A Coq Formalization of Taylor Models and Power Series for Solving Ordinary Differential Equations

Sewon Park
Graduate School of Informatics, Kyoto University, Japan

Holger Thies
Graduate School of Human and Environmental Studies, Kyoto University, Japan

Abstract

In exact real computation real numbers are manipulated exactly without round-off errors, making it well-suited for high precision verified computation. In recent work we propose an axiomatic formalization of exact real computation in the Coq theorem prover. The formalization admits an extended extraction mechanism that lets us extract computational content from constructive parts of proofs to efficient programs built on top of AERN, a Haskell library for exact real computation.

Many processes in science and engineering are modeled by ordinary differential equations (ODEs), and often safety-critical applications depend on computing their solutions correctly. The primary goal of the current work is to extend our framework to spaces of functions and to support computation of solutions to ODEs and other essential operators.

In numerical mathematics, the most common way to represent continuous functions is to use polynomial approximations. This can be modeled by so-called Taylor models, that encode a function as a polynomial and a rigorous error-bound over some domain. We define types of classical functions that do not hold any computational content and formalize Taylor models to computationally approximate those classical functions. Classical functions are defined in a way to admit classical principles in their constructions and verification. We define various basic operations on Taylor models and verify their correctness based on the classical functions that they approximate. We then shift our interest to analytic functions as a generalization of Taylor models where polynomials are replaced by infinite power series. We use the formalization to develop a theory of non-linear polynomial ODEs. From the proofs we can extract certified exact real computation programs that compute solutions of ODEs on some time interval up to any precision.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Exact real computation, Taylor models, Analytic functions, Computable analysis, Program extraction

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.30

Supplementary Material
Software (Source Code): `https://github.com/holgerthies/coq-aern` [20]
archived at `swh:1:dir:91c89245541a8dbcad3ab085bb8682112e684311`

Funding *Sewon Park*: has been supported by JSPS KAKENHI (Grant-in-Aid for JSPS Fellows) JP22F22071 as a JSPS International Research Fellow.
Holger Thies: Supported by JSPS KAKENHI Grant Numbers JP20K19744, JP23H03346 and JP24K20735.

1 Introduction

A typical problem in numerical analysis with vast applications across various fields such as physics, engineering, biology, and economics, is to approximate solutions to initial value problems (IVPs) of the form

$$\dot{y}(t) = f(y(t)) \;;\; y(t_0) = y_0$$

where $\dot{y}(t)$ denotes the derivative of the function $y(t)$ with respect to the time variable t. Approximating the solution accurately can be challenging as small perturbations in the initial condition can lead to large changes in the solution. Traditional numerical methods therefore often lack the robustness required in safety-critical applications. Thus, approaches to rigorously solve ODEs have been studied extensively. For example, methods from interval arithmetic such as [31] allow to compute the solution reliably in the sense that the resulting intervals are promised to contain the exact solutions. However, accumulation of over-estimations in interval computations make it challenging to obtain solutions in arbitrarily high accuracy. Exact real computation, based on the theory of constructive [4] and computable analysis [36], solves this problem by using infinite representations of real numbers. Instead of approximating a real number by an interval, real numbers are expressed exactly e.g. by an infinite stream of nested shrinking intervals. In this way, real numbers can be represented and manipulated exactly, without round-off errors and other sources of numerical uncertainty inherent in finite precision arithmetic. Implementations of exact real computation such as, for example, [30, 3, 22], provide abstract data types for real numbers hiding representation-specific details from their users. Consequently, users can write and reason about their programs intuitively regarding exact real numbers as the familiar abstract entities from mathematics. See [6, 32] for more about this aspect of exact real computation.

Recently, we have worked on axiomatically formalizing exact real computation in a type-theoretical setting [19] and implementing the idea as the Coq library cAERN [20]. By making use of Coq's code extraction features, we map axiomatically defined types and operations to abstract data types and primitive operations in AERN, a Haskell library for exact real computation [22]. As primitive real number operations are optimized in the implementation of AERN, the extracted certified exact real computation programs are usually more efficient than mapping everything to primitive data-types. For example, in [19] we show that the execution time of extracted programs is comparable to hand-written AERN programs.

The main goal of this work is to extend the cAERN formalization of exact real computation to solution operators for IVPs and other higher-order problems. When dealing with function spaces, representing functions accurately is essential for efficient computation. In numerics, the predominant approach to represent continuous real functions is through polynomial approximations. Taylor models [27] provide a systematic approach to approximating functions by polynomials. A Taylor model represents a function on some interval as a polynomial approximation (typically derived from the function's Taylor series expansion) together with a rigorous error bound on the interval. Many mathematical operations can be directly implemented on Taylor models, accounting for dependencies between variables and thus ensuring more accurate enclosures of function values. Recently the use of Taylor models has also been shown to be beneficial in efficient exact real computation [7].

In this work, we implement a solver for initial value problems for polynomial first-order ODEs and verify its correctness. The solver is based on computing the power series expansion of the solution up to an arbitrarily high degree. To this end we formalize Taylor models and their generalization as infinite power series. We use these to locally represent analytic functions exactly. Thus, our solver computes a functional representation of the solution $y(t)$ on a small time interval $[0, t_0]$, which can be used to approximate the value of the function in the interval up to any desired precision. We can then extend the solution by computing the value $y(t_0)$ and use it as as a new initial value to continue the procedure, similar to single step methods in numerics. Note that since we get an exact representation of the real number $y(t_0)$ we do not have to deal with approximations in this step (see Section 6 for details).

Although the solution method is generic, for simplicity we currently only consider one-dimensional ODEs in the Coq implementation. While this excludes most interesting applications, we think it demonstrates well how the method works, and it should not be difficult to extend to higher dimension.

Since we work on a constructive setting, saying R to be the type of exact real numbers, having a function $f : \mathsf{R} \to \mathsf{R}$ means that we already have a way to compute, in a sense, the function f exactly. Thus, it does not make much sense to develop a theory of approximating them using Taylor models or power series. In other words, the standard function type $\mathsf{R} \to \mathsf{R}$ is stronger than the type of functions that our Taylor models and power series intend to approximate. We therefore formalize a type of classical (partial) functions that is weaker than the standard functions. Classical functions can be constructed based on classical reasoning, e.g. the sign function of reals, but do not yield any computational contents. We define our Taylor models and power series to approximate such classical functions instead and use them as classical specifications which we can reason about based on the classical analysis.

The paper is structured as follows. In the rest of this section, we review the setting of our Coq development and some related works. In Section 2, we formalize classical partial functions and in Section 3 we define the notions of their continuity and differentiability. In Section 4 we formalize a variant of simple univariate Taylor models with real-valued polynomials and a real-valued error bound which we use to approximate classical partial functions. In Section 5 we then further extend this to analytic functions, which we can represent exactly by infinite sequences of polynomials. Finally, in Section 6 we use this to define polynomial initial value problems and solution operators for them.

All results in this paper have been implemented in Coq as an extension of the cAERN library [20]. The new formalization presented consists of approximately 8000 lines of code, while the complete cAERN library consists of approximately 29000 lines of code. The files relevant for the new formalization are ClassicalMonads, ClassicalPartiality, and ClassicalParitalReals for Section 2; ClassicalTopology, ClassicalContinuity, and ClassicalDifferentiability for Section 3; Poly and Taylormodel for Section 4; Powerseries for Section 5; and Ode for Section 6.

1.1 Background

In this paper, we focus mostly on our Coq development, and thus present most results directly in the syntax of Coq. However, it is also straightforward for readers who are not familiar with Coq syntax to translate it using type-theoretic constructions. For example, we use `Prop` for an impredicative type universe (of propositions) without large eliminations and `Type` for a type universe of (small) types. We write $(A \lor B)$: `Prop` for the disjunction in `Prop` whereas we write $A + B$: `Type` for the usual sum type. Similarly, by (`exists` $x : A,\ B\ x$) : `Prop`, we refer to the existence in `Prop` whereas $\{x : A \mid B\ x\}$: `Type` and $\{x : A\ \&\ B\ x\}$: `Type` denote the usual Σ-type. The Π-types are written as `forall` $x : A,\ B\ x$.

As in our previous works [19, 21], we assume axioms that make `Prop` classical, in particular the law of excluded middle of the form (`forall` P : `Prop`, $P \lor \neg P$) : `Prop`. We further assume the dependent function extensionality, classical propositional extensionality saying that for any two types $P\ Q$: `Prop`, $(P \to Q) \land (Q \to P)$ implies $P = Q$, and proof-irrelevance of classical propositions saying that any type P in `Prop` is a *subsingleton* type `forall` $x\ y : P,\ x = y$. We say a type P is a classical proposition when P : `Prop`. We also often write $a : A$ such that $B\ x$ classically exists to mean that `exists` $a : A,\ B\ x$ holds and write P or Q holds classically to mean that $(P \lor Q)$: `Prop` holds. Otherwise, we refer to the constructive variants.

As a direct consequence of making `Prop` classical, equality types become classical and proof-irrelevant. Therefore, we refer a map $f : A \to B$ being a type-theoretic equivalence by the (constructive) existence of its inverse $g : B \to A$ such that for all $x : A$, $g\ (f\ x) = x$ and for all $y : B$, $f\ (g\ y) = y$ hold.

In the previous works, based on this setting we axiomatically formalize types and operations used in exact real number computation. Most importantly, we assume that there is a type R for real numbers containing two distinct constants 0 and 1, the standard arithmetical operators and a semi-decidable comparison operator $<$. We restrict the reciprocal function x^{-1} to require a classical proof that $x \neq 0$ and the limit operator to receive a classical proof of the rapid convergence of a given sequence. The axioms are chosen carefully in a way that allows to reason about properties of real numbers classically, without breaking computational content. Hence, one important feature of the cAERN library (and thus the name) is that we can extract Haskell programs on top of the AERN library which is proven to be correct under the assumption that the basic operations in AERN are implemented correctly. This approach has the advantage that extracted programs are more efficient than extracting everything to basic types like integers, that programs are more readable, and that we can easily integrate certified code with non-verified code in the AERN library. Interesting examples we have formalized and extracted include the maximum function, the absolute value function, a root-finding functional from a constructive intermediate value theorem [17], real and complex square roots [18], computing fractals from proofs related to open, closed, compact, overt, located subsets of Euclidean spaces, and so on [21].

An important note to make here is that, to simplify presentation, the notations we use in this paper are not completely identical to the notations we use in the implementation. For example, we use the notation ^Real in the source code for exact real numbers not R as in this paper. However, the modifications are minimal and thus such correspondence should be clear for readers who also read the source code.

1.2 Related work

Approximating solution trajectories for ordinary differential equations is a key problem in numerical mathematics and methods for rigorous computation have been studied extensively (e.g. [2, 12]) and several rigorous tools have been developed (e.g. [23, 9]). In particular, formal verification of numerical methods for ODEs in proof assistants has been considered previously, e.g. in Isabelle [14, 13] and Coq [24]. The above mentioned works are based on interval arithmetic, thus verify that ODE solutions can be rigorously enclosed in some interval. Our work, on the other hand, is based on computable analysis and therefore, in a sense, uses a stronger notion of correctness, i.e. we need to show that we can make the enclosure at each point arbitrarily small. In this sense, our work is similar to constructive approaches to analysis. A larger formalization of constructive analysis in Coq can be found in the CoRN library [11] which also includes some previous results on constructive formalizations of ODE solutions in Coq [25]. In contrast to CoRN our goal is not to compute inside the proof assistant, but to extract exact real computation programs and to adhere to an abstract axiomatic formalization of real numbers similar to real number types in implementations of exact real computation (see [18] for details). Further, [25] uses the Picard Iteration algorithm, while our proposed method is based on high-order power series expansions. The method is motivated by results from real number complexity theory which suggests that it should be more efficient if the desired output precision is high [29, 8, 16]. We are not aware of any formal verification of this method.

Many verified solvers for ODEs use Taylor models. Taylor models are typically used to deal with the dependency problem of interval arithmetic and to get tighter enclosures of function values than what is achieved with simple intervals [27]. There already exist several formalizations of Taylor models in proof assistants, e.g. [35, 28]. In particular, [28] presents a formally verified implementation of univariate Taylor models in Coq. However, it should be mentioned that both our use case and implementation are quite different from the usual approaches in interval arithmetic. First, as our polynomials and error terms are exact reals, we do not need to deal with issues arising when implementing them using interval arithmetic. Second, while our Taylor models could also be used to compute (real-valued) interval enclosures of functions, our main motivation is to use them to represent certain types of functions exactly as an infinite sequence of polynomials and operate on them efficiently, more similar to the idea of a function space representation in computable analysis [33].

Lastly, although our main aim is to formalize computational results, to verify correctness a substantial amount of classical analysis is needed. While there already is a large number of Coq libraries that deal with classical analysis, e.g. the Coquelicot [5] and MathComp Analysis [1] libraries, making use of them directly is challenging, as we define our own type of (computational) real numbers. Formalizing classical facts about real numbers and functions is therefore also a significant part of our Coq development. On one hand, this allows us to formalize the statements exactly in the way we need them, and make them integrate well with our computational theorems. On the other hand, we do not aim to provide a complete library for classical analysis and do not think that our classical results exceed what has already been proven in the above mentioned libraries. Thus, better integration with these existing formalizations is a goal for future work.

2 Classical Functions

Under our setting, the standard function type of real numbers $\mathsf{R} \to \mathsf{R}$ denotes a structured set of continuously realizable or computable real functions, and thus excludes any discontinuous classical function. However, often discontinuous classical functions play an important role in specifying and developing a meta-theory for computable procedures. For example, though the sign function of real numbers is not computable or continuous, it is used everywhere for specifying a computation on the signs of its real number inputs.

2.1 Classicalizing Monad

Recall the classical singleton subset monad which also appears in [18]:

```
Definition ∇ (A : Type) := {S : A → Prop | exists! x : A, S x}
```

and its monad operations that are defined naturally. In this paper, we write $[x]$ for the monad unit on x and $\widetilde{A}$ as an abbreviation for $\nabla\ A$. Notice that this monad acts as an eraser that erases the computational contents such that $A \to \widetilde{B}$ denotes the set of classical functions from A to B. Though for different x and y we can prove $[x]$ and $[y]$ are distinct, both of the terms will be removed in program extraction. Note also that any classical function from constructive domain $f : A \to \widetilde{B}$ can be extended to be a function from the classical domain $f^{\dagger} : \widetilde{A} \to \widetilde{B}$ by the monadic bind. It is noteworthy that this mapping $f \mapsto f^{\dagger}$ is a type-theoretic equivalence between $A \to \widetilde{B}$ and $\widetilde{A} \to \widetilde{B}$.

As a simple sanity-check of our construction of the so-called *classicalizing monad*, we prove that under our set of axioms, any classical proposition P : `Prop` is classical also in the sense that the monad unit $[\cdot] : P \to \widetilde{P}$ admits an inverse. To construct the inverse, for any

$t : \widetilde{P}$, we destruct t to obtain its unique classical existential witness of P : `Prop`. Note that this construction is allowed because P is of type `Prop`. This construction defining the inverse follows straightforwardly from function extensionality and propositional extensionality.

Coq with our set of base axioms now admits three different ways to make a type A : `Type` classical: (1) by reflecting the type as a classical proposition

```
Definition Propize (A : Type) : Prop := exists _ : A, True
```

(2) by applying the double negation $(\neg\neg A)$: `Prop` (as a notation for $(A \to$ `False`$) \to$ `False`) and (3) by the classicalizing monad $\widetilde{A}$: `Type`. An advanced sanity-check is to verify their equivalences on subsingletons.

▶ **Lemma 1.** *For any subsingleton type A : `Type`, $\widetilde{A}$ is again a subsingleton. Moreover, the three constructions of making A classical are equivalent:*

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[d] \arrow[dr] & \\
\neg\neg A \arrow[r, leftrightarrow, "\simeq"'] & \texttt{Propize}A \arrow[r, leftrightarrow, "\simeq"'] & \widetilde{A}
\end{tikzcd}$$

Proof. The type $\widetilde{A}$ being a subsingleton follows directly from the extensionality axioms. As the three classical types are all subsingleton, we only need to construct mappings between each types where mappings between $\neg\neg A$ and `Propize`A can be constructed trivially as both are of type `Prop`.

Given t : `Propize`A, define an element of type $\widetilde{A}$ with `fun` $x : A \Rightarrow$ `True`. It being classically singleton follows from t : `Propize`A and that A is a subsingleton. For its inverse, we lift the trivial mapping $A \to$ `Propize`A w.r.t. the classicalizing monad to get a mapping of type $\widetilde{A} \to \widetilde{\texttt{Propize}A}$. Since `Propize`$A$ is of type `Prop`, we apply the previous observation that there is an inverse $\widetilde{\texttt{Propize}A} \to \texttt{Propize}A$ to the monad unit $[\cdot] : \texttt{Propize}X \to \widetilde{\texttt{Propize}A}$. Hence, post-composing this yields the desired inverse. ◀

Note that A being a subsingleton is necessary here because in the model when A is not a subsingleton, $\widetilde{A}$ is also not a subsingleton while $\neg\neg A$ and `Propize`A are.

The classicalizing monad being classical not only on subsingletons but for every types can be formalized in Coq as the following lemma:

▶ **Lemma 2.** *For any type A : `Type` and a subsingleton type P : `Type` there is a mapping that maps any $f : P \to \widetilde{A}$ to $\hat{f} : \neg\neg P \to \widetilde{A}$ such that the following diagram commutes:*

$$\begin{tikzcd}
P \arrow[r, "f"] \arrow[d, "\neg\neg\text{-intro}"'] & \widetilde{A} \\
\neg\neg P \arrow[ur, "\hat{f}"'] &
\end{tikzcd}$$

where $\neg\neg\text{-intro} : P \to \neg\neg P :=$ `fun` $x : P \Rightarrow$ `fun` $f : P \to$ `False` $\Rightarrow f\ x$ *is the double negation introduction. Moreover, the mapping is a type-theoretic equivalence*

$$(P \to \widetilde{A}) \simeq (\neg\neg P \to \widetilde{A})$$

where precomposing $\neg\neg$*-intro is the inverse. I.e., for each* $f : P \to \widetilde{A}$ *it holds that* $f = \hat{f} \circ \neg\neg\text{-intro}$ *and for each* $g : \neg\neg P \to \widetilde{A}$ *it holds that* $g = g \circ \widehat{\neg\neg\text{-intro}}$.

Proof. First, we construct the mapping $\hat{f}$ of type $\neg\neg P \to \widetilde{A}$ given $f : P \to \widetilde{A}$ and that P is a subsingleton. We apply the monadic bind on f and get $f^\dagger : \widetilde{P} \to \widetilde{A}$ reducing the goal to constructing a term of type $\widetilde{P}$ from $\neg\neg P$. Let us write $j : \neg\neg P \to \widetilde{P}$ for the mapping from Lemma 1. We conclude the construction with $\hat{f} := f^\dagger \circ j$.

Suppose any $f : P \to \widetilde{A}$ and $x : P$. We prove the equality between $\hat{f}(\neg\neg\text{-intro } x) \equiv f^\dagger(j(\neg\neg\text{-intro } x))$ and $f\ x$. As P is a subsingleton type, due to Lemma 1, $\widetilde{P}$ is also a subsingleton type. Hence, we can prove that $j(\neg\neg\text{-intro } x) = [x]$ in $\widetilde{P}$. That means we have $f^\dagger(j(\neg\neg\text{-intro } x)) = f^\dagger\ [x] = f\ x$ due to the monad axiom of ∇.

Suppose any $g : \neg\neg P \to \widetilde{A}$ and $x : \neg\neg P$. We need to prove $(g \circ \neg\neg\text{-intro})^\dagger(j\ x) = g\ x$. Since we are proving an equality as of type `Prop`, we can locate $y : P$ such that $x = \neg\neg\text{-intro } y$. Then, similarly to the previous case, $j\ x = [y]$. Therefore, $(g \circ \neg\neg\text{-intro})^\dagger(j\ x) = (g \circ \neg\neg\text{-intro})^\dagger([y]) = g(\neg\neg\text{-intro } y) = g\ x$ concludes the proof. ◀

A side note worth mentioning here is that thus the monad ∇ behaves like the double-negation sheafification in the setting of topos theory. As long as $\widetilde{A}$ is concerned, for a subsingleton P, the double negated classical version of it $\neg\neg P$ is equivalent to P itself.

For any subsingleton type P, the double negated excluded middle $\neg\neg(P + \neg P)$ is constructively provable and also is a subsingleton. Therefore, for any such P and a type A : `Type`, one can construct a term of type $\widetilde{A}$ by a case distinction on P or $\neg P$. Let us write $\mathsf{lem}(f) : \widetilde{A}$ for this construction from $f : P + \neg P \to \widetilde{A}$.

▶ **Example 3.** For a real number $x : \mathsf{R}$, define $f\ x : (x < 0) + \neg(x < 0) \to \widetilde{\texttt{bool}}$ by

```
f p := match p with
       | inl _ ⇒ [false]
       | inr _ ⇒ [true]
       end.
```

that returns [`true`] when its input is an evidence that $x \geq 0$ and returns [`false`] when its input is an evidence that $x < 0$. Then, (`fun` $x : \mathsf{R} \ \Rightarrow\ \mathsf{lem}(f\ x)) : \mathsf{R} \to \widetilde{\texttt{bool}}$ denotes the classical sign function.

Lemma 2 is effective enough to construct the reduction:

▶ **Example 4.** Suppose we have constructed $\mathsf{lem}(f : P + \neg P \to \widetilde{A}) : \widetilde{A}$ but know either P holds or not by having a term $t : P + \neg P$. In this case, we can prove

$$\mathsf{lem}(f) = f\ t$$

as a direct consequence of Lemma 2. Another important reduction case is when the value is irrelevant to the case distinction and is already known to be $y : \widetilde{A}$. When we have that $f\ (\mathsf{inl}\ t) = y$ for all $t : P$ and $f\ (\mathsf{inr}\ t) = y$ for all $t : \neg P$, then we can obtain $\mathsf{lem}(f) = y$. When we apply this to the classical sign function in Example 3, given $t : x < 0$ for example, we can obtain that the value of the sign function equals to [`false`].

2.2 Classical Partial Functions

We define a classical partial function as

$$f : A \to \widetilde{B_\perp}$$

where $B_\perp$ denotes $\mathsf{option}\ B$ in Coq, a maybe monad meaning that $B_\perp$ is an inductive type with two constructors $\mathsf{Some} : B \to B_\perp$ and $\mathsf{None} : B_\perp$. We define some obvious operations on partial functions such as $f\ x \downarrow y$ when $y : B$ to denote $f\ x = [\mathsf{Some}\ y]$ saying that $f\ x$ is defined to be y. We write $f\ x\downarrow$ for $\exists(y : Y).\ f\ x \downarrow y$.

An important way of constructing a classical partial function is from its classical graph or specification:

▶ **Lemma 5.** *Given a classical binary relation $G : A \to B \to$ `Prop`, such that for each $x : A$, $\{y : B \mid G\ x\ y\}$ is a subsingleton type, we can define the corresponding classical partial function $\hat{G} : A \to \widetilde{B_\perp}$ satisfying* `forall` $(x : A)\ (y : B),\ f\ x \downarrow y \leftrightarrow \hat{G}\ x\ y$.

Proof. Let us construct a mapping $\hat{G} : A \to \widetilde{B_\perp}$ from G by assuming any $x : A$ and constructing $y : \widetilde{B_\perp}$ such that there (classically) exists $y' : Y$ where $y \downarrow y'$. We apply Lemma 2 to the procedure of performing a case distinction on the constructive existence $\{y : B \mid G\ x\ y\}$, and returns the first projection $[\mathsf{Some}\ y]$ if there exists and returns $[\mathsf{None}]$ if not. This application is feasible as $\{y : B \mid G\ x\ y\}$ despite not being a classical proposition, is a subsingleton type. Proving that this resulting mapping satisfying the desired property is done by applying Example 4. ◀

2.3 Classical Partial Reals

Before concluding this section of introducing the formalization of classical partial functions, we present a special case when the classicalizing monad is applied to partial real numbers.

We define constants and arithmetical operations on classical partial reals $\widetilde{\mathsf{R}_\perp}$ naturally using the monad operations. Being natural here means that we can reason about the operations to be the exact values if and only if all the operands are defined.

An interesting operation that becomes possible using classical partial reals is the reciprocal operation where now we can define $0^{-1} = [\mathsf{None}]$ similarly to Example 3. Another interesting partial operation is the limit operation. Recall that the limit operation we have as primitive for computational R has to be provided a proof that the given sequence is rapidly converging. However, since we can prove for any sequence that there classically exists at most one limit point, we can make the classical partial operation of type $(\mathsf{N} \to \widetilde{\mathsf{R}_\perp}) \to \widetilde{\mathsf{R}_\perp}$ to obtain limit classically for any converging sequences. We further extend the distance function $\mathsf{dist} : \widetilde{\mathsf{R}_\perp} \to \widetilde{\mathsf{R}_\perp} \to \widetilde{\mathsf{R}_\perp}$, the absolute value function $\mathsf{abs} : \widetilde{\mathsf{R}_\perp} \to \widetilde{\mathsf{R}_\perp}$, and so on.

Classical relations on classical partial real numbers are defined in a way that they fail when the partial real numbers are not defined. For example, we define $x < y$ on $\widetilde{\mathsf{R}_\perp}$ such that `exists` $x'\ y' : \mathsf{R},\ x \downarrow x' \wedge y \downarrow y' \wedge x' < y'$. Of course, in the precise Coq formalization, we have to deal with the scopes but to simplify the presentation of this paper, we use the same function names and notations for both classical partial and exact real numbers as long as it is clear from the context or the formulation.

3 Classical Analysis: Continuity and Differentiability

We need a few more definitions from classical analysis to define the problems we are interested in. In particular, we need to define derivatives. We define classical (pointwise) continuity and differentiability closely to the standard definitions from analysis. However, in this work we only need those notions on compact intervals $[-r, r]$ around the origin. Classically, continuity and differentiability on compact domains correspond to their uniform versions. As the uniform versions are much easier to work with formally, we mostly formulate our results with respect to them. In our formal development we also show some results regarding pointwise continuity and differentiability, but we decided to omit them from this paper.

For any $r > 0$, we define the subset type $I\ r := \{x : \mathsf{R} \mid |x| \leq r\}$ with a coercion to R by the first projection. We use the following definitions for (uniform) continuity and differentiability on $I\ r$ of a classical partial function $f : \mathsf{R} \to \widetilde{\mathsf{R}_\perp}$. Note that both properties automatically imply that f is defined on the interval.

```
Definition uniformly_continuous f r := forall ϵ, (ϵ > 0) → exists δ,
  δ > 0 ∧ forall (x y : I r),  dist x y ≤ δ → dist (f x) (f y) ≤ ϵ.

Definition uniform_derivative f g r := forall ϵ, (ϵ > 0) → exists δ,
     δ > 0 ∧ forall (x y : I r), dist x y ≤ δ
     → abs (f y − f x − g x ∗ (y − x)) ≤ ϵ ∗ abs (y − x).
```

We show some classical results, such as that every continuous function is bounded on $I\ r$ and that uniform_derivative f f' r implies that both f and f' are uniformly continuous (and thus bounded). We also show basic properties of derivatives, such as the sum, product and chain rules. Although formal proofs of those statements get quite lengthy, they are very similar to classical textbook proofs and we thus omit them from the paper.

We further define higher derivatives inductively:

```
Fixpoint nth_derivative f g r n :=
    match n with
    | 0 ⇒ forall (x : I r), f x = g x
    | S n′ ⇒ exists f′, uniform_derivative f f′ r ∧ nth_derivative f′ g r n′
    end.
```

▶ Remark 6. By replacing classical existence by the constructive one, we can get constructive versions of the above definitions which can sometimes be useful for functions $f : \mathsf{R} \to \mathsf{R}$. We also show a few constructive statements in our Coq development, but as they are not needed for the results in this paper, we also decided to omit them here.

4 Polynomials and Taylor models

By the Stone-Weierstrass theorem, any continuous function on a closed interval can be uniformly approximated with arbitrarily small error (with respect to the supremum norm) by polynomials. In interval computation, a polynomial approximation together with a rigorous error bound is sometimes called a Taylor model [27]. Taylor models are usually used to approximate smooth functions using Taylor polynomials as the polynomial approximation (hence the name). Many mathematical operations (e.g. arithmetic, integrals, etc.) can be defined on Taylor models. When those operations are applied, dependencies between the variables are retained, giving tighter approximations for function values than simple intervals [26]. Our purpose for using Taylor models is slightly different from how they are used in interval computation. First, as we have exact real numbers in our formalization, we do not need to deal with intervals and can have a real-valued error bound. Secondly, we are mostly interested in sequences of Taylor models, that we use to approximate functions up to any absolute error, instead of mere approximations with fixed error bounds. Consequently, the operations we consider differ from what is usually considered in interval computation.

Before we use polynomials to approximate real valued functions, we need to define the polynomial functions themselves and some operations on them. We identify a polynomial with the list of its coefficients and define evaluation of the polynomial using Horner's scheme.

```
Definition poly := list R.
Fixpoint eval_poly (p : poly) (x : R) := match p with
                                         | []  ⇒ 0
                                         | h ::  t ⇒ h + x ∗ (eval_poly t x)
                                         end.
```

That is, we identify the list $[a_0, a_1, \ldots, a_d]$ with the polynomial function $\sum_{i=0}^{d} a_i x^i$, and the empty list with the zero polynomial. We allow the leading coefficient a_d to be zero, thus the length of the list can be larger than the actual degree of the polynomial. Nonetheless we sometimes use the notion $\deg p$ for the length of the list. As checking if a real number is equal to zero is undecidable, requiring that the leading coefficient is nonzero would lead to rather complicated and unnatural domains for most operations. We further define the norm $\|p\| := \max |p_i|$ and prove some of its properties, which will be useful later.

For most inductive proofs, it is simpler to consider straight-forward evaluation, i.e., compute $\sum_{i=0}^{d} a_i x^i$ by directly summing up the terms of the sum. We therefore also define this evaluation method, and show that they result in the same number.

For polynomials p_1, p_2 and a real number $\lambda : \mathsf{R}$, we define arithmetic operations $p_1 + p_2$, $p_1 \cdot p_2$ and λp_1. We currently do not use a sophisticated algorithm for multiplication. However, the statements are formulated in an abstract way, without explicit mention of the method:

```
Lemma mult_poly p1 p2 : {p3 | forall x, eval_poly p3 x = eval_poly p1 x * eval_poly p2 x}.
```

Thus, if we replace the algorithm encoded in the proof at some point, other parts of the formalization are not affected.

In this paper we are mostly concerned with results over compact intervals. For simplicity, we show most statements only with respect to intervals of the form $[-r, r]$ centered at 0. However, this easily generalizes to arbitrary intervals by shifting the polynomial accordingly:

```
Lemma shift_poly p1 c : {p2 | forall x, eval_poly p2 x = eval_poly p1 (x−c)}.
```

We often need to bound the values of a polynomial on intervals:

```
Lemma bound_polynomial p r : {B | forall x, abs x ≤ r → abs (eval_poly p x) ≤ B}.
```

Again, the explicit bound is only encoded in the proof and not given in the statement, so that it can be replaced easily. For now, we use the simple bound $B := \sum_{i=0}^{d} |a_i| \, r^i$.

For any polynomial $\sum_{i=0}^{d} a_i x^i$, we can compute its derivative $\sum_{i=0}^{d-1} (i+1) a_{i+i} x^i$, which is the uniform derivative of the polynomial on any interval $[-r, r]$.

```
Lemma derive_poly p : {p′ | forall r, uniform_derivative (eval_poly p) (eval_poly p′) r}
```

As a corollary, we also get that polynomials are uniformly continuous.

A central tool to approximate functions by polynomials is Taylor's theorem. We define a polynomial p to be the Taylor polynomial for a function $f : \mathsf{R} \to \widetilde{\mathsf{R}_\perp}$ at $x_0 = 0$ as follows.

```
Definition is_taylor_polynomial p f r := forall n, (n < length p) →
  (exists g, nth_derivative f g r n ∧ nth n p 0 = inv_factorial n * (g 0)).
```

The following variant of Taylor's theorem will be useful later.

```
is_taylor_polynomial p f r M →
    (exists g, nth_derivative f g r (length p) ∧ (forall (x : I r),  abs (g x)  ≤ M))
     → forall (x : I r), dist (eval_poly p x) (f x)
          ≤ inv_factorial (length p) ∗ M ∗ r ^(length p).
```

After having defined polynomials, we can now proceed to define Taylor models. A Taylor model for a function f consists of a polynomial p, a radius r and an error bound ϵ.

▶ **Definition 7.** *We define a Taylor model t :* `tm` *f for a classical partial function $f : \mathsf{R} \to \widetilde{\mathsf{R}_\perp}$ as a record consisting of a polynomial p_t, two real numbers $r_t, \epsilon_t : \mathsf{R}$ and a proof of the specification that $|f\ x - p_t\ x| \leq \epsilon_t$ for all x with $|x| \leq r_t$. The name of the field for r_t is* `tm_radius` *the name of the field for ϵ_t is* `tm_error`*.*

Arithmetic operations on functions can be extended to their Taylor model representation. For example, we show

```
Definition sum_tm f1 f2 (t1 : tm f1) (t2 : tm f2) :  tm (fun x ⇒ f1 x + f2 x).
Definition mult_tm f1 f2 (t1 : tm f1) (t2 : tm f2) :  tm (fun x ⇒ f1 x ∗ f2 x).
```

Proving those statements consists of two parts. The first is to define a polynomial that approximates the resulting function. Here, we can just use the corresponding operations on polynomials. The second is to define the radius and error bound. For both operations we can use the minimum of the two radii as the new radius. For addition we can simply add the error bounds. For multiplication we choose the error bound $\epsilon := B_1\epsilon_{t_1} + B_2\epsilon_{t_2} + \epsilon_1\epsilon_2$, where B_1 and B_2 are bounds for the polynomials p_{t_1} and p_{t_2}, respectively.

Note that we do not prune the degree of the resulting polynomial in our definition of multiplication. This means, however, that operating on Taylor models can increase the degree quickly, leading to performance issues. Thus, it might be necessary to reduce the degree at some point, for which we introduce the following operation.

```
Definition swipe_tm f (t : tm f) (m : N) : {t′ : tm f | deg t′ ≤ m}.
```

Here, deg t is defined as the length of the polynomial. The operation is performed by splitting the polynomial into two polynomials p_1 and p_2 such that $p_t\ x = (p_1\ x) + x^m(p_2\ x)$, compute a bound B for p_2 on the interval and "swiping" the bound into the error, i.e. defining the new Taylor model t' by the polynomial p_1 and error bound $\epsilon_{t'} = \epsilon_t + B$.

5 Exact Computation with Power Series

While Taylor models are interesting for computations with rigorous error bounds, we are mostly interested in representing certain functions exactly in the sense of computable analysis. That is, our main interest is to use sequences of polynomials that allow us to evaluate functions with any desired absolute precision. Formally, we will use a sequence of Taylor models that converges rapidly towards the function. That is, we say a sequence of Taylor models for a function f represents the function on $I\ r$ if all sequence elements have radius at least r and the n-th element has error at most 2^{-n}:

```
Definition represents f (t : nat → (tm f)) r :=
   forall n, (tm_error f (t n)) ≤ 2^−n ∧ (tm_radius f (t n)) ≥ r.
```

Let us proof a few facts about functions represented in that way.

▶ **Lemma 8** (represents_cont). **1.** *Any function that can be represented on $I\ r$ in the above sense is uniformly continuous on $I\ r$.*
2. *Whenever we have representations for f and g, we can compute representations for $f + g$ and fg, respectively.*

Proof. To show the first claim, let $t : \mathsf{N} \to \texttt{tm}\ f$ be a sequence of Taylor models that represents f and let $\epsilon > 0$ be arbitrary. By the Archimedean axiom there is some $n : \mathsf{N}$, such that $2^{-n} < \epsilon$. Choose the polynomial $p := t(n + 2)$ and recall that any polynomial is uniformly continuous. Thus, there is some $\delta > 0$, such that for all $x, y \in I\ r$, $|x - y| \leq \delta \to |(p\ x) - (p\ y)| \leq 2^{-(n+1)}$. But then $|(f\ x) - (f\ y)| = |(f\ x) - (p\ x) + (p\ y) - (f\ y) + (p\ x) - (p\ y)| < \epsilon$ holds.

To prove the second claim, we can simply perform the operations on the corresponding Taylor models and increase the index accordingly. For multiplication, this works as (by 1.) the function is bounded and thus the approximating polynomials can not grow much larger than that bound, showing that the error bound for the product of the approximating polynomials defined above gets arbitrarily small when increasing the index. ◀

Now assume that we have a sequence of Taylor models t representing a function f and a sequence t' of derivatives of t. The notion behaves well with respect to differentiability, in the sense that whenever the sequence t' converges to some function f', then f' is the derivative of f.

```
Lemma differentiable_limit f t f' t' r :
  (represents f t r) →
  (represents f' t' r) →
  (forall n, uniform_derivative_fun (eval_tm (t n)) (eval_tm (t' n))) r) →
  uniform_derivative f f' r.
```

Here, `eval_tm` is the evaluation of the polynomial recorded in the Taylor model.

However, note that in general the sequence of derivatives of a representation does not need to be convergent. Thus, let us now focus on a large class of functions where we can use the theorem to compute the derivatives, namely functions analytic at 0.

Analytic functions allow a canonical approximation by polynomials in the following sense.

A function $f : \mathbb{R} \to \mathbb{R}$ is called analytic at 0 if f is locally defined by a power series, i.e., $f(x) = \sum a_k x^k$ on the interval $(-R, R)$ for some $R > 0$ called radius of convergence. Any analytic function is smooth and the power series coincides with the Taylor series.

We want to represent analytic functions locally by their power series and define a calculus of function operations on that representation. Before we do that, let us first consider infinite sequences $a : \mathsf{N} \to \mathsf{R}$ and series $\sum_{i=0}^{\infty} a_i$ in a general sense. Note that for brevity, we use a_k instead of $(a\ k)$ to denote the k-th element of a sequence $a : \mathsf{N} \to \mathsf{R}$.

For an infinite sequence $a : \mathsf{N} \to \mathsf{R}$ we define a partial sum operation in the obvious way:

```
Fixpoint partial_sum a n := match n with
                            | 0 ⇒ (a 0)
                            | (S n') ⇒ (a n) + partial_sum a n'
                            end.
```

We can then define that the infinite sum $\sum_{i=0}^{\infty} a_k$ classically exists.

```
Definition is_sum a x := forall ϵ, ϵ > 0 →
   exists N, forall n, (n ≥ N) → dist (partial_sum a n) x ≤ ϵ.
```

We show a few classical results about sums, such as that the sum of a sequence is unique. Further, when the sequence decreases rapidly, we can compute the sum.

Let us proceed to power series, that is infinite series of the form $\sum_{n=0}^{\infty} a_k x^k$. For any sequence $a : \mathsf{N} \to \mathsf{R}$ and $x : \mathsf{R}$, we define $\mathsf{ps}\ a\ x :\equiv (\mathsf{fun}\ n \Rightarrow a_n x^n)$. For any power series a, we define a canonical partial function f_a by $f_a\ x = y \leftrightarrow \mathsf{is\_sum}\ (\mathsf{ps}\ a\ x)\ y$. On the other hand, we can also define a series to be the power series for some classical function g by

$$\mathsf{is\_ps\_for}\ g\ a :\Leftrightarrow \exists (r > 0).\ \forall (x : I\ r).\ f_a\ x = g\ x.$$

To compute the value $\sum_{n=0}^{\infty} a_n x^n$, we need to approximate how close the partial sums are to the limit, which in general can not be computed from the series alone [36, Sect. 6.5]. However, if the power series corresponds to some function that is analytic at 0, there is some $R > 0$ such that the series converges for all x with $|x| < R$. This gives us the following fact.

▶ **Fact 1.** *Assume there is some $R > 0$ such that $\sum_{n=0}^{\infty} a_n x^n$ converges whenever $|x| < R$. Then there exist constants $M, r \in \mathbb{R}$ such that $|a_n| \leq M r^{-n}$ for all $n \in \mathbb{N}$.*

Proof. Choose any $0 < r < R$. Then the series converges absolutely on $[-r, r]$. In particular, there is some $M \in \mathbb{R}$ such that $\sum_{i=0}^{\infty} |a_n| r^n = M$. As all summands are positive, $|a_n| r^n \leq M$ must hold for all n and thus the claim holds. ◀

On the other hand, assume we are given constants M, r as in Fact 1. Then for any $x : \mathsf{R}$ with $|x| < r$ and any $N : \mathsf{N}$, we can get the tail estimate

$$\left|\sum_{n=N+1}^{\infty} a_n x^n\right| \leq M \sum_{n=N+1}^{\infty} (\frac{x}{r})^n = M\frac{(\frac{x}{r})^{N+1}}{1-\frac{x}{r}}. \tag{1}$$

In particular, if we choose $|x| \leq \frac{r}{2}$, we get $\left|\sum_{n=N+1}^{\infty} a_n x^n\right| \leq M2^{-N}$, or put differently the sequence fun n ⇒ partial_sum (ps a x) $(n + \lceil \log M \rceil)$ defines a fast Cauchy sequence.

We call the constants M, r from Fact 1 a *series bound* for $(a_k)_{k\in\mathbb{N}}$ and encode power series together with the bound:

```
Record bounded_ps : Type := mk_bounded_ps
{   series : N → R;
    bounded_ps_M : N;
    bounded_ps_r : R;
    bounded_ps_rgt0 : bounded_ps_r > 0;
    bounded_ps_bounded: forall n, abs (a n) ≤ bounded_ps_M * bounded_ps_r^-n }.
```

Note that we chose r to be a real and M to be an integer. The main reason for that is that M is used to find the (integer valued) degree necessary to get a good approximation. If we chose M as real-valued, we could only compute an integer upper bound for $\log M$ non-deterministically, which would have made the statements slightly more complicated.

Using the tail bound from Equation (1), we can evaluate the power series on any $r' < r$. However, for simplicity we chose the fixed evaluation interval $[-\frac{r}{2}, \frac{r}{2}]$:

```
Lemma eval_val (a : bounded_ps) x :
    abs x ≤ (bounded_ps_r a) / 2 → {y | is_sum (ps series a x) y}.
```

To show that this is true, we show that computing the partial sum up to $n + \lceil \log M \rceil$ coefficients defines a fast Cauchy sequence, for which we compute the limit. Similarly, for any $n : \mathsf{N}$, we can canonically transform a bounded power series to a Taylor model for f_a with radius $\frac{r}{2}$ and error bound 2^{-n}. We denote this operation as to_taylor_model.

▶ Remark 9. The choice $\frac{r}{2}$ is somewhat arbitrary, but leads to simple bounds and evaluation is efficient in the region. While it would be possible to replace it by any $r' < r$, in most cases shifting the power series to a new center is more effective.

We next aim to define operations uniformly on power series, in a similar way as we did for Taylor models. In many cases, we can directly generalize results on Taylor models to results on power series by using the following lemma.

```
Lemma approx_ps f a :
    (forall n x, dist (eval_tm (to_taylor_model a n) x) (f x) ≤ 2^-n)
    → is_ps_for f a.
```

Thus, whenever we define an operation on power series, we only need to show that the operation does what it is supposed to do on the Taylor model approximations. Let us demonstrate this by showing that we can add power series. Other, more complicated operations like multiplication can be done similarly.

```
Lemma sum_ps (a b : bounded_ps) : {c : bounded_ps | is_ps_for (f_a + f_b) c }.
```

Proof. We define the bounded power series c by adding the coefficients of a and b and defining $r_c := \min r_a\ r_b$ and $M_c := M_a + M_b$. It is easy to see that this choice is a valid bound for the series. To show that the series converges to the sum, we use approx_ps, thus we have to show that the distance of the 2^{-n} Taylor model approximation to $f_a + f_b$ is at most 2^{-n}. However, the Taylor model is identical to summing Taylor models for $2^{-(n+1)}$ approximations of f_a and f_b, thus the claim holds. ◀

The representation for power series further allows us to compute derivatives.

```
Lemma derive_ps (a : bounded_ps) : {b : bounded_ps | uniform_derivative f_a f_b r_b}.
```

Proof. Let us first show how to bound the power series $b := \sum_{n=0}^{\infty}(n+1)a_{n+1}x^n$ of the derivative. We choose $r_b := \frac{r_a}{2}$. Then

$$|b_n| \leq (n+1)M_a r_a^{-(n+1)} = (n+1)2^{-n}(M_a r_a)r_b^{-n} \leq (M_a r_a)r_b^{-n}.$$

Thus taking any $M_b > M_a r_a$ works. To show that the power series converges to the derivative of f_a, it suffices to show that the sequence of Taylor models of b is a sequence of derivatives of the Taylor models for a. ◀

6 Ordinary Differential Equations

Let us now consider initial value problems (IVPs) for autonomous ordinary differential equations of the form $\dot{y} = f(y(t));\ y(0) = y_0$. We call f the right-hand side function, y_0 the initial value and y the solution of the IVP. It is well known that an initial value problem with analytic right-hand side function has an analytic solution. For simplicity we currently only consider polynomial right-hand side, which already includes many interesting applications. With a few minor adaptions, the same method works for general analytic functions. Note that as polynomials are analytic functions, the solution will again be analytic. However, in general the solution does not need to be polynomial.

In numerics, so-called single step methods such as Euler's method or Runge-Kutta methods are commonly used to solve initial value problems for ordinary differential equations. These methods approximate the solution by taking one step at a time from the current point to the next point, using information from the current point to approximate the solution at the next point. To approximate this solution, often polynomial approximations of a fixed degree are used, and to improve the accuracy the step size is reduced. However, such a method does usually not work well for high precisions as the number of steps grows exponentially with the precision. In exact real computation therefore a different method is more applicable [16, 8]. In this section we show how to implement a variant of this method in our formal development. The method is similar to a single step method, but instead of varying the step size for higher accuracy, we compute a local power series for the solution valid on a small interval around the current point. We can thus keep the step size fixed and get more accurate approximations by using more terms of the power series. This also integrates well with the tools in our library, as the evaluation of the power series can be defined by the limit operation as described in the previous section.

Let us first define a function to be the solution of a polynomial IVP as follows.

```
Definition pivp_solution p y y0 r :=
  (y 0) = y0 ∧ uniform_derivative y (fun t ⇒ (eval_poly p (y t))) r.
```

That is, `pivp_solution p y y0 r` says that the function $y : \mathsf{R} \to \mathsf{R}$ is a solution to the polynomial IVP $\dot{y} = p(y(t));\ y(0) = y_0$ on the interval $(-r, r)$.

It further suffices to only consider the case $y_0 = 0$, as we can always transform a general IVP to an IVP with initial value 0 by replacing $y(t)$ by $y(t) - y_0$ and $p(x)$ by $p(x + y_0)$. Also note that we can always assume that we start at time $t = 0$, as the ODE is autonomous.

Note that, although classically true, we have not formalized the proof of existence and uniqueness of the solution in our Coq development yet. As the focus of this work is to compute the solution, and thus the classical proof is less important, currently our formal statements include the additional assumption that the solution exists, which we plan to remove later. This is, however, irrelevant for the extracted programs as this non-computational assumption is removed in the extracted code anyway.

A well-known method to solve an IVP is the so-called Taylor series method (see e.g. [31]). The idea behind the method is to automatically generate the coefficients of the power series for the solution $y(t)$. To this end, let us define the following sequence of polynomials.

```
Fixpoint pn p n := match n with
                   | 0 ⇒ p
                   | S n' ⇒ n⁻¹ * (p * (derive_poly (pn p n')))
                   end.
```

We show that for each n, the polynomial (pn p n) applied to $y(t)$ gives the $(n+1)$-st derivative of the solution function y divided by $(n+1)!$:

```
  pivp_solution p y 0 r → nth_derivative y (fun t ⇒ (n+1)! * ((pn p n) (y t))) r (n+1)
```

The proof is by induction and is essentially an application of the chain rule for derivatives. It follows that evaluating the polynomial at $y_0 = 0$ gives the coefficients of the Taylor series of the solution:

```
Definition yn p n := match n with
                     | 0 ⇒ 0
                     | S n' ⇒ (eval_poly (pn p n') 0)
                     end.
```

```
Lemma pivp_ps_taylor_series p y r : pivp_solution p y 0 r →
      forall n, (is_taylor_polynomial (to_poly (yn p) n) y r).
```

However, the error bounds for the truncated Taylor series depend on the range of $y(t)$, which we do not know yet. In interval arithmetic, estimating the value of the solution $y(t)$ for an IVP thus usually consists of two steps. The first step is to find a coarse a priori enclosure for $y(t)$ on some interval. This bound can then be used in a second step to find a tighter bound for $y(t)$, e.g., by using it for the error bound in the power series method. Finding a good a priori enclosure is important, as it essentially determines the step size that can be used in the algorithm. An often used simple but effective method is the high order enclosure method [10]. However, its correctness depends on an application of Banach's fixed points theorem, and a formal proof seems challenging. We propose an alternative method, by bounding the coefficients of the Taylor series. Note, however, that a more sophisticated approach would give us better bounds, and thus should be considered in the future. We show

▶ **Lemma 10** (yn_bound). *Let $p = [a_1, \ldots, a_d]$ be a polynomial. For all $n : \mathsf{N}$, we get the bound* $|(\texttt{yn}\ p\ n)| \leq (d^2\|p\|)^n$.

Proof. Let $p = [a_1, \ldots, a_d]$ and recall that $\|p\| = \max|a_k|$. For any polynomials p, q it holds $\|pq\| \leq \deg p\|p\|\|q\|$ and $\|\texttt{derive\_poly}\ p\| \leq \deg p\|p\|$ We show for all n, $\|\texttt{pn}\ p\ n\| \leq (d^2\|p\|)^{n+1}$ from which the claim follows. The proof is by induction on n. For $n = 0$, we get

$\|p\| \leq d^2\|p\|$ which holds trivially. Now assume $\|\texttt{pn}\ p\ n\| \leq (d^2\|p\|)^{n+1}$. We have to show

$$\|\frac{1}{n+1}p \cdot (\texttt{derive\_poly}\ (\texttt{pn}\ p\ n))\| \leq (d^2\|p\|)^{n+2}.$$

We can further show that deg $(\texttt{pn}\ p\ n) = (n+1)d - 2n$. Thus we get

$$\|\frac{1}{n+1}p \cdot (\texttt{derive\_poly}\ (\texttt{pn}\ p\ n))\| \leq \frac{1}{n+1}\|p\|(n+1)d^2\|(\texttt{pn}\ p\ n)\|$$

from which the claim follows. ◀

Thus, putting the above results together, yn defines a bounded power series for the solution y with bounds $r = \frac{1}{(d^2\|p\|)}$ and $M = 1$.

```
Definition pivp_ps_exists p :
 {a:bounded_ps | exists r, r > 0 ∧ forall y, pivp_solution p y y0 r → is_ps_for (y−y0) a}.
```

Note that this also shows that the resulting function is analytic, as we can compute its power series and a positive lower bound on its radius of convergence. We can then use this bounded power series to compute a local solution $y(t)$ for some small $t > 0$. An important property of this method is that we do not need to evaluate the function on additional steps to increase the precision. More precisely, the precision is controlled by applying the limit operator on the sequence of partial sums of the power series at the point t. Thus, we get an exact representation of the number $y(t)$ as a converging sequence of values that only depend on the power series at 0. In contrast, when using the Euler or Runge-Kutta methods, the function needs to be evaluated on additional points to increase the precision, and the methods are therefore harder to adapt to our system.

Next, we want to extend the solution to a larger interval. To this end, we first define an operation that gives us one specific pair $(t_1, y(t_1))$ with $y_1 > 0$ from the local solution.

```
Lemma local_solution p y0 :
   {ty1 : R ∗ R | (fst ty) > 0 ∧
      exists r, r > 0 ∧ forall y, pivp_solution p y y0 r → (snd ty1) = (y (fst ty1))}.
```

The method works by simply applying the evaluation algorithm for the power series on the largest possible point in the evaluation interval and adding y_0.

▶ **Example 11.** Consider the initial value problem

$$\dot{y}(t) = 1 + y(t)^2 \ ; \ y(0) = y_0.$$

It has the explicit solution $y(t) = \tan(t + \arctan y_0)$. From the term (local_solution $[1\ 0\ 1]$ y_0) we can extract a program that computes the pair (t_1, y_1). The extracted program proceeds as follows. The first step is to transform the system to an equivalent one with initial value 0. Application of the shifting procedure yields the new system $\dot{y}(t) = p(y(t))$; $y(0) = 0$ with $p(x) = 1 + y_0^2 + 2y_0x + x^2$. The operator then computes the power series of the local solution. We get the evaluation interval $\frac{1}{8\|p\|}$ where $\|p\| = \max\ (1 + y_0^2)\ (2y_0)$ for the power series, which is a rather coarse under-approximation of the actual radius of convergence $\frac{\pi}{2} - \arctan y_0$ of the series. The solution operator will produce the value and time for this maximal point in the evaluation domain.

Next, we can turn this into a single step method to solve the IVP on a larger interval by repeatedly applying the local solution operator to get a new initial value.

That is, we prove the following statement.

```
Lemma solve_ivp p y0 n : {l : list (R * R) |
    length l = n+1 ∧
    forall m, m < n → (fst (nth m l (0,0)) < fst (nth (m+1) l (0,0)))
    forall y r, pivp_solution p y y0 r → forall ty, In ty l → (snd ty) = (y (fst ty))}.
```

The method produces a list $[(t_0, y_0), \ldots, (t_n, y_n)]$ such that the t_0 are increasing and $y(t_i) = y_i$. The step size is adaptive as when the growth of the function is faster in a region, the evaluation radius we compute gets smaller. Note that each $y(t_i)$ is given as an exact representation of the number and thus the step size only depends on parameters of the function and the initial value and not on the desired precision of the solution. Thus, we do not need to consider error propagation due to working with approximations, allowing for a simple proof of correctness.

If we consider again Example 11, we can see that starting with $y_0 = 0$, in each step the interval will get smaller and smaller as we approach $\frac{\pi}{2}$.

7 Conclusion and Future Work

We presented an extension of our formalization of exact real computation to include rigorous approximations of classical partial functions by polynomials, analytic functions and solutions to initial value problems for non-linear polynomial ordinary differential equations. One of the main limitations of the current work is that we only consider univariate functions, while most interesting problems for differential equations occur at higher dimensions. Technically, all the algorithms presented in this paper generalize directly to the multivariate setting and even certain forms of partial differential equations [34]. Extending the formalization should therefore not be too challenging, although it might require formalizing additional results from multivariate analysis. Another possible extension for which one dimensional functions suffice, is to consider holonomic functions. A function is holonomic if it is the solution to a linear differential equation of the form $p_r(x)f^{(r)}(x) + p_{r-1}(x)f^{(r-1)}(x) + \cdots + a_1(x)f'(x) + a_0(x)f(x) = 0$ where $p_0, \ldots, p_r$ are polynomials. Similar to the analytic case, it is possible to compute the power series of solutions to such equations [15] and thus it might be simpler to include in our formalization than extending to higher dimension.

Lastly, many of the algorithms currently encoded in the proofs are rather simple and not very efficient. Replacing them by more sophisticated algorithms would yield better extracted programs. In particular, the bound we currently use for the radius of convergence of an ODE solution is a very coarse approximation, and improving this bound would increase the step size and thus the efficiency of the algorithm drastically. Thus, we plan to verify better methods such as the higher-order enclosure method [10] in future work. Due to the abstract formulation of most results, exchanging algorithms in one part should have little effect on other parts of the formalization, allowing to replace those methods easily in the future.

References

1 Reynald Affeldt, Yves Bertot, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, et al. Mathcomp-analysis: Mathematical components compliant analysis library, 2022.

2 Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, pages 209–229, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

3 A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A.L. Sangiovanni-Vincentelli. Ariadne: a Framework for Reachability Analysis of Hybrid Automata. In *Proc. 17th Int. Symp. on Mathematical Theory of Networks and Systems*, Kyoto, 2006.

4 Errett Bishop and Douglas Bridges. *Constructive analysis*, volume 279. Springer Science & Business Media, 2012.

5 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. `doi:10.1007/s11786-014-0181-1`.

6 Franz Brauße, Pieter Collins, and Martin Ziegler. Computer science for continuous data. In François Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 62–82, Cham, 2022. Springer International Publishing.

7 Franz Brauße, Margarita Korovina, and Norbert Müller. Using taylor models in exact real arithmetic. In *Revised Selected Papers of the 6th International Conference on Mathematical Aspects of Computer and Information Sciences - Volume 9582*, MACIS 2015, pages 474–488, Berlin, Heidelberg, 2015. Springer-Verlag. `doi:10.1007/978-3-319-32859-1_41`.

8 Franz Brauße, Margarita Korovina, and Norbert Th Müller. Towards using exact real arithmetic for initial value problems. In *Perspectives of System Informatics: 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers 10*, pages 61–74. Springer, 2016. `doi:10.1007/978-3-319-41579-6_6`.

9 Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 258–263, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

10 George F Corliss and Robert Rihm. Validating an a priori enclosure using high-order taylor series. *Mathematical Research*, 90:228–238, 1996.

11 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer, 2004. `doi:10.1007/978-3-540-27818-4_7`.

12 Laurent Doyen, Goran Frehse, George J. Pappas, and André Platzer. *Verification of Hybrid Systems*, pages 1047–1110. Springer International Publishing, Cham, 2018. `doi:10.1007/978-3-319-10575-8_30`.

13 Fabian Immler. *A Verified ODE Solver and Smale's 14th Problem (Ein Verifizierter GDGL-Löser und Smales 14. Problem)*. PhD thesis, Technical University of Munich, Germany, 2018. URL: `https://mediatum.ub.tum.de/1422071`.

14 Fabian Immler. A verified ODE solver and the lorenz attractor. *J. Autom. Reason.*, 61(1-4):73–111, 2018. `doi:10.1007/S10817-017-9448-Y`.

15 Manuel Kauers. *D-finite Functions*, volume 30. Springer Nature, 2023.

16 Akitoshi Kawamura, Florian Steinberg, and Holger Thies. Parameterized complexity for uniform operators on multidimensional analytic functions and ODE solving. In *International Workshop on Logic, Language, Information, and Computation*, pages 223–236. Springer, 2018. `doi:10.1007/978-3-662-57669-4_13`.

17 Michal Konečný, Sewon Park, and Holger Thies. Axiomatic reals and certified efficient exact real computation. In *International Workshop on Logic, Language, Information, and Computation*, pages 252–268. Springer, 2021. `doi:10.1007/978-3-030-88853-4_16`.

18 Michal Konečný, Sewon Park, and Holger Thies. Certified computation of nondeterministic limits. In *NASA Formal Methods Symposium*, pages 771–789. Springer, 2022. `doi:10.1007/978-3-031-06773-0_41`.

19 Michal Konečný, Sewon Park, and Holger Thies. Extracting efficient exact real number computation from proofs in constructive type theory. *arXiv preprint arXiv:2202.00891*, 2022.

20 Michal Konečný, Sewon Park, and Holger Thies. cAERN: Axiomatic Reals and Certified Efficient Exact Real Computation. `https://github.com/holgerthies/coq-aern`, 2024.

21 Michal Konečný, Sewon Park, and Holger Thies. Formalizing Hyperspaces for Extracting Efficient Exact Real Computation. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, volume 272 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2023.59.

22 Michal Konečný. aern2-real: A Haskell library for exact real number computation. https://hackage.haskell.org/package/aern2-real, 2021.

23 Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dreach: δ-reachability analysis for hybrid systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

24 Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *J. Autom. Reason.*, 62(2):281–300, 2019. doi:10.1007/S10817-018-9463-7.

25 Evgeny Makarov and Bas Spitters. The picard algorithm for ordinary differential equations in coq. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 463–468, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39634-2_34.

26 Kyoko Makino and Martin Berz. Efficient control of the dependency problem based on taylor model methods. *Reliable Computing*, 5(1):3–12, 1999. doi:10.1023/A:1026485406803.

27 Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 6:239–316, 2003.

28 Érik Martin-Dorel, Laurence Rideau, Laurent Théry, Micaela Mayero, and Ioana Pasca. Certified, efficient and sharp univariate taylor models in coq. In *Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '13, pages 193–200, USA, 2013. IEEE Computer Society. doi:10.1109/SYNASC.2013.33.

29 Norbert Th Müller. Constructive aspects of analytic functions. In *Proc. Workshop on Computability and Complexity in Analysis*, volume 190, pages 105–114, 1995.

30 Norbert Th Müller. The iRRAM: Exact arithmetic in C++. In *International Workshop on Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. doi:10.1007/3-540-45335-0_14.

31 Nedialko S Nedialkov. Interval tools for odes and daes. In *12th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006)*, pages 4–4. IEEE, 2006.

32 Sewon Park, Franz Brauße, Pieter Collins, SunYoung Kim, Michal Konečný, Gyesik Lee, Norbert Müller, Eike Neumann, Norbert Preining, and Martin Ziegler. Semantics, specification logic, and hoare logic of exact real computation, 2024. arXiv:1608.05787.

33 Arno Pauly and Florian Steinberg. Comparing representations for function spaces in computable analysis. *Theory Comput. Syst.*, 62(3):557–582, 2018. doi:10.1007/S00224-016-9745-6.

34 Svetlana Selivanova, Florian Steinberg, Holger Thies, and Martin Ziegler. Exact real computation of solution operators for linear analytic systems of partial differential equations. In *Computer Algebra in Scientific Computing: 23rd International Workshop, CASC 2021, Sochi, Russia, September 13–17, 2021, Proceedings 23*, pages 370–390. Springer, 2021. doi:10.1007/978-3-030-85165-1_21.

35 Christoph Traut and Fabian Immler. Taylor models. *Arch. Formal Proofs*, 2018, 2018. URL: https://www.isa-afp.org/entries/Taylor_Models.html.

36 K. Weihrauch. Computable analysis. Springer, Berlin, 2000. doi:10.1007/978-3-642-56999-9.