Rigorous Polynomial Approximation using Taylor Models in Coq*

Nicolas Brisebarre[1], Mioara Joldeş[4], Érik Martin-Dorel[1], Micaela Mayero[1,2], Jean-Michel Muller[1], Ioana Paşca[1], Laurence Rideau[3], and Laurent Théry[3]

[1] LIP, CNRS UMR 5668, ENS de Lyon, INRIA Grenoble - Rhône-Alpes, UCBL, Arénaire, Lyon, F-69364
[2] LIPN, UMR 7030, Université Paris 13, LCR, Villetaneuse, F-93430
[3] Marelle, INRIA Sophia Antipolis - Méditerranée, Sophia Antipolis, F-06902
[4] CAPA, Dpt. of Mathematics, Uppsala Univ., Box S-524, 75120, Uppsala, Sweden

Abstract. One of the most common and practical ways of representing a real function on machines is by using a polynomial approximation. It is then important to properly handle the error introduced by such an approximation. The purpose of this work is to offer guaranteed error bounds for a specific kind of rigorous polynomial approximation called Taylor model. We carry out this work in the Coq proof assistant, with a special focus on genericity and efficiency for our implementation. We give an abstract interface for rigorous polynomial approximations, parameterized by the type of coefficients and the implementation of polynomials, and we instantiate this interface to the case of Taylor models with interval coefficients, while providing all the machinery for computing them. We compare the performances of our implementation in Coq with those of the Sollya tool, which contains an implementation of Taylor models written in C. This is a milestone in our long-term goal of providing fully formally proved and efficient Taylor models.

Keywords: certified error bounds, Taylor models, Coq proof assistant, rigorous polynomial approximation

1 Rigorous Approximation of Functions by Polynomials

It is frequently useful to replace a given function of a real variable by a simpler function, such as a polynomial, chosen to have values close to those of the given function, since such an approximation may be more compact to represent and store but also more efficient to evaluate and manipulate. As long as evaluation is concerned, polynomial approximations are especially important. In general the basic functions that are implemented in hardware on a processor are limited to addition, subtraction, multiplication, and sometimes division. Moreover, division is significantly slower than multiplication. The only functions of one variable

* This research was supported by the TaMaDi project of the French ANR (ref. ANR-2010-BLAN-0203-01).

that one may evaluate using a bounded number of additions/subtractions, multiplications and comparisons are piecewise polynomials: hence, on such systems, polynomial approximations are not only a good choice for implementing more complex functions, they are frequently the only one that makes sense.

Polynomial approximations for widely used functions used to be tabulated in handbooks [1]. Nowadays, most computer algebra systems provide routines for obtaining polynomial approximations of commonly used functions. However, when bounds for the approximation errors are available, they are not guaranteed to be accurate and are sometimes unreliable.

Our goal is to provide efficient and quickly computable *rigorous polynomial approximations*, i.e., polynomial approximations for which (i) the provided error bound is tight and not underestimated, (ii) the framework is suitable for formal proof (indeed, the computations are done in a formal proof checker), while requiring computation times similar to those of a conventional C implementation.

1.1 Motivations

Most numerical systems depend on standard functions like exp, sin, etc., which are implemented in libraries called `libm`s. These `libm`s must offer guarantees regarding the provided accuracy: they are heavily tested before being published, but for precisions higher that single precision, an exhaustive test is impossible [16]. Hence a proof of the behavior of the program that implements a standard function should come with it, whenever possible. One of the key elements of such a proof would be the guarantee that the used polynomial approximation is within some threshold from the function. This requirement is even more important when *correct rounding* is at stake. Most `libm`s do not provide correctly rounded functions, although the IEEE 754-2008 Standard for Floating-Point (FP) Arithmetic [22] recommends it for a set of basic functions. Implementing a correctly rounded function requires rigorous polynomial approximations at two steps: when actually implementing the function in a given precision, and—before that—when trying to solve the *table maker's dilemma* for that precision.

The 1985 version of the IEEE Standard for FP Arithmetic requires that the basic operations ($+$, $-$, $\times$, $\div$, and $\sqrt{\cdot}$) should produce *correctly rounded results*, as if the operations were first carried out in infinite precision and these intermediate results were then rounded. This contributed to a certain level of portability and provability of FP algorithms. Until 2008, there was no such analogous requirement for standard functions. The main impediment for this was the *table maker's dilemma*, which can be stated as follows: consider a function f and a FP number x. In most cases, $y = f(x)$ cannot be represented exactly. The correctly rounded result is the FP number that is closest to y. Using a finite precision environment, only an approximation $\hat{y}$ to y can be computed. If that approximation is not accurate enough, one cannot decide the correct rounding of y from $\hat{y}$. Ziv [41] suggested to improve the accuracy of the approximation until the correctly rounded value can be decided. A first improvement over that approach derives from the availability of tight bounds on the worst-case accuracy required to compute some functions [25], which made it possible to write a `libm`

with correctly rounded functions, where correct rounding is obtained at modest additional costs [37]. The TaMaDi project [32] aims at computing the worst-case accuracy for the most common functions and formats. Doing this requires very accurate polynomial approximations that are formally verified.

Beside the Table Maker's Dilemma, the implementation of correctly rounded elementary functions is a complex process, which includes finding polynomial approximations for the considered function that are accurate enough. Obtaining good polynomial approximations is detailed in [10,9,12]. In the same time, the approximation error between the function and the polynomial is very important since one must make sure that the approximation is good enough. The description of a fast, automatic and verifiable process was given in [23].

In the context of implementing a standard function, we are interested in finding polynomial approximations for which, given a degree n, the maximum error between the function and the polynomial is minimum: this "minimax approximation" has been broadly developed in the literature and its application to function implementation is discussed in detail in [12,33]. Usually this approximation is computed numerically [38], so an a posteriori error bound is needed. Obtaining a tight bound for the approximation error reduces to computing a tight bound for the supremum norm of the error function over the considered interval. Absolute error as well as relative errors can be considered. For the sake of simplicity, in this paper, we consider absolute errors only (relative errors would be handled similarly). Our problem can be seen as a univariate rigorous global optimization problem, however, obtaining a tight and formally verified interval bound for the supremum norm of the error function presents issues unsuspected at a first sight [14], so that techniques like interval arithmetic and Taylor models are needed. An introduction to these concepts is given below.

Interval arithmetic and Taylor models. The usual arithmetic operations and functions are straightforwardly extended to handle intervals. One use of interval arithmetic is bounding the image of a function over an interval. Interval calculations frequently overestimate the image of a function. This phenomenon is in general proportional to the width of the input interval. We are therefore interested in using thin input intervals in order to get a tight bound on the image of the function. While subdivision methods are successfully used in general, when trying to solve this problem, one is faced with what is known as a "dependency phenomenon": since function f and its approximating polynomial p are highly correlated, branch and bound methods based on using intervals of smaller width to obtain less overestimation, end up with an unreasonably high number of small intervals. To reduce the dependency, *Taylor models* are used. They are a basic tool for replacing functions with a polynomial and an interval remainder bound, on which basic arithmetic operations or bounding methods are easier.

1.2 Related work

Taylor models [27,35,28] are used for solving rigorous global optimization problems [27,5,14,6] and obtaining validated solutions of ODEs [34] with applica-

tions to critical systems like particle accelerators [6] or robust space mission design [26]. Freely available implementations are scarce. One such implementation is available in Sollya [13]. It handles univariate functions only, but provides multiple-precision support for the coefficients. It was used for proving the correctness of supremum norms of approximation errors in [14], and so far it is the only freely available tool that provides such routines. However, this remains a C implementation that does not provide formally proved Taylor models, although this would be necessary for having a completely formally verified algorithm.

There have been several attempts to formalize Taylor models (TMs) in proof assistants. An implementation of multivariate TMs is presented in [42]. They are implemented on top of a library of exact real arithmetic, which is more costly than FP arithmetic. The purpose of that work is different than ours. It is appropriate for multivariate polynomials with small degrees, while we want univariate polynomials and high degrees. There are no formal proofs for that implementation. An implementation of univariate TMs in PVS is presented in [11]. Though formally proved, it contains ad-hoc models for a few functions only, and it is not efficient enough for our needs, as it is unable to produce Taylor models of degree higher than 6. Another formalization of Taylor models in Coq is presented in [15]. It uses polynomials with FP coefficients. However, the coefficients are axiomatized, so we cannot compute the actual Taylor model in that implementation. We can only talk about the properties of the involved algorithms.

Our purpose is to provide a modular implementation of univariate Taylor models in Coq, which is efficient enough to produce very accurate approximations of elementary real functions. We start by presenting in Section 2 the mathematical definitions of Taylor models as well as efficient algorithms used in their implementation. We then present in Section 3 the Coq implementation. Finally we evaluate in Section 4 the quality of our implementation, both from the point of view of efficient computation and of numerical accuracy of the results.

2 Presentation of the Taylor Models

2.1 Definition, Arithmetic

A Taylor model (TM) of order n for a function f which is supposed to be $n+1$ times differentiable over an interval $[a, b]$, is a pair $(T, \boldsymbol{\Delta})$ formed by a polynomial T of degree n, and an interval part $\boldsymbol{\Delta}$, such that $f(x) - T(x) \in \boldsymbol{\Delta}, \forall x \in [a, b]$. The polynomial can be seen as a Taylor expansion of the function at a given point. The *interval remainder* $\boldsymbol{\Delta}$ provides an enclosure of the approximation errors encountered (truncation, roundings).

For usual functions, the polynomial coefficients and the error bounds are computed using the Taylor-Lagrange formula and recurrence relations satisfied by successive derivatives of the functions. When using the same approach for composite functions, the error we get for the remainder is too pessimistic [14]. Hence, an arithmetic for TMs was introduced: simple algebraic rules like addition, multiplication and composition with TMs are applied recursively on the

structure of function f, so that the final model obtained is a TM for f over $[a, b]$. Usually, the use of these operations with TMs offers a much tighter error bound than the one directly computed for the whole function [14]. For example, addition is defined as follows: let two TMs of order n for f_1 and f_2, over $[a, b]$: $(P_1, \boldsymbol{\Delta}_1)$ and $(P_2, \boldsymbol{\Delta}_2)$. Their sum is an order n TM for $f_1 + f_2$ over $[a, b]$ and is obtained by adding the two polynomials and the remainder bounds: $(P_1, \boldsymbol{\Delta}_1) + (P_2, \boldsymbol{\Delta}_2) = (P_1 + P_2, \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2)$. For multiplication and composition, similar rules are defined.

We follow the definitions in [23,14], and represent the polynomial T with *tight interval coefficients*. This choice is motivated by the ease of programming (rounding errors are directly handled by the interval arithmetic) and also by the fact that we want to ensure that the true coefficients of the Taylor polynomial lie inside the corresponding intervals. This is essential for applications that need to handle removable discontinuities [14]. For our formalization purpose, we recall and explain briefly in what follows the definition of valid Taylor models [23, Def. 2.1.3], and refer to [23, Chap. 2] for detailed algorithms regarding operations with Taylor models for univariate functions.

2.2 Valid Taylor Models

A Taylor model for a function f is a pair $(T, \boldsymbol{\Delta})$. The relation between f and $(T, \boldsymbol{\Delta})$ can be rigorously formalized as follows.

Definition 1. *Let $f : \boldsymbol{I} \to \mathbb{R}$ be a function, $\boldsymbol{x_0}$ be a small interval around an expansion point x_0. Let T be a polynomial with interval coefficients $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and $\boldsymbol{\Delta}$ an interval. We say that $(T, \boldsymbol{\Delta})$ is a Taylor model of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ when*

$$\begin{cases} \boldsymbol{x_0} \subseteq \boldsymbol{I} \text{ and } 0 \in \boldsymbol{\Delta}, \\ \forall \xi_0 \in \boldsymbol{x_0}, \exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta}, f(x) - \sum_{i=0}^{n} \alpha_i \, (x - \xi_0)^i = \delta. \end{cases}$$

Informally, this definition says that there is always a way to pick some values α_i in the intervals a_i so that the difference between the resulting polynomial and f around $\boldsymbol{x_0}$ is contained in $\boldsymbol{\Delta}$. This validity is the invariant that is preserved when performing operations on Taylor models. Obviously, once a Taylor model $(T, \boldsymbol{\Delta})$ is computed, if needed, one can get rid of the interval coefficients $\boldsymbol{a_i}$ in T by picking arbitrary α_i and accumulating in $\boldsymbol{\Delta}$ the resulting errors.

2.3 Computing the Coefficients and the Remainder

We are now interested in an automatic way of providing the terms $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and $\boldsymbol{\Delta}$ of Definition 1 for basic functions. It is classical to use the following

Lemma 1 (Taylor-Lagrange Formula). *If f is $n+1$ times differentiable on a domain $\boldsymbol{I}$, then we can expand f in its Taylor series around any point $x_0 \in \boldsymbol{I}$*

and we have: $\forall x \in \boldsymbol{I}$, $\exists \xi$ *between* x_0 *and* x *such that*

$$f(x) = \underbrace{\left(\sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!} (x - x_0)^i \right)}_{T(x)} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1}}_{\Delta(x,\xi)}.$$

Computing interval enclosures $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$, for the coefficients of T, reduces to finding enclosures of the first n derivatives of f at x_0 in an efficient way. The same applies for computing $\boldsymbol{\Delta}$ based on an interval enclosure of the $n+1$ derivative of f over $\boldsymbol{I}$. However, the expressions for successive derivatives of practical functions typically become very involved with increasing n. Fortunately, it is not necessary to generate these expressions for obtaining values of $\{f^{(i)}(x_0), i = 0, \ldots, n\}$. For basic functions, formulas are available since Moore [31] (see also [21]). There one finds either recurrence relations between successive derivatives of f, or a simple closed formula for them. And yet, this is a case-by-case approach, and we would like to use a more generic process, which would allow us to deal with a broader class of functions in a more uniform way suitable to formalization.

Recurrence Relations for D-finite Functions. An algorithmic approach exists for finding recurrence relations between the Taylor coefficients for a class of functions that are solutions of linear ordinary differential equations (LODE) with polynomial coefficients, called *D-finite functions*. The Taylor coefficients of these functions satisfy a linear recurrence with polynomial coefficients [40]. Most common functions are *D-finite*, while a simple counter-example is tan. For any D-finite function one can generate the recurrence relation directly from the differential equation that defines the function, see, e.g., the Gfun module in Maple [39]. From the recurrence relation, the computation of the first n coefficients is done in linear time. Let us take a simple example and consider $f = \exp$. It satisfies the LODE $f' = f$, $f(0) = 1$, which gives the following recurrence for the Taylor coefficients $(c_n)_{n \in \mathbb{N}}$: $(n+1)c_{n+1} - c_n = 0$, $c_0 = 1$, whose solution is $c_n = 1/n!$.

This property lets us include in the class of *basic functions* all the D-finite functions. We will see in Section 3.2 that this allows us to provide a uniform and efficient approach for computing Taylor coefficients, suitable for formalization. We note that our data structure for that is *recurrence relation + initial conditions* and that the formalization of the isomorphic transformation from the *LODE + initial conditions*, used as input in Gfun is subject of future research.

3 Formalization of Taylor Models in Coq

We provide an implementation[5] of TMs that is efficient enough to produce very accurate approximating polynomials in a reasonable amount of time. The work is carried out in the Coq proof assistant, which provides a formal setting where we will be able to formally verify our implementation. We wish to be as generic

[5] It is available at `http://tamadi.gforge.inria.fr/CoqApprox/`

as possible. A TM is just an instance of a more general object called *rigorous polynomial approximation* (RPA). For a function f, a RPA is a pair $(T, \boldsymbol{\Delta})$ where T is a polynomial and $\boldsymbol{\Delta}$ an interval containing the approximation error between f and T. We can choose Taylor polynomials for T and get TMs but other types of approximation are also available like Chebyshev models, based on Chebyshev polynomials. This generic RPA structure will look like:

```
Structure rpa := { approx: polynomial; error: interval }
```

In this structure, we want genericity not only for `polynomial` with respect to the type of its coefficients and to its physical implementation but also for the type for intervals. Users can then experiment with different combinations of datatypes. Also, this genericity lets us factorize our implementation and will hopefully facilitate the proofs of correctness. We implement Taylor models as an instance of a generic RPA following what is presented in Section 2. Before describing our modular implementation, we present the Coq proof assistant, the libraries we have been using and how computation is handled.

3.1 The Coq proof assistant

Coq [4] is an interactive theorem prover that combines a higher-order logic and a richly-typed functional programming language. Thus, it provides an expressive language for defining not only mathematical objects but also datatypes and algorithms and for stating and proving their properties. The user builds proofs in Coq in an interactive manner. In our development, we use the SSReflect [19] extension that provides its own tactic language and libraries.

There are two main formalizations of real numbers in Coq: an axiomatic one [29] and a constructive one [18]. For effective computations, several implementations of computable real numbers exist. A library for multiple-precision FP arithmetic is described in [8]. Based on this library, an interval arithmetic library is defined in [30]. It implements intervals with FP bounds. Also, the libraries [36] and [24] provide an arbitrary precision real arithmetic. All these libraries are proved correct by deriving a formal link between the computational reals and one of the formalizations of real numbers. We follow the same idea: implement a computable TM for a given function and formally prove its correctness with respect to the abstract formalization of that function in Coq. This is done by using Definition 1 and the functions defined in the axiomatic formalization.

The logic of Coq is computational: it is possible to write programs in Coq that can be directly executed within the logic. This is why the result of a computation with a correct algorithm can always be trusted. Thanks to recent progress in the evaluation mechanism [7], a program in Coq runs as fast as an equivalent version written directly and compiled in OCaml. There are some restrictions to the programs that can be executed in Coq: they must always terminate and be purely functional, i.e., no side-effects are allowed. This is the case for the above mentioned computable real libraries. Moreover, they are defined within Coq on top of the multiple-precision arithmetic library based on binary tree described

in [20]. So only the machine modular arithmetic (32 or 64 bits depending on the machine) is used in the computations in Coq.

For our development of Taylor models we use polynomials with coefficients being some kind of computable reals. Following the description in Section 2, we use intervals with FP bounds given by [30] as coefficients. Since the interval and FP libraries are proved correct, so is the arithmetic on our coefficients. By choosing a functional implementation for polynomials (e.g., lists), we then obtain TMs that are directly executable within Coq. Now, we describe in detail this modular implementation.

3.2 A Modular Implementation of Taylor Models

Coq provides three mechanisms for modularization: *type classes*, *structures*, and *modules*. Modules are less generic than the other two (that are first-class citizens) but they have a better computational behavior: module applications are performed statically, so the code that is executed is often more compact. Since our generic implementation only requires simple parametricity, we have been using modules.

First, abstract interfaces called `Module Type`s are defined. Then concrete "instances" of these abstract interfaces are created by providing an implementation for all the fields of the `Module Type`. The definition of `Module`s can be parameterized by other `Module`s. These parameterized modules are crucial to factorize code in our structures.

Abstract polynomials, coefficients and intervals. We describe abstract interfaces for *polynomials* and for their *coefficients* using Coq's `Module Type`. The interface for coefficients contains the common base of all the computable real numbers we may want to use. Usually coefficients of a polynomial are taken in a ring. We cannot do this here. For example, addition of two intervals is not associative. Therefore, the abstract interface for coefficients contains the required operations (addition, multiplication, etc.) only, where some basic properties (associativity, distributivity, etc.) are ruled out. The case of abstract polynomials is similar. They are also a `Module Type` but this time parameterized by the coefficients. The interface contains only the operations on polynomials (addition, evaluation, iterator, etc.) with the properties that are satisfied by all common instantiations of polynomials. For intervals, we directly use the abstract interface provided by the Coq.Interval library [30].

Rigorous polynomial approximations. We are now able to give the definition of our rigorous polynomial approximation.

```
Module RigPolyApprox (C : BaseOps)(P : PolyOps C)(I : IntervalOps).
Structure rpa : Type := RPA { approx : P.T;  error : I.type }.
```

The module is parameterized by `C` (the coefficients), by `P` (the polynomials with coefficients in `C`), and by `I` (the intervals).

Generic Taylor polynomials. Before implementing our Taylor models, we use the abstract coefficients and polynomials to implement generic Taylor polynomials. These polynomials are computed using an algorithm based on recurrence relations as described in Section 2.3. This algorithm can be implemented in a generic way. It takes as argument the relation between successive coefficients, the initial conditions and outputs the Taylor polynomial.

We detail the example of the exponential, which was also presented in Section 2.3. The Taylor coefficients $(c_n)_{n\in\mathbb{N}}$ satisfy $(n+1)c_{n+1} - c_n = 0$. The corresponding Coq code is

```
Definition exp_rec (n : nat) u := tdiv u (tnat n).
```

where `tdiv` is the division on our coefficients and `tnat` is an injection of integers to our type of coefficients. We then implement the generic Taylor polynomial for the exponential around a point x_0 with the following definition.

```
Definition T_exp n x0 := trec1 exp_rec (texp x0) n.
```

In this definition, `trec1` is the function in the polynomial interface that is in charge of producing a polynomial of size `n` from a recurrence relation of order 1 (here, `exp_rec`) and an initial condition (here, `texp x0`, the value of the exponential at `x0`). The interface also contains `trec2` and `trecN` for producing polynomials from recurrences of order 2 and order N with the appropriate number of initial conditions. Having specific functions for recurrences of order 1 and 2 makes it possible to have optimized implementations for these frequent recurrences. All the functions we currently dispose of in our library are in fact defined with `trec1` and `trec2`. We provide generic Taylor polynomials for constant functions, identity, $x \mapsto \frac{1}{x}$, $\sqrt{\cdot}$, $\frac{1}{\sqrt{\cdot}}$, exp, ln, sin, cos, arcsin, arccos, arctan.

Taylor models. We implement TMs on top of the RPA structure by using polynomials with coefficients that are intervals with FP bounds, according to Section 2. Yet we are still generic with respect to the effective implementation of polynomials. For the remainder, we also use *intervals with FP bounds*. This datatype is provided by the Coq.Interval library [30], whose design is also based on modules, in such a way that it is possible to plug all the machinery on the desired kind of Coq integers (i.e., `Z` or `BigZ`).

In a TM for a basic function (e.g., exp), polynomials are instances of the generic Taylor polynomials implemented with the help of the recurrence relations. The remainder is computed with the help of the Taylor-Lagrange formula in Lemma 1. For this computation, thanks to the parameterized module, we reuse the generic recurrence relations. The order-n Taylor model for the exponential on interval `X` expanded at the small interval `X0` is as follows:

```
Definition TM_exp (n : nat) X X0 :=
  RPA (T_exp n X0) (Trem T_exp n X X0).
```

We implement Taylor models for the addition, multiplication, and composition of two functions by arithmetic manipulations on the Taylor models of the two functions, as described in Section 2. Here is the example of addition:

```
Definition TM_add (Mf Mg : rpa) :=
  RPA (P.tadd (approx Mf) (approx Mg))
      (I.add (error Mf) (error Mg)).
```

The polynomial approximation is just the sum of the two approximations and the interval error is the sum of the two errors. Multiplication is almost as intuitive. We consider the truncated multiplication of the two polynomials and we make sure that the error interval takes into account the remaining parts of the truncated multiplication. Composition is more complex. It uses addition and multiplication of Taylor polynomials. Division of Taylor models is implemented in term of multiplication and composition with the inverse function $x \mapsto 1/x$. The corresponding algorithms are fully described in [23].

Discussion on the formal verification of Taylor models. The Taylor model `Module` also contains a version of Taylor polynomials defined with axiomatic real number coefficients. These polynomials are meant to be used only in the formal verification when linking the computable Taylor models to the corresponding functions on axiomatic real numbers. This link is given by Definition 1. The definition can be easily formalized in the form of a predicate `validTM`.

```
Definition validTM X X0 M f :=
  I.subset X0 X /\
  contains (error M) 0 /\
  let N := tsize (approx M) in
  forall x0, contains X0 x0 -> exists P, tsize P = N /\
    ( forall k, (k < N) ->
      contains (tnth (approx M) k) (tnth P k) ) /\
    forall x, contains X x ->
      contains (error M) (f x - teval P (x - x0)).
```

The theorem of correctness for the Taylor model of the exponential `TM_exp` then establishes the link between the model and the exponential function `Rexp` that is defined in the real library.

```
Lemma TM_exp_correct :
  forall X X0 n, validTM X X0 (TM_exp n X X0) Rexp.
```

Our goal is to formally prove the correctness of our implementation of Taylor models. We want proofs that are generic, so a new instantiation of the polynomials would not require changing the proofs. In a previous version of our Coq development we had managed to prove correct Taylor models for some elementary functions and addition. No proofs are available yet for the version presented here but adapting the proofs to this new setting should be possible.

4 Benchmarks

We want to evaluate the performances of our Coq implementation of Taylor models. For this we compare them to those of Sollya [13], a tool specially designed to handle such numerical approximation problems.

The Coq Taylor models we use for our tests are implemented with polynomials represented as simple lists with a linear access to their coefficients. The coefficients of the approximating polynomial in our instantiation of Taylor models as well as the interval errors are implemented by intervals with multiple-precision FP bounds as available in the Coq.Interval library described in [30]. Since we need to evaluate the initial conditions for recurrences, only the basic functions already implemented in Coq.Interval can have their corresponding Taylor models.

In Sollya, polynomials have interval coefficients and are represented by a (coefficient) array of intervals with multiple-precision FP bounds. Sollya's `autodiff()` function computes interval enclosures of the successive derivatives of a function at a point or over an interval, relying on interval arithmetic computations and recurrence relations similar to the ones we use in our Coq development. Thus, we use it to compute the Taylor models we are interested in.

Timings, accuracy and comparisons

We compare the Coq and the Sollya implementations presented above on a selection of several benchmarks. Table 1 gives the timings as well as the tightness obtained for the remainders. These benchmarks have been computed on a 4-core computer, Intel(R) Xeon(R) CPU X5482 @ 3.20GHz.

Each cell of the first column of Table 1 contains a target function, the precision in bits used for the computations, the order of the TM, and the interval under consideration. When "split" is mentioned, the interval has been subdivided into a specified amount of intervals of equal length (1024 subintervals for instance in line 3) and a TM has been computed over each subinterval. Each TM is expanded at the middle of the interval. The symbols $\mathrm{RD}_t(\ln 4)$, resp. $\mathrm{RU}_t(\ln 2)$, denote $\ln(4)$ rounded toward $-\infty$, resp. $\ln(2)$ rounded toward $+\infty$, using precision t.

Columns 2 and 3 give the total duration of the computations (for instance, the total time for computing the 1024 TMs of the third line) in Coq and Sollya respectively. Columns 4 and 5 present an approximation error obtained using Coq and Sollya, while the last column gives, as a reference, the true approximation error, computed by ad-hoc means (symbolically for instance), of the function by its Taylor polynomial. Note that when "split" is mentioned, the error presented corresponds to the one computed over the last subinterval (for instance, $[2 - 1/256, 2]$ for the arctan example). For simplicity, the errors are given using three significant digits.

In terms of accuracy, the Coq and Sollya results are close. We have done other similar checks and obtained the same encouraging results (the error bounds returned by Coq and Sollya have the same orders of magnitude). This does not prove anything but is nevertheless very reassuring. Proving the correctness of an implementation that produces too large bounds would be meaningless.

Coq is 6 to 10 times slower than Sollya, which is reasonable. This factor gets larger when composition is used. One possible explanation is that composition implies lots of polynomial manipulations and the implementation of polynomials as simple lists in Coq maybe too naive. An interesting alternative

Table 1. Benchmarks and timings for our implementation in Coq.

	Execution time		Approximation error		
	Coq	Sollya	Coq	Sollya	Mathematical
exp prec=120, deg=20 I=$[1, \mathrm{RD}_{53}(\ln 4)]$ no split	7.40s	0.01s	7.90×10^{-35}	7.90×10^{-35}	6.57×10^{-35}
exp prec=120, deg=8 I=$[1, \mathrm{RD}_{53}(\ln 4)]$ split in 1024	20.41s	3.77s	3.34×10^{-39}	3.34×10^{-39}	3.34×10^{-39}
exp prec=600, deg=40 I=$[\mathrm{RU}_{113}(\ln 2), 1]$ split in 256	38.10s	16.39s	6.23×10^{-182}	6.22×10^{-182}	6.22×10^{-182}
arctan prec=120, deg=8 I=$[1, 2]$ split in 256	11.45s	1.03s	7.43×10^{-29}	2.93×10^{-29}	2.85×10^{-29}
exp $\times$ sin prec=200, deg=10 I=$[1/2, 1]$ split in 2048	1m22s	12.05s	6.92×10^{-50}	6.10×10^{-50}	5.89×10^{-50}
exp/sin prec=200, deg=10 I=$[1/2, 1]$ split in 2048	3m41s	13.29s	4.01×10^{-43}	9.33×10^{-44}	8.97×10^{-44}
exp $\circ$ sin prec=200, deg=10 I=$[1/2, 1]$ split in 2048	3m24s	12.19s	4.90×10^{-47}	4.92×10^{-47}	4.90×10^{-47}

could be to use persistent arrays [2] to have more efficient polynomials. Another possible improvement is at algorithmic level: while faster algorithms for polynomial multiplication exist [17], currently in all TMs related works $O(n^2)$ naive multiplication is used. We could improve that by using a Karatsuba-based approach, for instance.

5 Conclusion and Future Works

We have described an implementation of Taylor models in the Coq proof assistant. Two main issues have been addressed. The first one is genericity. We wanted our implementation to be applicable to a large class of problems. This motivates our use of modules in order to get this flexibility. The second issue is efficiency. Working in a formal setting has some impact in terms of efficiency. Before starting to prove anything, it was then crucial to evaluate if the compu-

tational power provided by Coq was sufficient for our needs. The results given in Section 4 clearly indicate that what we have is worth proving formally.

We are in the process of proving the correctness of our implementation. Our main goal is to prove the validity theorem given in Section 2 formally. This is tedious work but we believe it should be completed in a couple of months. As we aim at a complete formalization, a more subtle issue concerns the Taylor models for the basic functions and in particular how the model and its corresponding function can be formally related. This can be done in an ad-hoc way, deriving the recurrence relation from the formal definition. An interesting future work would be to investigate a more generic approach, trying to mimic what is provided by the Dynamic Dictionary of Mathematical Functions [3] in a formal setting.

Having Taylor models is an initial step in our overall goal of getting formally proved worst-case accuracy for common functions and formats. A natural next step is to couple our models with some positivity test for polynomials, for example some sums-of-squares technique. This would give us an automatic way of verifying polynomial approximations formally. It would also provide another way of evaluating the quality of our Taylor approximations. If they turned out to be not accurate enough for our needs, we could always switch to better kinds of approximations such as Chebyshev truncated series, thanks to our generic setting.

References

1. Abramowitz, M., Stegun, I.A.: Handbook of mathematical functions with formulas, graphs, and mathematical tables, National Bureau of Standards Applied Mathematics Series, vol. 55. For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. (1964)
2. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with Imperative Features and Its Application to SAT Verification. In: ITP. LNCS, vol. 6172, pp. 83–98 (2010)
3. Benoit, A., Chyzak, F., Darrasse, A., Gerhold, S., Mezzarobba, M., Salvy, B.: The Dynamic Dictionary of Mathematical Functions (DDMF). In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) Mathematical Software - ICMS 2010, Lecture Notes in Computer Science, vol. 6327, pp. 35–41. Springer (2010)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)
5. Berz, M., Makino, K.: Rigorous global search using Taylor models. In: SNC '09: Proceedings of the 2009 conference on Symbolic numeric computation. pp. 11–20. ACM, New York, NY, USA (2009)
6. Berz, M., Makino, K., Kim, Y.K.: Long-term stability of the tevatron by verified global optimization. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 558(1), 1 – 10 (2006), proceedings of the 8th International Computational Accelerator Physics Conference - ICAP 2004
7. Boespflug, M., Dénès, M., Grégoire, B.: Full reduction at full throttle. In: CPP. LNCS, vol. 7086, pp. 362–377. Springer (2011)

8. Boldo, S., Melquiond, G.: Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In: Proceedings of the 20th IEEE Symposium on Computer Arithmetic. pp. 243–252. Tübingen, Germany (Jul 2011)
9. Brisebarre, N., Chevillard, S.: Efficient polynomial L^∞-approximations. In: Kornerup, P., Muller, J.M. (eds.) 18th IEEE SYMPOSIUM on Computer Arithmetic. pp. 169–176. IEEE Computer Society, Los Alamitos, CA (2007)
10. Brisebarre, N., Muller, J.M., Tisserand, A.: Computing Machine-efficient Polynomial Approximations. ACM Trans. Math. Software 32(2), 236–256 (2006)
11. Cháves, F.: Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves. These, Ecole normale supérieure de Lyon - ENS LYON (Sep 2007), `http://tel.archives-ouvertes.fr/tel-00177109/en/`
12. Chevillard, S.: Évaluation efficace de fonctions numériques. Outils et exemples. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France (2009), `http://tel.archives-ouvertes.fr/tel-00460776/fr/`
13. Chevillard, S., Joldeş, M., Lauter, C.: Sollya: An environment for the development of numerical codes. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) Mathematical Software - ICMS 2010. Lecture Notes in Computer Science, vol. 6327, pp. 28–31. Springer, Heidelberg, Germany (September 2010)
14. Chevillard, S., Harrison, J., Joldeş, M., Lauter, C.: Efficient and accurate computation of upper bounds of approximation errors. Theoretical Computer Science 16(412), 1523–1543 (2011)
15. Collins, P., Niqui, M., Revol, N.: A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq. In: NSV-3: Third International Workshop on Numerical Software Verification (2010)
16. de Dinechin, F., Lauter, C., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing. pp. 1318–1322. Dijon, France (2006), `http://www.lri.fr/~melquion/doc/06-mcms-article.pdf`
17. von zur Gathen, J., Gerhard, J.: Modern computer algebra. Cambridge University Press, New York, 2nd edn. (2003)
18. Geuvers, H., Niqui, M.: Constructive Reals in Coq: Axioms and Categoricity. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Proceedings of TYPES 2000. LNCS, vol. 2277, pp. 79–95. Springer, Durham, United Kingdom (2002)
19. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA (2008)
20. Grégoire, B., Théry, L.: A purely functional library for modular arithmetic and its application to certifying large prime numbers. In: IJCAR. LNCS, vol. 4130, pp. 423–437 (2006)
21. Griewank, A.: Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation. SIAM (2000)
22. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754™-2008 (Aug 2008)
23. Joldeş, M.: Rigourous Polynomial Approximations and Applications. Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, France (2011), `http://perso.ens-lyon.fr/mioara.joldes/these/theseJoldes.pdf`
24. Krebbers, R., Spitters, B.: Computer Certified Efficient Exact Reals in Coq. In: Calculemus/MKM. pp. 90–106. Bertinoro, Italy (2011)
25. Lefèvre, V., Muller, J.M.: Worst cases for correct rounding of the elementary functions in double precision. In: Burgess, N., Ciminiera, L. (eds.) Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16). Vail, CO (Jun 2001)

26. Lizia, P.D.: Robust Space Trajectory and Space System Design using Differential Algebra. Ph.D. thesis, Politecnico di Milano, Milano, Italy (2008)
27. Makino, K.: Rigorous Analysis of Nonlinear Motion in Particle Accelerators. Ph.D. thesis, Michigan State University, East Lansing, Michigan, USA (1998)
28. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. International Journal of Pure and Applied Mathematics 4(4), 379–456 (2003), http://bt.pa.msu.edu/pub/papers/TMIJPAM03/TMIJPAM03.pdf
29. Mayero, M.: Formalisation et automatisation de preuves en analyses réelle et numérique. Ph.D. thesis, Université Paris VI (2001)
30. Melquiond, G.: Proving Bounds on Real-Valued Functions with Computations. In: Proceedings of the 4th International Joint Conference on Automated Reasoning. pp. 2–17. Sydney, Australia (Aug 2008)
31. Moore, R.E.: Methods and Applications of Interval Analysis. Society for Industrial and Applied Mathematics (1979)
32. Muller, J.M.: Projet ANR TaMaDi – Dilemme du fabricant de tables – Table Maker's Dilemma (ref. ANR 2010 BLAN 0203 01), http://tamadiwiki.ens-lyon.fr/tamadiwiki/
33. Muller, J.M.: Elementary Functions, Algorithms and Implementation. Birkhäuser Boston, MA, 2nd edn. (2006)
34. Neher, M., Jackson, K.R., Nedialkov, N.S.: On Taylor model based integration of ODEs. SIAM J. Numer. Anal. 45, 236–262 (2007)
35. Neumaier, A.: Taylor forms – use and limits. Reliable Computing 9(1), 43–79 (2003)
36. O'Connor, R.: Certified exact transcendental real number computation in coq. In: Theorem Proving in Higher Order Logics. pp. 246–261 (2008)
37. Project, T.A.: CRlibm, Correctly Rounded mathematical library (July 2006), http://lipforge.ens-lyon.fr/www/crlibm/
38. Remez, E.: Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation (in French). C.R. Académie des Sciences, Paris 198, 2063–2065 (1934)
39. Salvy, B., Zimmermann, P.: Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. ACM Trans. Math. Software 20(2), 163–177 (1994)
40. Stanley, R.P.: Differentiably finite power series. European Journal of Combinatorics 1(2), 175–188 (1980)
41. Ziv, A.: Fast evaluation of elementary mathematical functions with correctly rounded last bit. ACM Trans. Math. Software 17(3), 410–423 (Sep 1991)
42. Zumkeller, R.: Formal Global Optimization with Taylor Models. In: Proc. of the 4th International Joint Conference on Automated Reasoning. pp. 408–422 (2008)