : \widehat{\mathbb{A}} \times \widehat{\mathbb{A}} \to \widehat{\mathbb{A}}$.
Multiplication $\widehat{\times} : \widehat{\mathbb{A}} \times \widehat{\mathbb{A}} \to \widehat{\mathbb{A}}$.
Norm $||\cdot||_u : \widehat{\mathbb{A}} \to \mathbb{F}$.

Alternatively, the **Unit** and **Scale** operations can be replaced with the operation

Constant $c : \mathbb{I} \to \widehat{\mathbb{A}}$.

The main use of the unital Banach algebra abstraction is that we can evaluate analytic functions on a Banach algebra.

Example The exponential function is defined by $\exp(x) = \sum_{n=0}^{\infty} x^n/n!$. We therefore have $||\exp(a) - \sum_{n=0}^{N} a^n/n!|| = ||\sum_{n=N+1}^{\infty} a^n/n!|| \leq \sum_{n=N+1}^{\infty} ||a||^n/n!$. Hence $\exp(a) \in \sum_{n=0}^{N} a^n/n! + B(\mathbf{0}, \sum_{n=N+1}^{\infty} ||a||^n/n!)$. Taking validated operations, we obtain

$$\exp(\hat{a}) = \hat{I} \mathbin{\widehat{+}} \hat{a} \mathbin{\widehat{+}} \frac{1}{2} \mathbin{\widehat{\cdot}} \hat{a} \mathbin{\widehat{\times}} \hat{a} \mathbin{\widehat{+}} \cdots \mathbin{\widehat{+}} \frac{1}{N!} \mathbin{\widehat{\cdot}} \hat{a}^N \mathbin{\widehat{+}} \left(\sum_{n=N+1}^{\infty} ||a||^n/n! \right) \cdot \widehat{E} \supset \sum_{n=0}^{N} \frac{a^n}{n!} + B\left(\mathbf{0}, \sum_{n=N+1}^{\infty} ||a||^n/n!\right)$$

where $\sum^u$ denotes summation with upward rounding.

If multiplication is commutative (as is the case with a function algebra), we can choose a scalar c minimising $||a - c||$. Then $\exp(a) = \exp(c)\exp(a - c)$ has faster convergence. Additionally, we have $\exp(a) = \exp(a/2^n)^{2^n}$, again yielding faster convergence.

Example Similarly, we can consider the reciprocal function. For simplicity, we compute $1/(1-x) = \sum_{n=0}^{\infty} x^n = \sum_{n=0}^{N-1} x^n + x^N(\sum_{n=0}^{\infty} x^n)$. Hence the norm of $1/(1-x) - \sum_{n=0}^{N-1} x^n$ is bounded by $||x^N||/(1-||x||) \leq ||x||^N/(1-||x||)$ if $||x|| < 1$.

Note that it may be the case that the computed $||\hat{x}||_u > 1$ even though $||x|| < 1$ whenever $\hat{x} \models x$. In this case, it is impossible to compute $1/(1-\hat{x})$, even though $1/(1-x)$ is defined for all x.

4.2 Evaluation Algebra

Recall from Definition 2.2 that the axiom for a validated version $\hat{f}$ of a function $f : X_1 \times \cdots \times X_n \to Y$ is that $\hat{f} : \widehat{X}_1 \times \cdots \times \widehat{X}_n \to \widehat{Y}$ satisfies $\hat{f}(\hat{x}_1, \ldots, \hat{x}_n) \models f(x_1, \ldots, x_n)$ whenever $\hat{x}_i \models x_i$ for all $i = 1, \ldots, x_n$. In practice, we often wish to define a type modelling functions $X_1 \times \cdots \times X_n \to Y$ without giving a canonical evaluation operation. For example, there are many interval evaluation algorithms for polynomial functions, some of which are faster, others more accurate.

Let F denote the type of continuous functions $X_1 \times \cdots \times X_n \to Y$. Define the *evaluation* operator $\text{eval} : F \times X_1 \times \cdots \times X_n \to Y$ by

$$\text{eval}(f, x_1, \ldots, x_n) = f(x_1, \ldots, x_n). \tag{4.1}$$

Let $\widehat{F}$ be a countable set representing validated functions with a modelling relation $\models$ between $\widehat{F}$ and F. By applying the axiom-schema for validated computation given in Definition 2.2, we find that a validated version of the evaluation operator must satisfy

$$\widehat{\text{eval}}(\hat{f}, \hat{x}_1, \ldots, \hat{x}_n) \models f(x_1, \ldots, x_n) \quad \text{whenever} \quad \hat{f} \models f \quad \text{and} \quad \hat{x}_i \models x_i \quad \text{for} \quad i = 1, \ldots, n. \tag{4.2}$$

This allows for different implementations of the evaluation operator on the same validated function type. Taking $\widehat{Y}$ to be a subset of Y and $\widehat{X}_i$ to be subsets of X_i, we equivalently have

$$\widehat{\text{eval}}(\hat{f}, \hat{x}_1, \ldots, \hat{x}_n) \supset \{f(x_1, \ldots, x_n) \mid f \models \hat{f} \quad \text{and} \quad x_i \in \hat{x}_i \quad \text{for} \quad i = 1, \ldots, n\}$$

The conditions for a validated version $\widehat{\text{op}}$ on functions of an operator op, and of the composition operator, are also easily deduced from the definitions.

The observant reader will have noticed that we have two notions of a validated version of a function $f : X_1 \times \cdots \times X_n \to Y$. The first can be considered a canonical notion; a validated version of f is a function $\hat{f} : \widehat{X}_1 \times \cdots \times \widehat{X}_n \to \widehat{Y}$ such that $\hat{f}(\hat{x}_1, \ldots, \hat{x}_n) \models f(x_1, \ldots, x_n)$ whenever $\hat{x}_i \models x_i$ for $i = 1, \ldots, n$. A second possibility is to define a custom type $\widehat{F}$ and a custom modelling relation $\models$ between $\widehat{F}$ and the type F of (continuous) functions $X_1 \times \cdots \times X_n$. In this latter case, we should also define at least one custom evaluation operator, whereas in the former case, there is a canonical evaluation operator given by

$$\text{eval}(\hat{f}, \hat{x}_1, \ldots, \hat{x}_n) = \hat{f}(\hat{x}_1, \ldots, \hat{x}_n) \tag{4.3}$$

which satisfies (4.2) by definition. In the rest of this paper, we shall denote by single italic letters $\hat{f}$ functions in a custom function space $\widehat{F}$, and by roman abbreviations $\widehat{\text{op}}$ validated versions of functions $\widehat{X}_1 \times \cdots \times \widehat{X}_n \to \widehat{Y}$.

As well as the evaluation operator, there are other important operations on function spaces. Given functions $f_j : X_1 \ldots, X_n \to Y_j$ and an operator $\text{op} : Y_1 \times \cdots \times Y_m \to Z$, we define $\text{op}(f_1, \ldots, f_m) : X_1 \times \cdots \times X_m \to Z$ pointwise by

$$\text{op}(f_1, \ldots, f_m)(x_1, \ldots, x_n) = \text{op}(f(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)). \tag{4.4}$$

For example, taking op to be addition $\text{add}(y_1, y_2) = y_1 + y_2$, we obtain

$$\text{add}(f_1, f_2)(x_1, \ldots, x_n) = (f_1 + f_2)(x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) + f_2(x_1, \ldots, x_n).$$

We can also define a *composition* operator on function spaces by

$$\mathrm{comp}(g, f_1, \ldots, f_m)(x_1, \ldots, x_n) = g(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)) \tag{4.5}$$

Finally, we can define constant functions

$$\mathrm{cnst}(y)(x_1, \ldots, x_n) = y \tag{4.6}$$

and, if $X_i = Y$ also *projection* or *coordinate* functions

$$\mathrm{proj}[i](x_1, \ldots, x_n) = x_i \tag{4.7}$$

We call the algebraic structure containing constant, projection, evaluation and composition functions, as well as operators on functions, an *evaluation algebra*. The main operations on an evaluation algebra together with their properties are:

Definition 4.2 (*Validated evaluation algebra*)

Evaluation $\widehat{\mathrm{eval}} : \widehat{F} \times \widehat{X}_1 \times \cdots \times \widehat{X}_n \to \widehat{Y}$ satisfying $\widehat{\mathrm{eval}}(\hat{f}, \hat{x}_1, \ldots, \hat{x}_n) \models f(x_1, \ldots, x_n)$ whenever $\hat{f} \models f$ and $\hat{x}_i \models x_i$.
Operators $\widehat{\mathrm{op}} : \widehat{F}_1 \times \cdots \times \widehat{F}_n$ satisfying $\widehat{\mathrm{op}}(\hat{f}_1, \ldots, \hat{f}_n) \models \mathrm{op}(f_1, \ldots, f_n)$ for any $\mathrm{op} : Y_1 \times \cdots \times Y_n$. ; necessarily then $\widehat{\mathrm{op}}(\hat{f}_1, \ldots, \hat{f}_n)(\hat{x}) \models \mathrm{op}(f_1(x), \ldots, f_n(x))$
Composition $\widehat{\mathrm{comp}} : \widehat{F} \times \widehat{G}_1 \times \cdots \times \widehat{G}_n \to \widehat{H}$ satisfying $\widehat{\mathrm{comp}}(\hat{f}, \hat{g}_1, \ldots, \hat{g}_n) \models f \circ (g_1, \ldots, g_n)$ whenever $\hat{f} \models f$ and $\hat{g}_i \models g_i$.

We also often use the following constructors

Constants $\widehat{\mathrm{cnst}} : \widehat{Y} \to \widehat{F}$ such that $\widehat{\mathrm{cnst}}(\hat{y})(x_1, \ldots, x_n) = \hat{y}$.
Projection If the type Y is the same as X_i, also $\widehat{\mathrm{proj}}[i] = \hat{\pi}_i \in \widehat{F}$ given by $\widehat{\mathrm{proj}}[i](\hat{x}_1, \ldots, \hat{x}_n) = \hat{x}_i$.

4.3 Differential Algebra

A *differential algebra* is an algebra A with a linear *differential* operator $\partial : A \to A$ satisfying the *Leibniz rule* $\partial(a_1 \times a_2) = \partial a_1 \times a_2 + a_1 \times \partial a_2$. In a unital differential algebra, we always have $\partial(1) = 0$. Often, we have several differentials ∂_i such that $\partial_i \partial_j = \partial_j \partial_i$. We say an element c is *constant* if $\partial_i c = 0$ for all i; in most important cases the constant elements are the scalar multiples of the unit. We say an element x_i is a *coordinate* if $\partial_i x_i = 1$ and $\partial_j x_i = 0$ for $i \neq j$.

An *antidifferential* $\int$ is a right inverse to the differential ∂, so is defined by $\partial \int f = f$. In the presence of multiple commuting differentials, we require $\partial_j \int_i 1 = 0$ whenever $i \neq j$, so $\int_i 1$ is a coordinate function x_i.

When working with the function algebra $C^\infty(\mathbb{R}^n; \mathbb{R})$, canonical antidifferentials are given by the definite integrals

$$\left(\textstyle\int_i f\right)(x_1, \ldots, x_n) = \displaystyle\int_0^{x_i} f(x_1, \ldots, x_{i-1}, \xi, x_{i+1}, \ldots, x_n)\, d\xi.$$

Further, the integral is defined on $C(\mathbb{R}^n; \mathbb{R})$ and is continuous in the topology of uniform convergence on compact sets.

While differentials of functions defined symbolically are easy to compute using the Leibniz rule, the derivative is discontinuous in the uniform topology. Consider an element $\hat{f}$ of a validated function type with $\hat{f} \models f \iff |\hat{f}(x) - f(x)| \leq \epsilon$ for $x \in [-1, +1]$. By taking $f(x) = \epsilon \sin(x/\delta)$ we have $f'(x) = (\epsilon/\delta)\cos(x/\delta)$, so $||f'|| = \epsilon/\delta$ which can be made arbitrarily large. However, 'on average', the fluctuations of f are approximately zero, so the zero function is a good *weak* derivative of f. We now formalise this notion in a way which can be used in the solution of algebraic equations.

We first need to define a special class of modelling relations. A function model $\hat{f} \in \widehat{F}$ is a *uniform* function model if there exists an exact function $\tilde{f}$ and a constant $\bar{e} \geq 0$ such that $\hat{f} \models f \iff \sup\{d(f(x), \tilde{f}(x)) \mid x \in D\} \leq \bar{e}$. We say that $\tilde{f}$ is the *centre* of $\hat{f}$.

Definition 4.3 (*Weak derivative*) Let $\hat{f}$ be a uniform function model and $\tilde{f}$ be the centre of $\hat{f}$. Then $\hat{g}$ is *weak derivative* of $\hat{f}$ if $\hat{g} \models \partial \tilde{f}/\partial x_i$.

In other words, $\hat{g}$ is a validated version of the exact derivative of $\tilde{f}$, where $\tilde{f}$ is the centre of the uniform model $\hat{f}$. If $\hat{f}$ were to consist only of functions of the form $f(x) = \tilde{f}(x) + c$ for $|c| \leq \bar{e}$, then $\hat{g}$ would model $\partial f/\partial x_i$ for all $f \dashv\models \hat{f}$. Since $\hat{f}$ consists of functions $f(x) = \tilde{f}(x) + \varepsilon(x)$ where ε is merely continuous, $\partial f/\partial x_i$ may have arbitrarily large values. However, $\hat{g}$ captures some "average" variation of f, in the sense that there is antiderivative $\int^{x_i} \hat{g} dx_i$ which is uniformly $\bar{e}$-close to f.

Note that the condition of Definition 4.3 is stronger than saying $\exists f \dashv\models \hat{f}, \hat{g} \models \partial f/\partial x_i$, but weaker than saying $\forall f \dashv\models \hat{f}, \hat{g} \models \partial f/\partial x_i$. In Sect. 6.2 we will show how to use the weak derivative in the solution of algebraic equations.

To summarise, the main operations on a validated differential algebra are:

Definition 4.4 (*Validated differential algebra*)

Antiderivative $\int_i : \widehat{F} \to \widehat{F}$.

If $\widehat{F}$is a set of uniform function models of the form $\hat{f} \models f \iff ||f - \tilde{f}||_D \leq \bar{e}$, then also

Weak derivative $\partial_i : \widehat{F} \to \widehat{F}$ such that $\partial_i \hat{f} \models \partial \tilde{f}/\partial x_i$.

5 Polynomial Models

The Taylor models of [13] provide an approximation of continuous functions by polynomials along with bounds or enclosures of the approximation and roundoff errors. We use the terminology 'polynomial model' rather than 'Taylor model' since the polynomial that we use in the approximation of a function f need not approximate the Taylor expansion of f. The advantage is that by giving a more relaxed notion of a model we get a more general function calculus. This will increase the robustness of our library with respect to possible changes and allow for possible use of alternative bases. A possible example would be the recent validated version of [7] which are useful for efficient evaluation of transcendental functions [17] or ODE solving [12]. Results on the correctness of Taylor model calculus were given in [15].

Formally, we can represent a polynomial p in n variables with coefficients in $\mathbb{X}$ by a list of pairs $\langle \alpha_i, a_i \rangle$ where $\alpha_i \in \mathbb{N}^n$ and $a_i \in \mathbb{X}$, i.e. as an element of $(\mathbb{N}^n \times \mathbb{X})^*$. The evaluation operator on $(\mathbb{N}^n \times \mathbb{R})^* \times \mathbb{R}^n$ is given by taking

$$\text{eval}(p, x) = \sum_{i=1}^{k} a_i \, x^{\alpha_i}. \tag{5.1}$$

for $p \in (\mathbb{N}^n \times \mathbb{R})^k$. By a slight abuse of notation, we henceforth write $\sum_{i=1}^{k} a_i \, x^{\alpha_i}$ for the polynomial defined by the list $[\langle \alpha_i, a_i \rangle]_{i=1}^{k}$. Here we use the convention that $\alpha_i := (\alpha_{i,1}, \ldots, \alpha_{i,n})$ and write x^{α_i} as a shorthand for $x_1^{\alpha_{i,1}} \cdots x_n^{\alpha_{i,n}}$. Note that different lists may define the same polynomial function. We denote the set of all polynomials over $\mathbb{R}^n$ with coefficients in $\mathbb{X}$ by $P_n[\mathbb{X}]$.

We define a *polynomial model* as a pair $\langle p, e \rangle$, where p is a multivariate polynomial with coefficients in $\mathbb{F}$, and $e \in \mathbb{F}^+$ is an error bound. We will sometimes write $p(x) \pm e$ for the polynomial model $\langle p, e \rangle$. We denote the set of all polynomial models by $\widehat{P}$.

Let E be the unit domain $[-1, +1]^n$, and $C(E; \mathbb{R})$ denote the set of continuous real-valued functions on E. We say that a polynomial model $\hat{f} = \langle p, e \rangle$ *models* a function $f : E \to \mathbb{R}$, denoted $\langle p, e \rangle \models f$, if

$$\forall x \in E, \ |\iota(p)(x) - f(x)| \leq \iota(e), \tag{5.2}$$

where $\iota(p)(x)$ is evaluated *exactly*; $\iota(p)(x) = \sum_{i=1}^{k} \iota(a_i) x^{\alpha_i}$. It can be seen that for a given $g \in C(E; \mathbb{R})$, by taking $e \in \mathbb{F}$ such that $\|g\|_\infty \leq \iota(e)$ we have

$$\{f \in C(E; \mathbb{R}) | \ \|f\|_\infty \leq e\} \ni g.$$

Hence this predicate gives rise to a complete modelling relation between $\widehat{P}$ and $C(E;\mathbb{R})$ in the sense of Definition 2.1.

It is clear that given a function $f \in C(E;\mathbb{R})$ there are many polynomial models $\langle p, e\rangle$ which model f. Some of these are more convenient or more efficient to work with than others. Given a total order on $\mathbb{N}^n$, we say a polynomial $p(x) = \sum a_i x^{\alpha_i}$ is sorted if $\alpha_i \leq \alpha_j$ whenever $i \leq j$. We say p has *unique coefficients* if $\alpha_i = \alpha_j$ whenever $i = j$. Clearly, we can always convert a list describing a polynomial to a sorted version. If arithmetic on coefficients is exact, then we can additionally convert a polynomial to a form with unique coefficients. Additionally, we can always obtain a simpler representation by removing some terms from p and accounting for them in the error e. We call such an operation *sweeping*.

We therefore consider the following simplifying operations on polynomials and polynomial models:

Sort $\mathrm{sort}(\langle p, e\rangle)$ is sorted.

Unique $\mathrm{uniq}(\langle p, e\rangle)$ has unique coefficients.

Sweep $\mathrm{sweep}(\langle p, e\rangle)$ has no more coefficients than $\langle p, e\rangle$.

Each of the operations should preserve the modelling relation; equivalently, it should be a validated version of the identity.

In the remainder of this section we provide validated versions of the following functions operating on $C(E;\mathbb{R})$, i.e. we define corresponding operations on $\widehat{P}$. To be complete, one has to prove that they satisfy the Definition 2.2. As such proofs follow a similar pattern we only show the proofs in few selected cases.

Constructors unit 1, coordinate x_i, ball 0 ± 1.

Evaluation $f, x \mapsto f(x)$

Norm $||f|| = \sup_{x\in E} |f(x)|$.

Codomain $[f(E)] \supset \{f(x) \mid x \in E\}$.

Arithmetic scalar multiplication $c \cdot f$, sum $f_1 + f_2$, product $f_1 \times f_2$.

Composition $f, g \mapsto f \circ g$.

Antidifferentiation $\int f dx_i = \int_{a_i}^{x_i} f(x_1, \ldots, x_{i-1}, w, x_{i+1}, \ldots, x_n) dw$.

Weak differentiation $\hat{f} = \{f \mid \sup_{x\in D} |f(x) - p(x)| \leq e\} \implies \partial_i \hat{f} = \partial_i p$.

Refinement $\hat{f}_1 \prec \hat{f}_2 \implies (\hat{f}_1 \models f \implies \hat{f}_2 \models f)$.

In particular, the polynomial models form a validated unital Banach algebra, with unit the constant model 1, and ball 0 ± 1, a validated evaluation algebra, and a validated differential algebra.

For the arithmetic operations and composition, we shall only work with sorted polynomial with unique coefficients, and this property will be preserved.

5.1 Simplification Operations

We begin with a generally useful lemma.

Lemma 5.1 *Suppose $\langle p, e\rangle$ and $\langle p', e'\rangle$ are polynomial models, and there exists a constant $\delta \in \mathbb{R}$ such that $|\iota(p)(x) - \iota(p')(x)| \leq \delta$ whenever $x \in E$, and $e' \geq e + \delta$. Then $\langle p, e\rangle \prec \langle p', e'\rangle$, equivalently, $\langle p, e\rangle \models f \implies \langle p', e'\rangle \models f$.*

Proof If $\langle p, e\rangle \models f$, then $|f(x) - p(x)| \leq e$ for all $x \in E$. Then $|f(x) - p'(x)| \leq |f(x) - p(x)| + |p(x) - p'(x)| \leq e + \delta = e'$ for all $x \in E$. □

5.1.1 Sorting

It is clear that sorting the coefficients of a polynomial does not change the evaluation or the modelling relationship.

Lemma 5.2 $\mathrm{sort}(\langle p, e\rangle) \models f(x) \iff \langle p, e\rangle \models f(x)$.

5.1.2 Unique

Let $p = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$ and suppose p has sorted coefficients. Then we define the unique coefficients operator by

$$\text{uniq}(\langle p, e\rangle) = \begin{cases} \langle p, e\rangle & \text{if} \quad k = 0, 1; \\ a_1 x_1^{\alpha} + \text{uniq}(\langle \sum_{j=2}^{k} a_j x_j^{\alpha}, e\rangle) & \text{if} \quad k = 1 \text{ or } \alpha_1 \neq \alpha_2; \\ \text{uniq}(\langle (a_1 +_n a_2) x^{\alpha_2} + \sum_{j=3}^{k} a_j x^{\alpha_j}, \ e +_u ((a_1 +_u a_2) -_u (a_1 +_l a_2)) \div_u 2\rangle) & \text{if} \quad \alpha_1 = \alpha_2. \end{cases}$$

Lemma 5.3 $\text{uniq}(\langle p, e\rangle) \models f(x)$.

Proof It suffices to consider the final case. Let $p' = (a_1 +_n a_2)x^{\alpha_2} + \sum_{j=3}^{k} a_j x^{\alpha_j}$ and $e' = e + ((a_1 +_u a_2) -_u (a_1 +_l a_2)) \div_u 2$. Set $\alpha = \alpha_1$. Then $|p'(x) - p(x)| = |(a_1 +_n a_2)x^{\alpha} - (a_1 x^{\alpha} + a_2 x^{\alpha})| = |((a_1 +_n a_2) - (a_1 + a_2))x^{\alpha}| \leq |(a_1 +_n a_2) - (a_1 + a_2))||x^{\alpha}| \leq |(a_1 +_n a_2) - (a_1 + a_2))|$. By Lemma 3.1, we have $|(a_1 +_n a_2) - (a_1 + a_2))| \leq ((a_1 +_u a_2) -_u (a_1 +_l a_2)) \div_u 2$, and the result follows from Lemma 5.1. □

5.1.3 Sweep

Let $p = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$. Then we define the sweep operator by $\text{sweep}(\langle p, e\rangle, j) = \langle p', e'\rangle$ where

$$p'(x) = \sum_{\substack{i=1 \\ i!=j}}^{k} a_i x^{\alpha_i}; \quad e' = e +_u |a_j|_e.$$

Lemma 5.4 $\text{sweep}(\langle p, e\rangle) \models f(x)$.

Proof Then $|p(x) - p'(x)| \leq |a_j x^{\alpha_j}| \leq |a_j||x^{\alpha_j}| = |a_j|$. The result follows from Lemma 5.1. □

5.2 Constructors

Constructors provide the basis of our function calculus by providing polynomial models out of other existing objects. There are many possible constructors definable. Here we show the implementation of the most useful ones. The proof that they satisfy Definition 2.2 is trivial.

Unit We have that $\langle [\langle (0, \ldots, 0), 1_e\rangle], 0_e\rangle \models f(x) := 1$.
Coordinate For each $j \leq n$ we have that $\langle [\langle (0, \ldots, 1, \ldots, 0), 1_e\rangle], 0_e\rangle \models f(x_1, \ldots, x_n) := x_j$.
Ball We have that $\langle [], 1_e\rangle \models f(x) \iff \sup\{|f(x)| \mid x \in E\} \leq 1$.

We can define an interval constant function for $\hat{c} = [l, u] \in \mathbb{I}$ as $\hat{c} \mapsto \langle m, r\rangle$ where $m = (l +_n u) \div_n 2$ and $r = \max(m -_u l, u -_u m)$. Using the unit and ball constructions, this becomes $m \cdot 1 + r \cdot (0 \pm 1)$.

5.3 Evaluation

The definition of the modelling relationship for $\langle p, e\rangle$ made use of the exact evaluation for the injection $\iota(p)$ of $p \in P_n[\mathbb{F}]$ into $P_n[\mathbb{R}]$. The formula (5.1) defines a simple evaluation scheme, but there are many other evaluation schemes, notably Horner's rule, which give the same answer when computed using exact arithmetic, but may be more efficient or give more accurate results when computed using interval arithmetic. Rather than focus on the correctness of a single evaluation algorithm, we give a result which yields correctness for any evaluation algorithm.

Theorem 5.5 *Suppose* `eval` *is an algorithm for evaluating polynomials with real coefficients which uses only the arithmetical operators* $+, -, \times$ *and rational constants. Then applying* `eval` *to the evaluation of* $P_n[\mathbb{F}]$ *on* $\mathbb{I}^n$ *gives a validated evaluation operator* $\widehat{\text{eval}} : P_n[\mathbb{F}] \times \mathbb{I}^n \to \mathbb{I}$ *which is correct in the sense that* $\widehat{\text{eval}}(p, \hat{x}) \models \iota(p)(x)$ *whenever* $\hat{x} \models x$.

Further, if $\hat{f} = \langle p, e\rangle$ *is a polynomial model, then defining* $\widehat{\text{eval}}(\hat{f}, \hat{x}) = \widehat{\text{eval}}(p, \hat{x}) + [-e, +e]$ *is a correct evaluation scheme for polynomial models and interval vectors.*

Proof For a floating-point polynomial p, applying the evaluation algorithm yields $\text{eval}(p, x) := \texttt{eval}(\iota(p), x) = \iota(p)(x)$. Replacing x by $\hat{x}$ gives $\widehat{eval}(p, \hat{x}) := \texttt{eval}(p, \hat{x}) \models \text{eval}(\iota(p), x)$ whenever $\hat{x} \models x$. If $\langle p, e\rangle \models f$, then for any $x \in \mathbb{R}^n$, $|f(x) - \iota(p)(x)| \leq e$, so $f(x) \in \iota(p)(x) + [-e, +e] \subset \widehat{\text{eval}}(p, \hat{x}) + [-e, +e]$ whenever $\hat{x} \models x$. □

In fact, it suffices in Theorem 5.5 to consider evaluation for polynomials with rational coefficients on $\mathbb{Q}^n$ since only arithmetical operations are used. Note that we allow an algorithm to use information about *structural* zeroes, but not branching based on a comparison $x \geq 0$.

Corollary 5.6 *The direct evaluation scheme*

$$\widehat{\text{eval}}(\langle p, e\rangle, \hat{x}) = [-e, +e] + \sum_{i=1}^{k} [a_i, a_i] \prod_{j=1}^{n} \hat{x}_j^{\alpha_{i,j}}$$

is a correct validated evaluation operator for polynomial models acting on interval vectors.

In practice, an evaluation algorithm based on Horner's scheme is often used. By Theorem 5.5, in order to prove correctness of a particular implementation, we need only prove correctness of the scheme using standard exact arithmetic. A simplified version of the algorithm used in Ariadne is given in Appendix A.2

5.4 Norm and Codomain

Define a validated norm $\widehat{\text{nrm}} : \widehat{P} \to \mathbb{F}$ by

$$\widehat{\text{nrm}}\left(\sum_{i=1}^{k} a_i x^{\alpha_i} \pm e\right) = \sum_{i=1}^{k}{}^{u} |a_i| +_u e.$$

Note that this is the extension of function nrm defined in Sect. 5.5.4 from polynomials to polynomial models.

Lemma 5.7 $\widehat{\text{nrm}}\langle p, e\rangle \geq ||f||$ *whenever* $\langle p, e\rangle \models f$

Proof For any $f \Rightarrow \langle p, e\rangle$, we have $f(x) = p(x) + d(x)$ with $|d(x)| \leq e$ for all x. Then

$$\begin{aligned} ||f(x)|| &= \sup\left\{\left|\sum \iota(a_i)x^{\alpha_i} + d(x)\right| \mid x \in X\right\} \leq \sum |\iota(a_i)||x|^{\alpha_i} + ||d|| \\ &\leq \sum |\iota(a_i)| + e \leq \sum{}^{u} |a_i| +_u e = \widehat{\text{nrm}}\langle p, e\rangle. \end{aligned}$$

□

Note that this upper bound for the norm may not be a good over-approximation to the supremum norm. For example, taking $p = -1 + 2x^2$ and $e = 0$, we have $\widehat{\text{nrm}}\langle p, e\rangle = 3$ but $||f|| = \sup\{|p(x)| \mid x \in E\} = 1$.

As an immediate corollary, we obtain an interval over-approximation to the range of any $f \Rightarrow \langle p, e\rangle$. Define a validated codomain function as

$$\widehat{\text{codom}}\langle p, e\rangle = [-\hat{n}, +\hat{n}] \quad \text{where} \quad \hat{n} = \widehat{\text{nrm}}(\langle p, e\rangle).$$

Corollary 5.8 $\widehat{\text{codom}}\langle p, e\rangle \supset f(E)$ *whenever* $\langle p, e\rangle \models f$

5.5 Arithmetic

The functions in this section provide a basic (polynomial) arithmetic module for polynomial models. The two main functions are sum and product. While most of the other functions below can be considered as special case of sum and product, evidently it is more efficient to have specialised implementation for them. In fact our implementation of multiplication uses the implementation of scalar and monomial operations.

5.5.1 Multiplication by Unit Monomial

Let $p = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$. Suppose $\langle p, e\rangle \models f$ and $\alpha \in \mathbb{N}^n$. Then we define the unit monomial multiplication $x^\alpha \langle p, e\rangle := \langle x^\alpha \cdot p, e\rangle$ where

$$x^\alpha \cdot p := \sum_{i=1}^{i=k} a_i x^{\alpha+\alpha_i}.$$

Lemma 5.9 $x^\alpha \langle p, e\rangle \models x^\alpha f(x)$.

Proof Given $x \in E$ set $p_\alpha := x^\alpha \cdot p$. Then we have $|p_\alpha(x) - x^\alpha f(x)| \le |x^\alpha| \cdot |p(x) - f(x)| \le e.$, i.e. $\langle p_\alpha, e\rangle \models x^\alpha f(x)$. □

5.5.2 Scalar Multiplication

Let $p = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$. Suppose $\langle p, e\rangle \models f$ and $c \in \mathbb{F}$. Then we define the scalar multiplication $c\langle p, e\rangle := \langle c \cdot p, e'\rangle$ where

$$c \cdot p := \sum_{i=1}^{k} (c \times_n a_i) x^{\alpha_i},$$

$$e' := |c|_e \times_u e +_u \sum_{i=1}^{k}{}_u (c \times_u a_i -_u c \times_d a_i) \div_u 2.$$

Here the notation $\sum_{i=1}^{k}{}_u x_i$ is used to denote $x_1 +_u \cdots +_u x_k$.

Lemma 5.10 $c\langle p, e\rangle \models \imath(c) f(x)$.

Proof Given $x \in E$ set $p_c := c \cdot p$. First note that

$$\begin{aligned}
|\imath(c)p(x) - p_c(x)| &= \left|\sum_{i=1}^{k} (\imath(c)\imath(a_i) - \imath(c \times_n a_i)) x^{\alpha_i}\right| \le \sum_{i=1}^{k} |\imath(c)\imath(a_i) - \imath(c \times_n a_i)|\, |x^{\alpha_i}| \\
&\le \sum_{i=1}^{k} |\imath(c)\imath(a_i) - \imath(c \times_n a_i)| \le \sum_{i=1}^{k} \imath((c \times_u a_i -_u c \times_d a_i) \div_u 2) \\
&\le \imath\left(\sum_{i=1}^{k}{}_u (c \times_u a_i -_u c \times_d a_i) \div_u 2\right),
\end{aligned}$$

where the last two step are obtained by applying, respectively, Lemma 3.1 and axiom `flt_add_u`. Then we have

$$\begin{aligned}
|p_c(x) - \imath(c) f(x)| &\le |\imath(c)| \cdot |f(x) - p(x)| + |\imath(c)p(x) - p_c(x)| \\
&\le |\imath(c)|\imath(e) + \imath\left(\sum_{i=1}^{k}{}_u (c \times_u a_i -_u c \times_d a_i) \div_u 2\right)
\end{aligned}$$

$$\leq \iota(|c|_e)\iota(e) + \iota\left(\sum_{i=1}^{k}{}_u(c \times_u a_i -_u c \times_d a_i) \div_u 2\right)$$
$$\leq \iota\left(|c|_e \times_u e +_u \sum_{i=1}^{k}{}_u(c \times_u a_i -_u c \times_d a_i) \div_u 2\right),$$

where in the last two steps the axioms `flt_abs_e`, `flt_mul_u` and `flt_add_u` are applied. □

5.5.3 Sum

The simplest way to define the sum operator on polynomials is to catenate the lists of index–coefficient pairs, sort and make them unique.

$\langle p_1, e_1\rangle + \langle p_2, e_2\rangle = \text{uniq}\,\text{sort}(\langle \text{cat}(p_1, p_2), e_1 +_u e_2\rangle)$

If $p_1 = \sum_{i=1}^{k_1} a_i x^{\alpha_i}$ and $p_2 = \sum_{j=1}^{l} b_j x^{\beta_j}$ are sorted with unique coefficients, we can define $p_1 +_n p_2$ recursively by

$$\sum_{i=1}^{k} a_i x^{\alpha_i} +_n \sum_{j=1}^{l} b_j x^{\beta_j} = \begin{cases} \sum_{i=1}^{k} a_i x^{\alpha_i} & \text{if } \ l = 0; \\ \sum_{j=1}^{l} b_j x^{\beta_j} & \text{if } \ k = 0; \\ a_1 x^{\alpha_1} + \left(\sum_{i=2}^{k} a_i x^{\alpha_i} +_n \sum_{j=1}^{l} b_j x^{\beta_j}\right) & \text{if } \ \alpha_1 < \beta_1; \\ b_1 x^{\beta_1} + \left(\sum_{i=1}^{k} a_i x^{\alpha_i} +_n \sum_{j=2}^{l} b_j x^{\beta_j}\right) & \text{if } \ \alpha_1 > \beta_1; \\ (a_1 +_n b_1) x^{\alpha_1} + \left(\sum_{i=2}^{k} a_i x^{\alpha_i} +_n \sum_{j=2}^{l} b_j x^{\beta_j}\right) & \text{if } \ \alpha_1 = \beta_1. \end{cases}$$

Then we define the sum on $\widehat{P}$ as $\langle p_1, e_1\rangle + \langle p_2, e_2\rangle := \langle p', e'\rangle$ where $p' = p_1 +_n p_2$ and

$$e' := e_1 +_u e_2 +_u \sum_{\substack{i,j=1 \\ \alpha_i = \beta_j}}^{i,j=k,l} (a_i +_u b_j -_u a_i +_d b_j) \div_u 2.$$

Lemma 5.11 $\langle p_1, e_1\rangle + \langle p_2, e_2\rangle \models f_1 + f_2$.

Proof Given $x \in E$, set $p_+ := p_1 +_n p_2$. First note that

$$\begin{aligned} |p_1(x) + p_2(x) - p_+(x)| &= \left| \sum_{\substack{\alpha_i = \beta_i \\ a_i, b_i \neq 0}} (\iota(a_i) + \iota(b_i) - \iota(a_i +_n b_i)) x^{\alpha_i} \right| \\ &\leq \sum_{\substack{\alpha_i = \beta_i \\ a_i, b_i \neq 0}} |\iota(a_i) + \iota(b_i) - \iota(a_i +_n b_i)| \, |x_i^{\alpha}| \\ &\leq \sum_{\substack{\alpha_i = \beta_i \\ a_i, b_i \neq 0}} |\iota(a_i) + \iota(b_i) - \iota(a_i +_n b_i)| \\ &\leq \sum_{\substack{\alpha_i = \beta_i \\ a_i, b_i \neq 0}} \iota((a_i +_u b_i -_u a_i +_d b_i) \div_u 2) \\ &= \iota\left(\sum_{\substack{\alpha_i = \beta_i \\ a_i, b_i \neq 0}}{}_u (a_i +_u b_i -_u a_i +_d b_i) \div_u 2\right), \end{aligned}$$

where the last two steps are obtained by applying, respectively, Lemma 3.1 and axiom `flt_add_u`.

Hence by using the above inequality and applying axiom `flt_add_u` we have

$$\begin{aligned}|p_+(x) - (f_1(x) + f_2(x))| &\le |f_1(x) - p_1(x)| + |f_2(x) - p_2(x)| + |(p_1(x) + p_2(x)) - p_+(x)| \\ &\le \iota(e_1) + \iota(e_2) + \iota\left(\sum_{\substack{\alpha_i=\beta_i \\ a_i,b_1\neq 0}}^{u} (a_i +_u b_i -_u a_i +_d b_i) \div_u 2\right) \\ &\le e_1 +_u e_2 +_u \sum_{\substack{\alpha_i=\beta_i \\ a_i,b_1\neq 0}}^{u} (a_i +_u b_i -_u a_i +_d b_i) \div_u 2.\end{aligned}$$

□

5.5.4 Product

So far we have defined the operations by directly constructing a polynomial model from scratch, by showing what the sparse polynomial component and the error component should be. In principle it is possible to define the product of two polynomial models too in this way, by using the convolution product. However in the n-dimensional case, assessing the error using this approach is not efficient. So we define the product in terms of simpler arithmetic operations.

First, given a sparse polynomial $p = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$ we define

$$\mathrm{nrm}(p) := \sum_{i=1}^{k}{}^{u} |a_i|_e.$$

Note that for $x \in E$ we have $|p(x)| \le \mathrm{nrm}(p)$ because of axiom `flt_add_u`.

Next let $p_1 = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$ and $p_2 = \sum_{i=1}^{i=l} b_i x^{\beta_i}$ and assume $\langle p_1, e_1\rangle \models f_1$ and $\langle p_2, e_2\rangle \models f_2$.

Lemma 5.12 *Let $d_i(x) := f_i(x) - p_i(x)$. Then*

$$\langle \bar{0}, e_1 \times_u \mathrm{nrm}(p_2) +_u e_2 \times_u \mathrm{nrm}(p_1) +_u e_1 \times_u e_2\rangle \models d_1 p_2 + d_2 p_1 + d_1 d_2,$$

where $\bar{0}$ is the constant zero polynomial.

Proof Given $x \in E$ we have

$$\begin{aligned}|0 - d_1(x)p_2(x) + d_2(x)p_1(x) + d_1(x)d_2(x)| &\le \iota(e_1)|p_2(x)| + \iota(e_2)|p_1(x)| + |\iota(e_1)\iota(e_2)| \\ &\le \iota(e_1)\iota(\mathrm{nrm}(p_2)) + \iota(e_2)\iota(\mathrm{nrm}(p_1)) + \iota(e_1)\iota(e_2) \\ &\le \iota(e_1 \times_u \mathrm{nrm}(p_2) +_u e_2 \times_u \mathrm{nrm}(p_1) +_u e_1 \times_u e_2);\end{aligned}$$

where in the last step axioms `flt_add_u` and `flt_mul_u` are used. □

The next operation that we need is a polynomial multiplication that outputs a polynomial *model*. We define

$$\begin{cases} \bar{0} \times_P p_2 := \bar{0}; \\ p_1 \times_P p_2 := a_1 \cdot x^{\alpha_1} \cdot \langle p_2, 0_e\rangle + (\sum_{i=2}^{k} a_i x^{\alpha_i}) \times_P p_2. \end{cases}$$

Note that this operation, which is defined recursively here, can be understood as the product of two polynomial models whose error is 0_e.

Lemma 5.13 $p_1 \times_P p_2 \models p_1 p_2$.

Proof We proceed by induction on the length (i.e. number of distinct coefficients) of p_1. The base case $p_1 = \bar{0}$ vacuously holds. For the induction step assume $p_1 = \iota(a_1)x^{\alpha_1} + p_1'$ such that $p_1' \times_P p_2 \models p_1' p_2$. Then

$$p_1 \times_P p_2 = a_1 \cdot x^{\alpha_1} \cdot \langle p_2, 0_e\rangle + p_1' \times_P p_2$$

Note that $\langle p_2, 0_e\rangle \models p_2$ and hence by Lemmas 5.10 and 5.9 we have $a_1 x^{\alpha_1}\langle p_2, 0_e\rangle \models \iota(a_1)x^{\alpha_1} p_2$. The conclusion follows form this and the induction hypothesis, by applying Lemma 5.11. □

Finally we can define the product of two polynomial models:

$$\langle p_1, e_1\rangle \times \langle p_2, e_2\rangle := p_1 \times_P p_2 + \langle \bar{0}, e_1 \times_u \mathrm{nrm}(p_2) +_u e_2 \times_u \mathrm{nrm}(p_1) +_u e_1 \times_u e_2\rangle.$$

Lemma 5.14 $\langle p_1, e_1\rangle \times \langle p_2, e_2\rangle \models f_1 f_2$.

Proof Taking d_i as in Lemma 5.12, note that $f_1 f_2 = p_1 p_2 + d_1 p_2 + d_2 p_1 + d_1 d_2$. So we can rewrite the right hand side accordingly. The conclusion then follows by applying Lemmas 5.11–5.13. □

5.6 Composition

Having defined arithmetical operators on polynomial models, we can define composition in a similar way to evaluation. Indeed, we can define evaluation on any algebra over $\mathbb{R}$, and apply this to the algebra $C(\mathbb{R}^n; \mathbb{R})$ to prove computability.

Theorem 5.15 *Let* `eval` *be an evaluation algorithm for $P_n[\mathbb{R}]$ on $\mathbb{R}^n$ using only rational constants, addition and multiplication. Let $\mathbb{A}$ be any algebra over $\mathbb{R}$, and $\widehat{\mathbb{A}}$ be a validated version of $\mathbb{A}$. Then* `eval` *yields a correct validated evaluation* $\widehat{\mathrm{eval}} : \widehat{P} \times \widehat{\mathbb{A}}^n \to \widehat{\mathbb{A}}$.

Proof The algorithm `eval` can be used to evaluate $P_n[\mathbb{R}]$ on $\mathbb{A}^n$. The result follows since $\widehat{\mathbb{A}}$ is a validated model of $\mathbb{A}$ with the required operations. □

Using Lemmas 5.11, 5.13, and 5.14 we immediately obtain the a validated function composition:

Corollary 5.16 *Let* `eval` *be an evaluation algorithm for $P_n[\mathbb{R}]$ on $\mathbb{R}^n$ using only rational constants, addition and multiplication. Let $\hat{g}_i = \langle q_i, d_i\rangle$ be polynomial models such that $\hat{g} \models g_i$ for $i = 1, \ldots, n$, and $\hat{f} = \langle p, e\rangle \models f$ be a polynomial model. Then the evaluation algorithm* eval *on p applied to $(\langle q_1, d_1\rangle, \ldots, \langle q_n, d_n\rangle$ induces a validated composition operator $\widehat{comp} : \widehat{P} \times \widehat{P}^n \to \widehat{P}$ by $(\hat{f}, (\hat{g}_1, \ldots, \hat{g}_n)) \mapsto$* `eval`$(p, (\hat{g}_1, \ldots, \hat{g}_n)) + [-e, +e]$.

5.7 Antidifferentiation

let μ_j be the jth unit basis multi-index $(\mu_j)_j = 1$, $(\mu_j)_k = 0$ otherwise. Then the formula for the exact antidifferentiation operator on polynomials is

$$\int_j \sum_{i=1}^{k} a_i x^{\alpha_i} = \sum_{i=1}^{k} \frac{a_i}{\alpha_{i,j} + 1} x^{\alpha_i + \mu_j}.$$

Let $p = \sum_{i=1}^{i=k} a_i x^{\alpha_i}$. Then we define the antidifferential $\int_j \langle p, e\rangle := \langle p', e'\rangle$ where

$$p' := \sum_{i=1}^{k} \big(a_i \div_n (\alpha_{i,j} + 1)\big) x^{\alpha_i + \mu_j},$$

$$e' := e +_u \frac{1}{2}_e \times_u \sum_{i=1}^{k}{}^{u} \big(a_i \div_u (\alpha_{i,j} + 1) -_u a_i \div_d (\alpha_{i,j} + 1)\big).$$

Lemma 5.17 *If $\langle p, e\rangle \models f$, then $\int_j \langle p, e\rangle \models \int_j f$.*

Proof Given $x \in E$ note that $|f(x) - p(x)| \le e$, $|x_j| \le 1$ and $|x^\alpha| \le 1$ for all multi-indices α. We then have

$$\begin{aligned}
&\left| \int_0^{x_j} p(x_1, \ldots, w_j, \ldots, x_n)\, dw_j - \sum_{i=1}^{k} \big(a_i \div_n (\alpha_{i,j}+1)\big) x^{\alpha_i+\mu_j} \right| \\
&= \left| \sum_{i=1}^{k} \big(\iota(a_i)/(\alpha_{i,j}+1) - \iota(a_i \div_n (\alpha_{i,j}+1))\big) x^{\alpha_i} \right| \\
&\le \sum_{i=1}^{k} \big|\iota(a_i)/(\alpha_{i,j}+1) - \iota(a_i \div_n (\alpha_{i,j}+1))\big| \\
&\le \frac{1}{2}_e \times_u \sum_{i=1}^{k}{}^{u} \big(\iota(a_i) \div_u (\alpha_{i,j}+1) -_u a_i \div_d (\alpha_{i,j}+1)\big)
\end{aligned}$$

and

$$\begin{aligned}
&\left| \int_0^{x_j} f(x_1, \ldots, w_j, \ldots, x_n)\, dw_j - \int_0^{x_j} p(x_1, \ldots, w_j, \ldots, x_n)\, dw_j \right| \\
&\le \left| \int_0^{x_j} |f(x_1, \ldots, w_j, \ldots, x_n) - p(x_1, \ldots, w_j, \ldots, x_n)|\, dw_j \right| \\
&\le |x_j| \sup_{x \in E} |f(x) - p(x)| \le e.
\end{aligned}$$

Hence $|| \int_j f - p' || \le e'$ as required. □

5.8 Weak Differentiation

Define the weak differential $\hat{\partial}_j \langle p, e \rangle := \langle p', e' \rangle$ where

$$p' := \sum_{\substack{i=1 \\ \alpha_{i,j} \neq 0}}^{k} \big(a_i \times_n \alpha_{i,j}\big) x^{\alpha_i - \mu_j}, \quad e' := \frac{1}{2}_e \times_u \sum_{\substack{i=1 \\ \alpha_{i,j} \neq 0}}^{k}{}^{u} \big(a_i \times_u \alpha_{i,j} -_u a_i \times_d \alpha_{i,j}\big).$$

Lemma 5.18 *If $\langle p, e \rangle \models f$, then $\hat{\partial}_j \langle p, e \rangle \models \hat{\partial}_j f$.*

Proof An exact formula for the derivative of p is

$$\partial_j \sum_{i=1}^{k} a_i x^{\alpha_i} = \sum_{\substack{i=1 \\ \alpha_{i,j} \neq 0}}^{k} a_i \alpha_{i,j}\, x^{\alpha_i - \mu_j}.$$

The result follows since $\hat{\partial}_j \langle p, e \rangle = \hat{\partial}_j \langle p, 0 \rangle \models \partial_j \iota(p)$. □

5.9 Refinement

Recall from Definition 2.3 that $\hat{f}_1$ *refines* $\hat{f}_2$, denoted $\hat{f}_1 \prec \hat{f}_2$, if $\hat{f}_1 \models f \implies \hat{f}_2 \models f$. Define a validated refinement operator by $\widehat{\text{rfn}}(\langle p_1, e_1 \rangle, \langle p_2, e_2 \rangle)$ if $\widehat{\text{nrm}}(p_1 \widehat{-} p_2) +_u e_1 \le e_2$.

Lemma 5.19 *If* $\widehat{\text{rfn}}(\langle p_1, e_1\rangle, \langle p_2, e_2\rangle)$*, then* $\langle p_1, e_1\rangle \prec \langle p_2, e_2\rangle$.

Proof Suppose $f \rightrightarrows \langle p_1, e_1\rangle$. Then $||f - p_2|| \leq ||p_1 - p_2|| + ||f - p_1||$. Since subtraction is correct, we have $\widehat{\text{sub}}(\langle p_1, 0\rangle, \langle p_2, 0\rangle) \models p_1 - p_2$, so $||p_1 - p_2|| \leq \widehat{\text{nrm}}(p_1 \widehat{-} p_2)$. Thus $||f - p_2|| \leq \widehat{\text{nrm}}(p_1 \widehat{-} p_2) +_u e_1 \leq e_2$. Hence $f \rightrightarrows \langle p_2, e_2\rangle$. □

Note that $\widehat{\text{rfn}}(\langle p_1, e_1\rangle, \langle p_2, e_2\rangle)$ is allowed to return `false` even if $\langle p_1, e_1\rangle \prec \langle p_2, e_2\rangle$. It is unreasonable to expect to be able to efficiently implement an exact version of $\prec$, since this relies on computing $||p_1 - p_2||$ exactly. In practice, it suffices to have the sufficient condition for refinement defined above.

5.10 Abstraction

We consider a simple form of abstraction, namely a *sweep* operation, in which terms are discarded independently.

Let $\pi : \mathbb{N}^n \times \mathbb{F} \to \mathbb{B}$. Define sweep$(\langle p, e\rangle, J) = \langle \sum_{j:\neg\pi(a_j,\alpha_j)} \alpha_j x^{a_j}, e +_u \sum^u_{j:\pi(a_j,\alpha_j)} |alpha_j|\rangle$ The sweep operation reduces the number of terms in the polynomial by discarding according to some predicate π depending on the Typical predicates are to remove terms under some threshold, $\pi(a, \alpha) = (\alpha < \delta)$ or above some fixed degree, $\pi(a, \alpha) = (|a| > m)$.

6 Applications of Validated Arithmetic

6.1 Solution of Ordinary Differential Equations

The *Picard* operator for the solution of an ordinary differential equation $\dot{x} = f(x)$ is defined by the iteration

$$\phi_{n+1}(x, t) = x + \int_0^t f(\phi_n(x, s))\, ds.$$

The only operators needed to iterate the Picard operator on domain D for time interval $[0, h]$ are:

- The coordinate functions x_i over $D \times [0, h]$.
- Composition of functions $f \circ \phi_n$.
- The antiderivative operator $\int_0^t \cdots ds$
- The addition operator $+$.

Additionally, we need to give an initial condition ϕ_0 and a termination condition.

There are two possible approaches to computing a validated version $\hat{\phi}$ of the exact flow ϕ using the Picard operator:

1. Pre-compute a *bounding box* B for the solution, and take $\hat{\phi}_0$ to be a function model such that $B \subset \hat{\phi}_0(x, t)$ for all $x \in D$ and $t \in [0, h]$. Then any $\hat{\phi}_m$ models the solution.
2. Hot-start the solution by computing an approximate solution $\tilde{\phi}$ and approximate error bound $\tilde{\epsilon}$, and using $\tilde{\phi} \pm k\tilde{\epsilon}$ as the initial $\hat{\phi}_0$. During the iteration, attempt to validate $\hat{\phi}_{n+1}$ refines $\hat{\phi}_n$ Then $\hat{\phi}_m$ models the solution ϕ for all $m > n$. Usually, if $\hat{\phi}_1$ does not refine $\hat{\phi}_0$, a different constant multiplier for the error will be chosen.

The first approach is guaranteed to succeed, since the bounding box ensures that $\hat{\phi}_0 \models \phi$. In the second approach, it may be possible to accelerate the computation by choosing a good initial solution, but the approach may fail if it the result cannot be validated.

6.2 Solution of Algebraic Equations

Let $f \in C^1(\mathbb{R}^n; \mathbb{R}^n)$. Define the set of differentiably transverse zeros of f as

$Z_{diff}(f) = \{x \in \mathbb{R}^n \mid f(x) = 0$ and $Df(x)$ is non-singular$\}$.

The *interval Newton operator* is one of the main tools to compute $Z_{diff}(f)$. Let $X \in \mathbb{I}^n$ and $x \in X \cap \mathbb{F}^n$, and define

$$N(X) = x - [Df(X)]^{-1} f(x) \tag{6.1}$$

where $[Df(X)] = \text{hull}\{Df(x) \mid x \in X\}$ is the interval hull of the set of Jacobian matrices of f over X. It can be shown that if $N(X) \subset X$, then f has a unique zero in X, and if $N(X) \cap X = \emptyset$, then f has no zeros in X. A proof is given in Appendix B.

Note that the interval Newton operator relies on computing the Jacobian matrix of f, which, as remarked in Sect. 4.3, cannot be carried out for function models based on uniform approximation.

We can instead use a more general version of the Newton operator which is valid for *continuous* functions $f \in C(\mathbb{R}^n; \mathbb{R}^n)$. Take any $p \in C^1(\mathbb{R}^n; \mathbb{R}^n)$ and set $e(x) = f(x) - p(x)$ so $e \in C(\mathbb{R}^n; \mathbb{R}^n)$. Define the perturbed interval Newton operator by

$$N(f, p, X) = x - [Dp(X)]^{-1}(p(x) + e(X)). \tag{6.2}$$

We can show that if f has a zero at $x^* \in X$, then $x^* \in N(f, p, X)$. Further,

Theorem 6.1 *If $N(f, p, X) \subset X$, then f has a zero X.*

We can use this result to compute zeros of function models. If p is differentiable and $\hat{f} = \{f \mid \sup_{x \in D} |f(x) - p(x)| \leq \epsilon\}$, define the weak derivative $D\hat{f}$ to be $\widehat{D}p$. Define

$$N(\hat{f}, X) = x - [D\hat{f}(X)]^{-1}(\hat{f}(x)). \tag{6.3}$$

Corollary 6.2 *If $N(\hat{f}, X) \subset X$, then any $f \Rightarrow \hat{f}$ has a zero in X.*

Hence the weak derivative of $\hat{f}$ can be used to find sets containing a zero of any $f \Rightarrow \hat{f}$.

Example Let $f(x) = x^3$, which has a non-transverse zero at $x = 0$. By taking $p(x) = x$, we have $e(x) = x^3 - x$. Take $X = [-1, 1]$ and $x = 0$. Then we have $p(0) = 0$, $Dp(X) = \{1\}$ and $e(X) = [-2/3\sqrt{3}, +2/3\sqrt{3}]$, yielding $x - \langle Dp(X)\rangle^{-1}(p(x) + e(X)) = [-2/3\sqrt{3}, +2/3\sqrt{3}] \approx [-0.385, +0.385] \subset X$. This is a numerical proof that f has a zero in $[-1, +1]$.

7 From Validation to Formal Verification

We present an account of a validated calculus that is implemented in Ariadne. A path for future work would be to go one step further and apply *formal verification* to obtain a higher level of assurance about the correctness of the computations in Ariadne. Such high level assurance would be very beneficial for safety critical application domains such as aviation or medicine. In order to achieve high level assurance one has to resort to formal method techniques to verify the design patterns as well the actual code of the implementation.

The usual way of verifying hybrid systems is to apply model checking to hybrid automata, e.g. based on methods such as the predicate abstraction [1]. In practice this approach is helpful in verifying several types of properties of systems, however there are circumstances where model checking can be prone to state explosion. Still, a satisfactory level of assurance can be achieved by combining model-checking with theorem proving in the logically rich environment of a theorem prover. This enables one to validate the correctness of model checking algorithms. Further on, one can enhance and simplify model checking algorithms by proving properties about classes of systems such as modular decomposition and symmetry reduction.

Another reason to use a logical framework is the generic structure of the Ariadne. As is evident by our modular treatment of effective, validated and approximate data structures in the previous section, the Ariadne's template library can be viewed as a hierarchical mathematical structure. Hence by applying tools that have an expressive language allows to deal with mathematical subtleties in verification of our tool.

In the long run we plan to take this approach: we use theorem proving tools, model-checkers and logical frameworks to offer the user a high-level of assurance. Such an endeavour is a multilayer task but we can identify two

main tasks: (1) verifying Ariadne's primitives for function calculus (Ariadne's kernel), which are currently based on polynomial models; (2) verifying Ariadne's algorithms for reachability analysis.

These two task certainly aim at different type of problems. Problem (1) roughly consists of verifying the material we have introduced in this article. Such a verification is tantamount to a rigorous analysis of the proofs. Fortunately this need be done only once: when we have checked that all the error analysis in our function calculus is correct, we are sure that the implemented C++ library makes the right decision with respect to round-off errors.

Problem (2) on the other hand is more complex and in some sense open-ended. For each particular property of hybrid systems and for each invocation of any algorithm to solve that we should provide a certificate of the correctness alongside the answer. Any solution to this problem either would be an ever evolving formal verification suite on its own, or more practically, it will require introducing annotations and assertions inside the C++ code of each algorithm. In the latter case a platform would be used to statically generate proof obligations from the assertions and feed them to a theorem prover using as much automation as possible. Experience in a context related to ours, namely verification of floating point C programs[4,3], using the Frama-C platform [10] indicates that this approach is in principle feasible. In the future work we plan to investigate applying this approach for tackling problem (2).

Here we focus on problem (1); as our verification tool we use Coq which is an integrated theorem prover and a logical framework that is also capable of formalising mathematics [16]. This means that in addition to the model checking of hybrid automata, we can use Coq to verify the algorithms for approximating elementary functions and for numerical differentiation and ultimately for solving ODEs. Ultimately this will result in a hybrid systems analyser embedded in Coq. This analyser will enable the user to state and prove properties by model checking or other techniques and it will use the Ariadne tool as an oracle for computations.

In this section one way of (partially) tackling problem (1) in Coq by formalising the algorithms for basic operations on polynomial models from Sect. 5 considered as sparse polynomials with floating-point coefficients. The correctness proofs will be based on the abstract data type of floating point numbers that was introduced in Sect. 3.

We base the presentation on a syntax based on Coq language which is based on the constructive type theory. However the reader unfamiliar with type theory should be able to follow the exposition. One point to bear in mind is that `f x` and `f x y` denote function applications $f(x)$ and $f(x, y)$, respectively. All the results in this section are (formalised) Coq version of the material in the earlier sections. In fact the mathematical proofs above are more legible than Coq proof scripts. So in general we refrain from giving the proofs. Also most of the details of the implementation are given in the appendix, however the main definitions and auxiliary lemmas used in the main proofs are presented. The complete Coq formalisation including all the proofs is available for download at http://www.cwi.nl/~milad/programs/coq/PolynomialModels1D.tar.gz.

7.1 Floating-Point Data Type in Coq

The abstract data type of `Float` can be readily translated to Coq in several ways. We capture the data type, parametrised by a carrier set, as a record type.[2] This means that our type `Float`, will be an abstract data type that contains a carrier set `crr`, denoted $\mathbb{F}$, with some operations and satisfying some axioms.[3]

The type $\mathbb{R}$ from the standard library of Coq is used as the type of idealised real numbers. This means that they are assumed, non-constructively, to provide a model of real numbers, but we cannot use them for actual computations. They are only used in our meta reasoning about the correctness of the validations.

We assume the existence of a function $\texttt{inj} : \mathbb{F} \longrightarrow \mathbb{R}$ which we treat as an injection of the floats into the reals. As in Sect. 3.1, the axioms govern the arithmetic operations in $\mathbb{F}$ and are stated in terms of ι. Again subtraction, $<$ and division by integers are defined operations.

We can *instantiate* this abstract data type by providing suitable candidates for $\mathbb{F}$ and the required operations and proving the axioms, although this is not needed in current project. Nevertheless the concrete type in [11] provides

[2] Alternatively one could use *module types* or *type classes*. But records suffice for our purpose.

[3] Given `F:Float` we usually identify `F` with its carrier set. In Coq this is achieved by using a coercion.

instances for these axioms. Furthermore, certain subsets of 32-bit or 64-bit floating point numbers (e.g. normalised numbers and excluding NaN and ±Inf) can be shown to satisfy our axioms. Other possible instantiations are rational numbers, p-adic numbers, axiomatic real numbers form the standard library of Coq and, constructive exact real numbers.

The Float record is defined by the following operations and axioms:

```
Record Float : Type :=
{ crr :> Set
; inj : crr -> R
; zero_exact : crr
; one_exact : crr
; add_up : crr -> crr -> crr
; add_down : crr -> crr -> crr
; add_near : crr -> crr -> crr
; neg_exact : crr -> crr
; sub_up := fun x y : crr => add_up x (neg_exact y)
; abs_exact : crr -> crr
; mul_up : crr -> crr -> crr
; mul_down : crr -> crr -> crr
; mul_near : crr -> crr -> crr
; ndiv_up : crr -> nat -> crr
; div2_up := fun x : crr => ndiv_up x 2

; flt_zero: inj zero_exact = 0
; flt_one: inj one_exact = 1
; flt_neg: forall x, inj(neg_exact x) = - inj(x)
; flt_abs: forall x, inj(abs_exact x) = Rabs ( inj(x) )
; flt_add_u: forall x y,  (inj x) + inj(y) <= inj(add_up x y)
; flt_add_d: forall x y,  inj(add_down x y) <= inj(x) + inj(y)
; flt_add_n : forall x y z,
        Rabs ( inj(add_near x y) - (inj(x) + inj(y))) <=
                                                  Rabs (inj(z) - (inj(x) + inj(y)))
; flt_mul_u: forall x y,  inj(x) * inj(y) <= inj(mul_up x y)
; flt_mul_d: forall x y,  inj(mul_down x y) <= inj(x) * inj(y)
; flt_mul_n : forall x y z,
        Rabs ( inj(mul_near x y) - (inj(x) * inj(y))) <=
                                                  Rabs (inj(z) - (inj(x) * inj(y)))
; flt_ndiv_u: forall x m, 0<m -> inj(x) / m <= inj (ndiv_up x m)
}.
```

It will also be useful to add functions explicitly giving the error bounds of Lemma 3.1.

```
Definition add_err_bnd x y := div2_up ( sub_up (add_up x y) (add_down x y) ).
Definition mul_err_bnd x y := div2_up ( sub_up (mul_up x y) (mul_down x y) ).
```

Having implemented the axioms, we can proceed to prove properties of this data type. This means that we derive properties for any type $\mathbb{F}$ satisfying the axiomatisation. For example we can prove, progressively, the following lemmas about addition. Note that the last one corresponds with Lemma 3.1.

```
Lemma flt_add_n_u_abs: forall x y, Rabs( inj(add_near x y) - (inj(x)+inj(y)) ) <=
                                              Rabs (inj(add_up x y) - (inj(x)+inj(y))).
Lemma flt_add_n_d_abs: forall x y, Rabs( inj(add_near x y) - (inj(x)+inj(y)) ) <=
                                              Rabs(inj(x)+inj(y) - inj(add_down x y) ).
Lemma flt_add_n_u: forall x y, Rabs( inj(add_near x y) - (inj(x)+inj(y)) ) <=
                                                inj(add_up x y) - (inj(x)+inj(y)).
Lemma flt_add_n_d: forall x y, Rabs( inj(add_near x y) - (inj(x)+inj(y)) ) <=
                                                inj(x)+inj(y) - inj(add_down x y).
Lemma flt_add_n_u_d_R: forall x y, Rabs( inj(add_near x y) - (inj(x)+inj(y)) ) <=
                                          (1/2)*( inj(add_up x y) - inj(add_down x y) ).
Lemma flt_add_n_u_d: forall x y, Rabs( inj(add_near x y)- (inj(x)+inj(y)) ) <=
                                         (1/2)* inj(sub_up (add_up x y) (add_down x y)).
Lemma flt_add_n_u_d_div2: forall x y, Rabs( inj(add_near x y) - (inj(x)+inj(y)) ) <=
                                    inj(div2_up(sub_up (add_up x y) (add_down x y))).
```

Remark 7.1 For technical reasons, the real Coq code is slightly more complicated; we need to add statements of the form

```
Implicit Arguments inj_R [f].
Implicit Arguments add_near [f].
```

to allow `Float` to be used as an abstract data type, and change the statement of the Lemmas slightly to ensure that the arguments represent the same type of `Float`

```
Lemma flt_mul_n_u_d_div2: forall {F:Float} x y,
                                Rabs ( (@inj_R  F (mul_near x y)) - ((inj_R x)*(inj_R y)) )
                                   <= inj_R (div2_up (sub_up (mul_up x y) (mul_down x y))).
```

7.2 Polynomial Models in Coq

As polynomial models are closely related to Taylor models, relevant to our work is the rigorous proof of correctness of Makino–Berz algorithms [13] in [15]. Inspecting through the proofs in Sect. 5 and those in [15] one can see that core arguments are the same. So this section can be seen as formalising the proofs from [15] in Coq. Hence our work is related to several other developments of Taylor models in theorem provers: in Coq [18] using as coefficients the constructive real numbers; or in PVS [8] or HOL Light [9] using rational interval arithmetic. In contrast in our work we use floating-point coefficients. As pointed out in [18] this makes the formalisation more cumbersome but it enables us to be as close to the actual Ariadne kernel as possible. Regarding floating points, we restricted ourselves to an abstract data type covering only the most general properties that we needed for our function calculus. Much work has been done on concrete implementations of floating point arithmetic and their properties in Coq in [11] and more recently by Boldo et al. [2,5].

An alternative approach would be to use the formalisation of floating-point arithmetic being developed in the Flocq project [6]. We have decided not to use this approach since we can achieve the required results on polynomial models with only the relatively small number of operations used here. However, it would also be interesting to prove our results using Flocq, which provides handling of $\pm\infty$ and `NaN`.

The remainder of the section is presented for a type Float that satisfies the axioms in Sect. 7.1. All the Coq functions and theorems that we present are parametrised with respect to this type Float.

First we need a type for univariate sparse polynomials with coefficients in Float, which is simply a list of $(\mathbb{N}, \mathbb{F})$-pairs

```
list (nat * Float)
```

consisting of degrees and coefficients, i.e. a list $[\langle n_0, a_0\rangle, \ldots, \langle n_{k-1}, a_{k-1}\rangle]$ is intended to denote the *symbolic*[4] polynomial $a_0 x^{n_0} + \cdots + a_{k-1}x^{n_{k-1}}$ where $a_i \in \mathbb{F}$. We define an inductive predicate `is_sorted(p)` specifying that p is sorted with respect to the degrees.

```
Inductive is_sorted {A:Type} : list (nat * A) -> Prop :=
  | is_sorted_nil : is_sorted nil
  | is_sorted_one : forall m a, is_sorted (cons (m,a) nil)
  | is_sorted_cons : forall m (a:A) xs a0, head xs = Some a0 -> (m<fst a0) ->
                                                 is_sorted xs -> is_sorted (cons (m,a) xs).
```

A *polynomial model* is a pair composed of such a sparse polynomial and a floating-point error bound.

```
Record PolynomialModel := { polynomial :> list (nat * Float)
                          ; error: Float
                          }.
```

[4] This is symbolic because the evaluation can be done in several ways, on $\mathbb{R}$ or $\mathbb{F}$, and using various rounding modes.

In order to capture the modelling relation in (5.2) we need to define how to *axiomatically* evaluate a polynomial. Note that the evaluation of $p(x)$ in (5.2) does not have to be a computable evaluation; rather it is a semantic notion relating a family of polynomial with a function approximated by them. Since our goal is merely to prove the correctness with respect to the modelling relation, at this we do not need a computable notion of evaluation that we introduced in Sect. 4.2.

So we define the axiomatic evaluation of a polynomial p at a given point $x \in [-1, 1]$ in the straightforward recursive way (by descending degrees), using arithmetic operations in $\mathbb{R}$.

```
Fixpoint ax_eval_polynomial  (p: list(nat * Float)) (x: R) } : R :=
   match p with
   | nil => 0
   | (n,a) :: q => inj(a)*x^(n) + ax_eval_polynomial q x
   end.
```

Here `(n,a) :: q` denotes the case where `p` is a list with the first element `(n,a)` (which is a pair describing a monomial) and the tail `q` which is another (possibly empty) list containing the remaining coefficients.

Note in particular that in axiomatic evaluation the error of a polynomial model does *not* play a role. This is another difference between our polynomial models and traditional Taylor models.[5]

Then the binary predicate `models` between a polynomial model $\langle p, e\rangle$ and a function $f : \mathbb{R} \longrightarrow \mathbb{R}$ specifies that f is approximated by p with error ϵ on interval $[-1, 1]$ (cf. the *containment property* in [15]).

```
Definition models t f := forall x, -1 <= x <= 1 ->
                  Rabs( (ax_eval_polynomial t.(polynomial) x) - f(x) ) <= inj(t.(error)).
```

It is immediate that two extensionally equal functions can be substituted in a modelling relation, a property that we will repeatedly use in our Coq proofs:

```
Lemma models_extensional: forall t f1 f2, models t f1 -> ( forall x, f1 x = f2 x )
                                                                    -> models t f2.
```

Furthermore we can prove that the real image of the error is positive:

```
Lemma polynomial_model_error_nonneg: forall t f, models t f -> 0 <= inj(t.(error)).
```

The definition of some basic polynomial models, namely the zero model and constant model is given in Sect. 7.2.1. An important construct on polynomial models is the tail constructor, which constructs the polynomial model $\sum_{i=2}^{k} a_i x^{n_i} \pm e$.

```
Definition tail_polynomial_model t : PolynomialModel :=
   match t with
   | {| polynomial:= nil |} => zero_polynomial_model
   | {| polynomial:= (n,a) :: q; error:= e |}
           => {| polynomial:= q; error := e |}
   end.
```

Our first correctness theorem shows the semantics of the tail constructor:

```
Theorem tail_polynomial_model_correct: forall t f, models t f ->
forall n a q, t.(polynomial) = (n,a) :: q ->
                          models (tail_polynomial_model t) ( fun x => f(x) - inj(a)*x^n).
```

This format will be repeated in all the coming correctness theorems.

[5] However, this apparent discrepancy is rather superficial. The actual notion of evaluation in Taylor models corresponds with our computable evaluation of section (4.1).

7.2.1 Constructors for Polynomial Models

We define the following constructors on polynomial models.

```
Definition zero_polynomial_model : PolynomialModel :=
   {| polynomial := nil; error:= zero_exact |}.

Definition unit_polynomial_model : PolynomialModel :=
   {| polynomial := (0,one_exact) :: nil; error:= zero_exact |}.

Definition coordinate_polynomial_model : PolynomialModel :=
   {| polynomial := (1,one_exact) :: nil; error := zero_exact |}.

Definition ball_polynomial_model : PolynomialModel :=
   {| polynomial := nil; error := one_exact |}.

Definition constant_polynomial_model (a:Float) : PolynomialModel :=
   {| polynomial := (0,a) :: nil; error := zero_exact |}.
```

To prove correctness of these constructors, we need to show that the results model the correct functions, e.g.

```
Theorem unit_polynomial_model_correct: models  unit_polynomial_model ( fun x => 1).
```

7.2.2 Norm of a Polynomial Model

A validated norm of a polynomial model can be defined in terms of the polynomial part and the uniform error part. An over-approximation to the norm of the polynomial is defined recursively:

```
Fixpoint norm_polynomial_up (p: list (nat * Float)) : Float :=
   match p with
   | nil => zero_exact
   | (n,a) :: q => add_up (abs_exact a) (norm_polynomial_up q)
   end.

Definition norm_polynomial_model t :=
   add_up (norm_polynomial_up t.(polynomial)) t.(error).
```

Correctness of the norm is provided by the following theorem.

```
Theorem norm_polynomial_model_correct: forall t f
   models t f ->
       forall x : R, -1 <= x <= 1 -> Rabs (f x) <= ifr(norm_polynomial_model t).
```

7.3 Arithmetic of Polynomial Models

We show the formalisation of the basic arithmetic operations on polynomial models, namely multiplication by a unit monomial, scalar multiplication by a float, and addition.

7.3.1 Scaling by a Unit Monomial

The simplest case is the multiplication by a unit monomial x^n. This is a function

$\texttt{power\_polynomial\_model} : \mathbb{N} \longrightarrow \texttt{PolynomialModel} \longrightarrow \texttt{PolynomialModel}$

We define this function in terms of the corresponding function on lists.

```
Fixpoint power_polynomial n p :  list (nat * Float) :=
   match p with
   | nil => nil
   | (m,a) :: q => ( n+m , a) :: power_polynomial n q
   end.
```

We can show that this function preserves sorting of the polynomial.

```
Lemma power_polynomial_sorted: forall n p,
                                          is_sorted p -> is_sorted (power_polynomial n p).
```

The definition of `power_polynomial_model` is then

```
Definition power_polynomial_model n t : PolynomialModel :=
   {| polynomial := power_polynomial n t.(polynomial)
    ; error := t.(error) |}.
```

The correctness of the definition is proved in the theorem below.

```
Theorem power_polynomial_model_correct: forall n t f,
                  models t f -> models (power_polynomial_model n t) ( fun x => x^n*f(x)).
```

7.3.2 Scalar Multiplication

Likewise the scalar multiplication is a function

`scale_polynomial_model` : $\mathbb{F} \longrightarrow$ `PolynomialModel` $\longrightarrow$ `PolynomialModel`

that corresponds with the one defined in Sect. 5.5.2. Here we have to define the sparse polynomial and the error component of the scaled polynomial model. The first component is defined using an auxiliary recursive function `scale_polynomial_near` that in the recursion step maps a polynomial given as `(n,a)::q` to `(n,mul_near c a)::scale_polynomial_near c q` where `c` is the scaling factor.

```
Fixpoint scale_polynomial_near c p : list (nat * Float) :=
   match p with
   | nil => nil
   | (n,a) :: q => ( n , mul_near c a) :: scale_polynomial_near c q
   end.
```

We can prove by induction on the structure of the polynomial (which is merely a list), that the operation `scale_polynomial_near` outputs a polynomial that is sorted with respect to the degrees.

```
Lemma scale_polynomial_near_sorted: forall c p,
                                     is_sorted p -> is_sorted (scale_polynomial_near c p).
```

The error component of the scaled polynomial is the function `scale_polynomial_near_error` defined as follows.

```
Definition scale_polynomial_near_error c : list (nat * Float) -> Float :=
  fold_right
   ( fun na => add_up (mul_err_bnd c (snd na)) )
   zero_exact.
```

Here `scale_polynomial_near_error` calculates the sum $\sum^u (c \times_u a_i -_u c \times_d a_i) \div_u 2$ using a right fold.

The error component of the polynomial model is then

```
Definition scale_polynomial_model_error c t : Float :=
   add_up (mul_up (abs_exact c) t.(error))
                                          (scale_polynomial_near_error c t.(polynomial)).
```

These lead to the definition of the scaled model as follows.

```
Definition scale_polynomial_model c t : PolynomialModel :=
   {| polynomial := scale_polynomial_near c t.(polynomial)
    ; error := scale_polynomial_model_error c t |}.
```

The semantics of this operation with respect to the modelling relation is captured by the following correctness theorem (cf. Lemma 5.10).

```
Theorem scale_polynomial_model_correct: forall c t f,
            models t f -> models (scale_polynomial_model c t) ( fun x => inj(c) * f(x)).
```

7.3.3 Addition

Addition of polynomial models is implemented as an operation:

`add_polynomial_models` : `PolynomialModel` ⟶ `PolynomialModel` ⟶ `PolynomialModel`

The polynomial component needs an auxiliary function that adds up two polynomials using `add_near`. This is listed as the Coq function `add_polynomials_near`. Using a curried version of this alongside a lemma stating the sortedness of the sum we can form the polynomial component of the sum: the reason why we need to take a single argument `p1p2` rather than two arguments `p1` and `p2` is that the well-foundedness of the function requires recursion on a single argument.

```
Function add_polynomials_near (p1p2: list (nat * Float) * list (nat * Float))
                    {measure ( fun l1l2 => (length (fst l1l2) + length (snd l1l2))) p1p2}
                                                                    : list (nat * Float) :=
   match p1p2 with
   | (nil, nil) => nil
   | (nil, (n2,a2)::q2) => (n2,a2)::q2
   | ((n1,a1) :: q1, nil) => (n1,a1)::q1
   | ((n1,a1) :: q1 , (n2,a2)::q2) =>
       match lt_eq_lt_dec n1 n2 with
       | inleft (left _) => (n1,a1) :: add_polynomials_near (q1,(n2,a2)::q2)
       | inleft (right _) => (n1,add_near a1 a2) :: add_polynomials_near (q1,q2)
       | inright _ => (n2,a2) :: add_polynomials_near ((n1,a1)::q1,q2)
       end
   end.
```

For calculating the error we use the following function that recursively outputs the list containing the summands $(a_i +_u b_i -_u a_i +_d b_i) \div_u 2$ corresponding to the common degrees (cf. 5.5.3).

```
Function add_polynomials_near_errors (p1p2: list (nat * Float) * list (nat * Float))
                          {measure ( fun l1l2 => (length (fst l1l2) + length (snd l1l2)))}
                                                                    : list (nat * Float) :=
   match p1p2 with
   | (nil, nil) => nil
   | (nil, (n2,a2)::q2) => nil
   | ((n1,a1)::q1, nil) => nil
   | ((n1,a1)::q1, (n2,a2)::q2) =>
       match lt_eq_lt_dec n1 n2 with
       | inleft (left _) => add_polynomials_near_errors (q1,(n2,a2)::q2))
       | inleft (right _) => (n1,add_err_bnd a1 a2)
                                                  :: add_polynomials_near_errors (q1,q2)
       | inright _ => add_polynomials_near_errors ((n1,a1)::q1,q2)
       end
   end.
```

The error of the polynomial addition is computed by summing the errors of the original terms.

```
Definition sum_add_up : list(nat * Float) -> Float :=
                                      fold_right ( fun na => add_up (snd na)) zero_exact.
```

Then we can sum up the elements of the output list using `add_up` to obtain the error component.

```
Definition add_polynomial_models_error t1 t2 : Float :=
   add_up (add_up t1.(error) t2.(error))
       (sum_add_up (add_polynomials_near_errors (t1.(polynomial),t2.(polynomial)))).
```

The sum of the two polynomial models is then defined as

```
Definition add_polynomial_models t1 t2 : PolynomialModel :=
   {| polynomial:= add_polynomials_near (t1.(polynomial),t2.(polynomial))
    ; error:=add_polynomial_models_error t1 t2 |}.
```

We can show that the polynomial addition preserves sorting.

```
Lemma add_polynomials_sorted : forall p1 p2, is_sorted p1 -> is_sorted p2 ->
                                                 is_sorted (add_polynomials_near (p1,p2)).
```

The correctness theorem for addition, i.e. the formalised version of 5.11, is the following.

```
Theorem add_polynomial_models_correct:
forall t1 t2 f1 f2, models t1 f1 -> models t2 f2 ->
                          models (add_polynomial_models t1 t2) ( fun x => f1(x) + f2(x)).
```

7.3.4 Multiplication

The multiplication of polynomial models can be defined in terms of other operations. We first define multiplication of a polynomial model by a polynomial in term of multiplication by monomials ax^n.

```
Definition multiply_monomial_polynomial_model n a t :=
   scale_polynomial_model a (power_polynomial_model n t).

Fixpoint multiply_polynomial_polynomial_model p t :=
   match p with
   | nil => zero_polynomial_model
   | (n,a) :: q =>
       add_polynomial_models
           ( multiply_monomial_polynomial_model a n t )
           ( multiply_polynomial_polynomial_model q t )
   end.
```

We also need multiplication by an error bound, which is a special case of the polynomial model $0 \pm e$.

```
Definition multiply_error_polynomial_model e t :=
   | polynomial:= nil; error:=mul_up e (norm_polynomial_model t) |.
```

The multiplication operation is then:

```
Definition multiply_polynomial_models t1 t2 :=
   add_polynomial_models
       (multiply_polynomial_polynomial_model t1.(polynomial) t2)
       (multiply_error_polynomial_model t1.(error) t2).
```

The correctness result is formalised as follows:

```
Theorem multiply_polynomial_models_correct: forall t1 t2 f1 f2,
       models t1 f1 -> models t2 f2 ->
                     models (multiply_polynomial_models t1 t2) ( fun x => f1(x) * f2(x)).
```

This finishes the Coq formalisation of basic arithmetic operations and their machine-checked correctness proofs. We emphasise that such a restricted (one-dimensional) version of our polynomial model calculus is really just the beginning of a much larger verification endeavour that we sketched in the beginning of this section. Still, our formalisation shows that the axiomatisation for the floating point structure are indeed enough for basic validated arithmetic.

In fact by inspecting the Coq code one can observe that the axiom `flt_one` is *not* necessary for basic arithmetic that we implemented. Furthermore the axiom `flt_add_n` can be replaced by weaker statement `flt_add_n_u_abs` that was presented as lemma in Sect. 7.1. Likewise, the axioms `flt_ndiv_n` and `flt_ndiv_d` are not used in our development. In fact the binary operation `ndiv`, division of float by arbitrary natural number, is needed later on for antidifferentiation (Sect. 5.7): if we restrict ourselves to basic arithmetic we could replace `ndiv` and its three axioms by a constant $\frac{1}{2}_e \in \mathbb{F}$ and an axiom $\iota\left(\frac{1}{2}_e\right) = \frac{1}{2}$.

Acknowledgments Helpful discussions with Mioara Joldeş and Rolland Zumkeller are acknowledged. The second author was supported by a VENI Grant from The Netherlands Organisation for Scientific Research (NWO).

Appendix A: Ariadne C++ Code

A.1 Additions and Scalar Multiplication

```
PolynomialModel add(const PolynomialModel& x, const PolynomialModel& y)
{
   PolynomialModel r(x.arguement_size());
   set_rounding_upward();
   Float te=0.0;
   PolynomialModel::const_iterator xiter=x.begin();
   PolynomialModel::const_iterator yiter=y.begin();
   while(xiter!=x.end() && yiter!=y.end()) {
       if(xiter->index()<yiter->index()) {
           r.append(xiter->index(),xiter->coefficient());
           ++xiter;
       } else if(yiter->index()<xiter->index()) {
           r.append(yiter->index(),yiter->coefficient());
           ++yiter;
       } else {
           assert(xiter->index()==yiter->index());
           const Float& xv=xiter->coefficient();
           const Float& yv=yiter->coefficient();
           set_rounding_upward();
           Float u=xv+yv;
           Float ml=-xv; ml-=yv;
           te+=(u+ml);
           set_rounding_to_nearest();
           Float c=xiter->coefficient()+yiter->coefficient();
           if(c!=0) { r.append(xiter->index(),c); }
           ++xiter; ++yiter;
       }
   }

   while(xiter!=x.end()) {
       r.append(xiter->index(),xiter->coefficient());
       ++xiter;
   }
   while(yiter!=y.end()) {
       r.append(yiter->index(),yiter->coefficient());
       ++yiter;
   }

   set_rounding_upward();
   r.error()=x.error();
```

```
   r.error()+=y.error();
   r.error()+=(te/2);
   set_rounding_to_nearest();
}

void _scal_approx(PolynomialModel& r, const Float& c)
{
   Float& re=r.error();
   set_rounding_upward();
   Float u,ml;
   Float te=0; // Twice the maximum accumulated error
   Float pc=c;
   Float mc=-c;
   for(PolynomialModel::const_iterator riter=r.begin(); riter!=r.end(); ++riter) {
       const Float& rv=riter->coefficient();
       u=rv*pc;
       ml=rv*mc;
       te+=(u+ml);
   }
   re*=abs(c);
   re+=te/2;

   set_rounding_to_nearest();
   Float m=c;
   for(PolynomialModel::iterator riter=r.begin(); riter!=r.end(); ++riter) {
       riter->coefficient()*=m;
   }
   return;
}
```

A.2 Evaluation Using Horner's Rule

The following algorithm [14] evaluates a polynomial using Horner's rule, assuming terms are ordered reverse-lexicographically. In other words, the term in x^α precedes that in x^β if, and only if, for some $j, \alpha_j > \beta_j$ and $\alpha_i < \beta_I$ for all $i < j$.

```
template<class X, class Y>
Y horner_evaluate(const Polynomial<X>& p, const Vector<Y>& x)
{
   typedef typename Polynomial<X>::const_iterator const_iterator;
   Y z=x[0]*0; // The zero element of the ring Y
   array<Y> r(p.argument_size(),z); // An array of working ``registers''

   const_iterator iter=p.begin();
   const uint n=p.argument_size();
   uint k=n;   // The current working register
   uint j=k;   // The lowest register containing a non-zero value
   MultiIndex na=iter->index().begin(); // The values of the next multi-index
   MultiIndex a=na;
   X c=iter->coefficient();
   ++iter;
   while(iter!=p.end()) {
       na=iter->index().begin();
       k=n-1;
       while(a[k]==na[k]) { --k; }

       // Set r[k]=(((c+r[0])*x[0]^a[0]+r[1])
       //             *x[1]^a[1]+...+r[k])*x[k]^(a[k]-na[k])
       Y t=c;
       for(uint i=0; i!=min(j,k); ++i) {
```

```
            for(uint ii=0; ii!=a[i]; ++ii) {
                t=t*x[i];
            }
        }
        for(uint i=min(j,k); i!=k; ++i) {
            t=t+r[i];
            for(uint ii=0; ii!=a[i]; ++ii) {
                t=t*x[i];
            }
            r[i]=z;
        }
        if(j<=k) {
            t=t+r[k];
        }
        for(uint ii=na[k]; ii!=a[k]; ++ii) {
            t=t*x[k];
        }
        r[k]=t;
        j=k;
        c=iter->coefficient();
        a=na;
        ++iter;
    }
    // Set r=(((c+r[0])*x[0]^a[0]+r[1])*x[1]^a[1]+...+r[n-1])*x[n-1]^(a[n-1])
    Y t=c;
    for(uint i=0; i!=j; ++i) {
        for(uint ii=0; ii!=a[i]; ++ii) {
            t=t*x[i];
        }
    }
    for(uint i=j; i!=n; ++i) {
        t=t+r[i];
        for(uint ii=0; ii!=a[i]; ++ii) {
            t=t*x[i];
        }
    }
    return t;
}
```

Appendix B: The Perturbed Interval Newton Operator

Theorem B.1 *Let $f \in C(\mathbb{R}^n; \mathbb{R}^n)$, $p \in C^1(\mathbb{R}^n; \mathbb{R}^n)$ and $X \subset \mathbb{R}^n$ be convex and compact. Let N_p be the operator defined by*

$$N_p(f, X, p, x) = x - \langle Dp(X)\rangle^{-1}(p(x) + e(X)). \tag{B.1}$$

where $e(x) = f(x) - p(x)$. If f has a zero at $x^* \in X$ then $x^* \in N_p(f, X, p, x)$, and if $N_p(f, X, p, x) \subset X$, then f has a zero X.

In the proof we will need the difference operator

$$Dp(x_1, x_2) = \int_0^1 Dp((1-s)x_1 + sx_2)ds \tag{B.2}$$

which is easily seen to satisfy

$$p(x_1) - p(x_2) = Dp(x_1, x_2)\,(x_1 - x_2) \tag{B.3}$$

by the mean value theorem. By definition, if X is convex, then $Dp(x_1, x_2) \in \text{conv}\{Dp(x) \mid x \in X\} = \langle Dp(X)\rangle$ for any $x_1, x_2 \in X$.

Proof Suppose x^* is a zero of f. Then $p(x^*) = f(x^*) - e(x^*) = -e(x^*)$, so $p(x^*) - p(x) = -e(x^*) - p(x)$. Then $Dp(x^*, x)(x^* - x) = -e(x^*) - p(x)$, so $x^* = x - Dp(x^*, x)^{-1}(p(x) + e(x^*))$. Taking convex hulls yields $x^* \in x - \langle Dp(X)\rangle^{-1}(p(x) + e(X)) = N_p(f, X, p, x)$ as required.

Suppose $N_p(f, X, p, x) \subset X$. Fix $x \in X$ and define the Newton-like function $n(z) = z - Dp(x, z)^{-1} f(z)$, so that $n(z) = z \iff f(z) = 0$. For $z \in X$,

$$\begin{aligned} n(z) &= z - Dp(x, z)^{-1}(p(z) + e(z)) = z - Dp(z, x)^{-1}(p(x) + Dp(z, x)(z - x) + e(z)) \\ &= z - (z - x) - Dp(z, x)^{-1}(p(x) + e(z)) = x - Dp(z, x)^{-1}(p(x) + e(z)) \\ &\in x - \langle Dp(X)\rangle^{-1}(p(x) + e(X)) = N_p(f, X, p, x). \end{aligned}$$

Hence $n(X) \subset X$, so n is a function mapping a compact convex subset of $\mathbb{R}^n$ into itself, so has a fixed point x^* which is a zero of f. □

References

1. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embed. Comput. Syst. **5**(1), 152–199 (2006)
2. Boldo, S.: Preuves formelles en arithmétiques à virgule flottante. PhD thesis, École Normale Supérieure de Lyon (2004)
3. Boldo, S.: Formal verification of numerical programs: from C annotated programs to coq proofs. In: Fainekos, G., Goubault, E., Putot, S. (eds.) Proceedings of the NSV-3, Edinburgh, UK (2010) (Federated Logic Conference)
4. Boldo, S., Filliâtre, J.-C.: Formal verification of floating-point programs. In: Muller, J.-M., Kornerup, P. (eds.) Proceedings of the ARITH-18, pp. 187–194. IEEE Computer Society Press, New York (2007)
5. Boldo, S., Filliâtre, J.-C., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) Proceedings of the Calculemus/MKM 2009. LNCS, vol. 5625, pp. 59–74. Springer, Berlin (2009)
6. Boldo, S., Melquiond, G.: Flocq: a unified library for proving floating-point algorithms in Coq (2010). http://hal.inria.fr/inria-00534854
7. Brisebarre, N., Joldeş, M.: Chebyshev interpolation polynomial-based tools for rigorous computing. Technical Report RR2010-13, École Normale Supérieure de Lyon (2010)
8. Cháves Alonso, F.J.: Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves. PhD thesis, École Normale Supérieure de Lyon (2007)
9. Chevillard, S., Harrison, J., Joldeş, M., Lauter, C.: Efficient and accurate computation of upper bounds of approximation errors. Technical Report RR2010-13, École Normale Supérieure de Lyon (2010)
10. Correnson, L., Cuoq, P., Puccetti, A., Signoles, J.: Frama-C user manual. CEA LIST (2010). http://frama-c.com
11. Daumas, M., Rideau, L., Théry, L.: A generic library for floating-point numbers and its application to exact computing. In: Boulton, R.J., Jackson, P.B. (eds.) Proceedings of the TPHOLS 2001. LNCS, vol. 2152, pp. 169–184. Springer, Berlin (2001)
12. Driscoll, T.A., Bornemann, F., Trefethen, L.N.: The chebop system for automatic solution of differential equations. BIT **48**(4), 701–723 (2008)
13. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. Int. J. Pure Appl. Math. **4**(4), 379–456 (2003)
14. Pena, J.M., Sauer, T.: On the multivariate horner scheme. SIAM J. Numer. Anal. **37**(4), 1186–1197 (2000)
15. Revol, N., Makino, K., Berz, M.: Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. J. Log. Algebr. Program. **64**(1), 135–154 (2005)
16. The Coq Development Team: Reference manual, version 8.3. INRIA (2010). http://coq.inria.fr/refman
17. Trefethen, L.N.: Computing numerically with functions instead of numbers. Math. Comput. Sci **1**(1), 9–19 (2007)
18. Zumkeller, R.: Formal global optimisation with Taylor models. In: Furbach, U., Shankar, N. (eds.) Proceedings of the IJCAR 2006. LNCS, vol. 4130, pp. 408–422. Springer, Berlin (2006)