Certified exact real arithmetic using co-induction in arbitrary integer base

Nicolas Julien

Certified exact real arithmetic using co-induction in arbitrary integer base

Nicolas Julien

INRIA Sophia Antipolis

Abstract. In this paper we describe some certified algorithms for exact real arithmetic based on co-recursion. Our work is based on previous experiences using redundant digits of base 2 but generalizes them using arbitrary integer bases. The goal is to take benefit of fast native integer computation. We extend a technique to compute converging series. We use this technique to compute the product and the inverse. We describe how we implement and certify our algorithms in the proof system Coq and evaluate the efficiency of the library inside the prover.

1 Introduction

We built a library to describe computations on real numbers in a certified way. This library can be used inside a theorem prover and it relies on a particular form of recursive programming known as co-recursion. The data manipulated in this library are streams of signed digits, in other words infinite sequences. The central concept is the computation of series, which was already studied in [2]. We suggest a few improvements on the known results and we implement division, a function that had not been considered yet in this particular framework. One of the original characteristics of our work is that our library is parametrized by the base used to interpret the digit streams.

First we will see why we represent real numbers as streams of signed digits of an arbitrary positive integer base. Then we will describe how the formalization of the base influences the complexity of the operations and what are the solutions we provide to adapt the algorithms to this new framework. We will also see how we improve the technique to compute converging series. Finally after giving an idea of the formalization in Coq [3,5], we will illustrate the benefits in efficiency of using large bases with some benchmarks.

2 Representation of real numbers

It is well known that datatypes containing only finite objects are not suitable for representing real numbers. Real numbers are commonly described in computer programs as floating point numbers. This representation actually describes a finite subset of rational numbers. When accumulating computations on such approximations one has to handle round-off problems in order to avoid erroneous results [9].

A common approach used in every day life is to view real numbers as fractional numbers with a possibly infinite fractional part. More formally these representations are sequences of digits i.e. function from positive integers to integers so that the sequence $0.x_1 \dots x_n \dots$ actually has the value $\sum_{i=1}^{\infty} \frac{x_i}{\beta^i}$. In such a representation, every day practice relies on the x_i being between 0 and $\beta - 1$, but a more efficient approach is to take x_i between $-\beta + 1$ and $\beta - 1$. This extension with signed digits adds redundancy but this redundancy supports more efficient algorithms and is some time essential to ensure that some computations terminate. Some other models of exact real arithmetic have been implemented using infinite datatypes. For instance with continued fractions [14], regular functions of rationals [12] or streams of linear fractional transformations [6].

Describing the sequences of digits as simple functions from positive integers to integers seems appropriate to work in a higher order setting but this technique has flaws when you compute the function for a given n and then you want to compute for a higher integer. The second time you need to recompute all that was already computed for n.

With co-inductive types we can represent a real number in a more natural way i.e. as the infinite stream of its digits. Co-inductive objects are lazily evaluated, thus the first n digits of a stream will be reused when computing the following digits and then avoid re-computation. These objects are defined by co-recursive functions and properties over these functions can be proved by co-induction.

Since co-induction [7] in Coq provides such a framework to reason on infinite datatypes, several certified implementations of exact real arithmetic arose using streams of a set of three redundant digits [2,4] or streams of linear fractional transformations [11].

We chose to represent real numbers as streams of signed digits of an arbitrary integer base. The set of signed digits of a base β is $\{-\beta+1, \dots, \beta-1\}$. The negative digits will be written with a bar : $\bar{1} = -1$. This representation is redundant in the sense that a number will have several representations. For instance the streams $3::3::3::3\dots$ and $4::\bar{7}::3::3::3\dots$ are suitable representations of the number $\frac{1}{3} = 0.33333\dots$ in base ten.

Often, we will not make any difference between a stream and the real value it represents. The stream beginning by the digit k and followed by the stream s is denoted by $k::s$ and its value is $\frac{k+s}{\beta}$, since $\sum_{i=1}^{\infty} \frac{d_i}{\beta^i} = \frac{d_1 + \sum_{i=1}^{\infty} \frac{d_{i+1}}{\beta^i}}{\beta}$.

The interval of numbers that can be represented by a stream beginning with a digit k is $[\frac{k-1}{\beta}, \frac{k+1}{\beta}]$. Indeed the value of a stream $k::s$ is $\frac{k+s}{\beta}$ and a stream s represents a value of [-1, 1]. The redundancy comes from the fact that two consecutive intervals of this kind overlap : $[\frac{k-1}{\beta}, \frac{k+1}{\beta}] \cap [\frac{(k+1)-1}{\beta}, \frac{(k+1)+1}{\beta}] = [\frac{k}{\beta}, \frac{k+1}{\beta}]$. The benefit of the redundancy can be understood noticing that the magnitude of overlaps of intervals of consecutive digits is a constant equal to $\frac{1}{\beta}$. Thus knowing an interval of this magnitude containing a real is always enough to decide the first digit for one of its representations.

Since we have a way to describe all the real numbers in [-1, 1] we can use a couple (mantissa, exponent) to represent all the real numbers. So when describing an algorithm, we first describe the part on the mantissa and then we extend it to

the full representation. Since the second part is standard however the mantissa is represented, we will focus on the first part.

To compute the mantissa of a real number we try to obtain an interval which contains the number such that we can decide the first digit of the number. Thus the operations on mantissa are described by co-recursion as functions that produce the first digit and are called recursively to produce the following digits.

3 Computing addition

The problem with addition is that the result of adding two numbers of [-1, 1] is not in [-1, 1] but in $[-2, 2]$ and so it could not be represented only with a stream. Bertot proposes to avoid this by first defining the "half-sum" $x, y \mapsto \frac{x+y}{2}$. Then he defines the function that multiplies a number by 2 if it's in $[0, \frac{1}{2}]$, $x \mapsto \begin{cases} 2x & \texttt{if} \quad x \leq \frac{1}{2} \\ 1 & \texttt{otherwise} \end{cases}$. Finally he obtains some kind of addition by composing the two functions.

As we work with an arbitrary integer base we have adapted this idea replacing 2 by the base in the division and the multiplication. Thus we first describe a function that given a base, two streams and a integer remainder computes a stream of the sum of the streams and the remainder, all divided by the base :

$$\text{sum\_div\_base}_\beta \begin{cases} \text{[-1, 1]}^2 \times [-\beta + 2, \beta - 2] \mapsto \text{[-1, 1]} \\ x, y, r \mapsto \frac{x+y+r}{\beta} \end{cases}$$

This operation is stable in [-1, 1] and we provide an algorithm to compute it :

- We read the first digit of x and y. We now have k_1, k_2, x' and y' such that $x = k_1 :: x', y = k_2 :: y'$. Thus we have :

$$\texttt{sum\_div\_base}(\beta, x, y, r) = \frac{r + \frac{x'+y'+k_1+k_2}{\beta}}{\beta}.$$

  Since k_1 and k_2 are signed digits of the base, $-2\beta + 2 \leq k_1 + k_2 \leq 2\beta - 2$. Thus r seems to be a good candidate for a first digit of the result and $k_1 + k_2$ for the next remainder.
- If $\beta - 1 \leq k_1 + k_2$ then $k_1 + k_2$ is too big to be a suitable remainder, but $k_1 + k_2 - \beta$ is acceptable. Hence the first digit can be $r + 1$ which is in the set of signed digits of the base because r is the previous remainder $-\beta + 2 \leq r \leq \beta - 2$.

$$\begin{aligned} \texttt{sum\_div\_base}(\beta, x, y, r) &= \frac{r + 1 + \frac{x'+y'+k_1+k_2-\beta}{\beta}}{\beta} \\ &= (r+1) :: \texttt{sum\_div\_base}(\beta, x', y', k_1 + k_2 - \beta). \end{aligned}$$

– If $k_1 + k_2 \leq -\beta + 1$ then $k_1 + k_2$ is too small to be a suitable remainder, but $k_1 + k_2 + \beta$ is acceptable. Hence the first digit can be $r - 1$ which is in the set of signed digits of the base because r is the previous remainder $-\beta + 2 \leq r \leq \beta - 2$.

$$\begin{aligned}\texttt{sum\_div\_base}(\beta, x, y, r) &= \frac{r - 1 + \frac{x' + y' + k_1 + k_2 + \beta}{\beta}}{\beta} \\ &= (r-1)::\texttt{sum\_div\_base}(\beta, x', y', k_1 + k_2 + \beta).\end{aligned}$$

– Otherwise $k_1 + k_2$ is a suitable value for the remainder and r can be the first digit.

$$\begin{aligned}\texttt{sum\_div\_base}(\beta, x, y, r) &= \frac{r + \frac{x' + y' + k_1 + k_2}{\beta}}{\beta} \\ &= r::\texttt{sum\_div\_base}(\beta, x', y', k_1 + k_2).\end{aligned}$$

We now have to define the function that multiplies a stream by the base when the result can be represented by a stream or gives a dummy result otherwise.

$$\texttt{mult\_base} : \begin{cases} \mathbb{Z} \times [\text{-}1,\ 1] \mapsto [\text{-}1,\ 1] \\ \\ \beta, x \qquad\quad \mapsto \begin{cases} -1 & \text{if } x \leq \frac{-1}{\beta} \\ 1 & \text{if } x \geq \frac{1}{\beta} \\ x \times \beta & \text{otherwise} \end{cases} \end{cases}.$$

We first have to notice that the real numbers -1 and 1 obviously have only one representation since they are the edges of the representable values. The only way to represent 1 (resp. -1) is the sequence where only the maximal (resp. minimal) digit $\beta - 1$ (resp. $-\beta + 1$) occurs. This is justified by the following equality.

$$\sum_{i=1}^{\infty} \frac{\beta - 1}{\beta^i} = \frac{\beta - 1}{\beta} \sum_{i=0}^{\infty} \frac{1}{\beta^i} = \frac{\beta - 1}{\beta} \frac{1}{1 - \frac{1}{\beta}} = 1$$

Now we can describe the algorithm of the function `mult_base` :

– We read the first digit of x : $x = k_1::x'$
– If $k_1 = 0$ then the result is x' : $\beta\frac{0 + x'}{\beta} = x'$
– If $k_1 \leq -2$, then we can deduce that $x \leq \frac{-1}{\beta}$ hence the result must be the constant -1.
– If $k_1 \geq 2$, then we can deduce that $x \geq \frac{1}{\beta}$ hence the result must be the constant 1.
– If $k_1 = 1$ we need to look at one more digit of x : $x = 1::k_2::x''$
  • If $k_2 < 0$, then we use the redundancy of the representation to reduce the problem to a previous case $x = 1::k_2::x'' = 0::(k_2 + \beta)::x''$, hence the result is $(k_2 + \beta)::x''$.
  • If $k_2 > 0$, we also come to a previous case $x = 1::k_2::x'' = 2::(k_2 - \beta)::x''$, hence the result is 1.

- If $k_2 = 0$ then we need a recursive call :

$$\begin{aligned}\texttt{mult\_base}(\beta, 1::0::x'') &= \beta \times \frac{1 + \frac{0+x''}{\beta}}{\beta} \\ &= \frac{\beta - 1 + \beta \times \frac{1+x''}{\beta}}{\beta} \\ &= \beta - 1::\texttt{mult\_base}(\beta, 1::x'').\end{aligned}$$

  In this case we can't know if our parameter x is lesser or equal to $\frac{1}{\beta}$. And maybe it won't be possible even if we read an arbitrary large finite number of more digits of x. Nevertheless, if it was lesser or equal, we would know that it would be close enough to $\frac{1}{\beta}$ to be sure that the result could begin with the digit $\beta - 1$. And if it was greater, then the result should be the constant 1 that begins with the same digit $\beta - 1$. So it's correct to produce the digit and to let the recursive call try to decide later or construct the constant 1 step by step.

– If $k_1 = -1$ we have a symmetrical reasoning as the previous case $k_1 = 1$.

By composing our two functions, we finally get a function over two mantissas which computes their sum when this is in [-1, 1].

$$\texttt{add}(\beta, x, y) = \texttt{mult\_base}(\beta, \texttt{sum\_div\_base}(\beta, x, y, 0))$$

$$\texttt{add} : \begin{cases} \mathbb{Z} \times [\text{-}1,\ 1] \times [\text{-}1,\ 1] \mapsto [\text{-}1,\ 1] \\ \\ \beta, x, y \qquad\qquad\qquad \mapsto \begin{cases} -1 & \text{if } x + y \leq -1 \\ 1 & \text{if } x + y \geq 1 \\ x + y \text{ otherwise} \end{cases} \end{cases}$$

4 The function `make_digit`

We have described an algorithm for addition which computes the first digit of their sum from the first two digits of the parameters and so on recursively. Another method is possible when we know that one argument is close enough to zero. In this case computing a prefix of the other parameter is enough to compute the first digit of their sum.

Indeed, suppose that we want to add x and y and we know $|y| \leq \frac{\beta-2}{2\beta^2}$. If we compute the first two digits of $x = d_1::d_2::x''$ then we know a frame of x of magnitude $\frac{2}{\beta^2}$. Thus we know a frame of their sum of magnitude $\frac{1}{\beta}$ which is enough to compute the first digit of the sum.

We propose here to define a function `make_digit` which from the stream x representing a real number, gives another stream $k::x'$ representing the same number but such that k could be the possible first when adding a number close enough to zero.

Let y be a number "close enough of 0" :

$$x + \frac{-\beta + 2}{2\beta^2} \leq \texttt{make\_digit}(\beta, x) + y \leq x + \frac{\beta - 2}{2\beta^2}.$$

We describe the algorithm of `make_digit` as follow :

- We first look at the first two digits of $x = k_1 :: k_2 :: x''$. So we have

$$\frac{k_1 + \frac{k_2 + x''}{\beta}}{\beta} + \frac{-\beta + 2}{2\beta^2} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + \frac{k_2 + x''}{\beta}}{\beta} + \frac{\beta - 2}{2\beta^2}$$

$$\frac{k_1 + \frac{2k_2 + 2x'' - \beta + 2}{2\beta}}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + \frac{2k_2 + 2x'' + \beta - 2}{2\beta}}{\beta}$$

  Since $x'' \in$ [-1, 1] we also have

$$\frac{k_1 + \frac{2k_2 - \beta}{2\beta}}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + \frac{2k_2 + \beta}{2\beta}}{\beta}$$

- If $-\beta \leq 2k_2 \leq \beta$, then we have

$$\frac{k_1 - 1}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + 1}{\beta}$$

  Thus the result can start with the digit k_1 and the remaining stream is $k_2 :: x''$:

$$\texttt{make\_digit}(\beta, x) = k_1 :: k_2 :: x'' = x.$$

- Otherwise, if $\beta < 2k_2$
  - If $k_1 \neq \beta - 1$, then we use the redundancy of the representation $k_1 :: k_2 :: x'' = k_1 + 1 :: k_2 - \beta :: x''$, we can then show that

$$\frac{(k_1 + 1) - 1}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{(k_1 + 1) + 1}{\beta}$$

    The result can be :

$$\texttt{make\_digit}(\beta, x) = k_1 + 1 :: k_2 - \beta :: x''.$$

  - Otherwise $k_1 + 1 = \beta$, it is not in the set of digits, but since we suppose the result is in [-1, 1], we have

$$\frac{(\beta - 1) - 1}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq 1 = \frac{(\beta - 1) + 1}{\beta}$$

    The result can be :

$$\texttt{make\_digit}(\beta, x) + y = \beta - 1 :: k_2 :: x''.$$

- Otherwise $2k_2 < -\beta$ and we have a similar reasoning.

5 Computing series

In his work Bertot describes how to compute converging series using the technique described in the function `make_digit`. Indeed a converging series can be split into a finite part and an infinite part as close to zero as needed. We have adapted this technique for an arbitrary integer base and improved its description by defining the function `make_digit`.

To compute the stream of a series $\sum_{i=0}^{\infty} a_i$ which converges in [-1, 1], we start with defining a more general function

$$f(\beta, j, n, r) = \beta^j \times \sum_{i=n}^{\infty} a_i + r.$$

Then we should compute the series as a particular case : $f(\beta, 0, 0, 0) = \sum_{i=0}^{\infty} a_i$.

We first have to find a $p \geq n$ such that $|\beta^j \times \sum_{i=p}^{\infty} a_i| \leq \frac{\beta-2}{2\beta^2}$. Because the series is converging we know that such a p exists.

$$f(\beta, j, n, r) = (\beta^j \times \sum_{i=n}^{p-1} a_i + r) + \beta^j \times \sum_{i=p}^{\infty} a_i$$

We are in the situation that `make_digit` was designed for. We can compute a digit k and a stream r' such that $k::r' = \beta^j \times \sum_{i=n}^{p-1} a_i + r$ with k a possible first digit of the result of $f(\beta, j, n, r)$. Then we can produce this k as the first digit and continue computing the series with a recursive call :

$$\begin{aligned} f(\beta, j, n, r) &= (\beta^j \times \sum_{i=n}^{p-1} a_i + r) + \beta^j \times \sum_{i=p}^{\infty} a_i \\ &= \frac{k + r'}{\beta} + \beta^j \times \sum_{i=p}^{\infty} a_i \\ &= \frac{k + \beta \times \beta^j \sum_{i=p}^{\infty} a_i + r'}{\beta} \\ &= k::f(\beta, j+1, p, r') \end{aligned}$$

Thus the schema to define a co-recursive function that computes a series is :

- To find the p that split the series such that the infinite part is close enough to 0. This is often done with a recursive function.
- To compute a possible first digit using `make_digit` and produce it.
- To define the stream of the following digits with a co-recursive call.

The part of the function that uses `make_digit` to produce the first digit and perform the recursive call does not depend on the series and can be formalized once and for all. The parameter r of f could be understood as the difference between the series and its approximation given by its first digit. In general, the

parameters of the function f will not exactly be the ones we used to describe the technique. Sometimes a parameter is simplified or hidden in another parameter. And often we will use extra parameters to describe intermediate computations. For instance to avoid re-computing a $n!$ that appears in a series.

The number $\beta^j \times \sum_{i=n}^{p-1} a_i + r$ we give to `make_digit` is not necessarily in [-1, 1]. But the result of adding the stream representing $\beta^j \times \sum_{i=n}^{p-1} a_i$ and the stream r is guaranteed only if it is in [-1, 1]. To ensure the correctness of our computations we previously relied on tricks, like grouping terms of the series to have only positive terms.

We recently found a better solution. It consists in guaranteeing that the parameter r is always inside the interval $[-\frac{\beta+2}{2\beta}, \frac{\beta+2}{2\beta}]$. This parameter is either the initial value 0 or computed by `make_digit` at a previous step. The function `make_digit` is trying to produce r as close to 0 as possible, thanks to redundancy. We can see in its algorithm that r is not in this interval only if the input is too close to the bounds of [-1, 1] where redundancy cannot be used i.e. outside $[-\frac{2\beta^2-\beta-2}{2\beta^2}, \frac{2\beta^2-\beta-2}{2\beta^2}]$. But when computing series, the input of `make_digit` is the sum of a previous r and the finite part of the series : $\beta^j \times \sum_{i=n}^{p-1} a_i$. Our solution is to consider series converging inside $[-\frac{\beta-4}{2\beta}, \frac{\beta-4}{2\beta}]$. Then splitting the series in order to ensure its infinite part is in $[-\frac{\beta-2}{2\beta^2}, \frac{\beta-2}{2\beta^2}]$ forces its finite part in $[-\frac{\beta^2-3\beta-2}{2\beta^2}, \frac{\beta^2-3\beta-2}{2\beta^2}]$. Thus the sum of the finite part of the series and a number in $[-\frac{\beta+2}{2\beta}, \frac{\beta+2}{2\beta}]$ is a suitable input for `make_digit` to produce a stream that is still in $[-\frac{\beta+2}{2\beta}, \frac{\beta+2}{2\beta}]$ an so on.

A simple way to compute series converging in [-1, 1] but outside $[-\frac{\beta-4}{2\beta}, \frac{\beta-4}{2\beta}]$ is to divide its terms by the base β by adding a zero in front of the stream of each finite part of the series. Thus the series will converge in the right interval if $\beta \geq 6$ and finally we multiply it by the base to compute the initial series.

6 Computing multiplication

Multiplication can be defined as a series and we can use the previous technique to define it.

$$u \times v = \sum_{i=1}^{\infty} \frac{u_i}{\beta^i} \times v$$

Bertot shows that in the multiplication, the parameter y that appears in the technique we described, is useless. Indeed at each step it is multiplied by β but then it is used in a computation where it is divided by β. The parameter n is also useless because $\sum_{i=n}^{\infty} \frac{u_i}{\beta^i}$ is the original stream u without its first n digits. The general function we have to defined is $f(\beta, u, v, r) = u \times v + r$. To do this, we first have to read the first digit of $u = k::u'$, then :

$$f(\beta, u, v, r) = \frac{k + u'}{\beta} \times v + r = \frac{k \times v}{\beta} + r + \frac{u' \times v}{\beta}$$

Then we use `make_digit` to find the first digit of the result.
If $d::r' = \texttt{make\_digit}(\frac{k\times v}{\beta} + r)$ and $|\frac{u'\times v}{\beta}| \leq \frac{\beta-2}{2\beta^2}$ then

$$f(\beta, u, v, r) = d::f(\beta, u', v, r')$$

As we said before, we can see here the division of $u' \times v$ by the base which is canceled by the multiplication that appears in the recursive call. We need that the inequality $|\frac{u'\times v}{\beta}| \leq \frac{\beta-2}{2\beta^2}$ holds to make this definition valid. A simple way to ensure it is to divide the parameter v by the base, adding the digit 0 in the head of the stream, and multiply the result by the base :

$$\beta \times f(\beta, u, 0::v, r) = \beta \times u \times \frac{v}{\beta} + r = u \times v + r$$

Then it becomes trivial that for all streams u' and v representing numbers in [-1, 1], $|\frac{u'\times \frac{v}{\beta}}{\beta}| \leq \frac{\beta-2}{2\beta^2}$ holds.

Another problem we have to deal with is to ensure that the parameter we give to `make_digit` can be computed i.e. $\frac{k\times v}{\beta} + r \in$ [-1, 1]. Our first approach was to modify the algorithm, reading one more digit of u. In this way, thanks to redundancy, we could easily modify the argument given to `make_digit` in order to be in [-1, 1]. But seven different cases had to be considered so it became much more difficult to understand the algorithm and especially proof. When we improved our specification of `make_digit` we decided to use it to simplify our multiplication. We just have to ensure that $|\frac{k\times v}{\beta}| \leq \frac{\beta-3\beta-2}{2\beta}$. Here adding a zero in front of v is almost enough. Adding a second one makes the needed inequality holds.

So we now compute the multiplication in this way :

$$\beta \times \beta \times f(\beta, u, 0::0::v, 0) = \beta \times \beta \times u \times \frac{\frac{v}{\beta}}{\beta} + 0 = u \times v$$

The last requirement we need is a way to compute the multiplication of a stream by a digit as $\frac{k\times v}{\beta}$ occurs in the parameter of `make_digit`.

First we defined it as a series similarly as the multiplication of streams. Among the requirements for this series, we need to compute the representation of rational numbers. This easy operation was already done in the first basic functions we defined.

Unfortunately this computation of multiplication was less efficient than the definition of Bertot for the base two. Indeed when using arbitrary integers base the multiplication of a stream by a digit becomes a problem. In base 2 it was just a multiplication by 0 or by 1 or a division by the base i.e. direct operations but now we compute it as a series.

We found inspiration in Avizienis' work [1] to improve our multiplication of a stream by a digit. The idea is that with the set of digits we use, we have enough redundancy to produce at the same time the digits of the addition of $\lceil\frac{\beta}{2}\rceil$ numbers. This addition can be defined in the same schema we used for the

addition of two streams.

$$Add_{\beta,n} \begin{cases} [\text{-}1,\ 1]^n \times [-\beta+n, \beta-n] \mapsto [\text{-}1,\ 1] \\ x_1, \ldots, x_n, r \mapsto \frac{x_1+\ldots+x_n+r}{\beta} \end{cases}$$

Since r is a remainder of a division by β, the thinnest set of values it needs is $[-\lfloor\frac{\beta}{2}\rfloor, \lfloor\frac{\beta}{2}\rfloor]$. And thus the maximal number of streams we can add is $\lceil\frac{\beta}{2}\rceil$.

Actually, using this idea, we defined a particular case of this addition where all inputs are the same stream. This define the multiplication of a stream by a digit of $[0, \lceil\frac{\beta}{2}\rceil]$. To compute multiplication by a digit larger than $\lceil\frac{\beta}{2}\rceil$, we use this multiplication and an extra addition : $\frac{k\times x}{\beta} = \frac{(k-\beta)\times x}{\beta} + x$. If the digit is negative, then we proceed as described with the positive value and then we compute the opposite.

7 Computing inverse

The inverse function does not fit well with the tradition of theorem provers to only support total functions. Moreover the inverse function cannot be extended easily into a total function because it is undecidable to know whether a given stream represents 0. A second problem is that the inverse of a number in [-1, 1] normally is outside this interval.

A way to avoid this problem is to define a function that is only guaranteed to coincide with $x \mapsto \frac{1}{\beta^n x}$ when the input is outside $[-\frac{1}{\beta^n}, \frac{1}{\beta^n}]$ and may return a value that cannot be trusted otherwise.

The algorithm we propose is the following.

- If $n = 0$ the result should be x itself. Indeed the result we produce has to be correct only if $1 = \beta^0 \leq |x|$. Then x should be equal to -1 or 1 because it is in [-1, 1]. In both cases we have $\frac{1}{x} = x$.
- Otherwise $1 \leq n$ and we have to read the first digit of $x = k_1 :: x'$.
  - If $2 \leq |k_1|$, we can turn the function into a converging series :

$$\frac{1}{\beta^n \times \frac{k_1+x'}{\beta}} = \frac{1}{\beta^{n-1}} \frac{1}{k_1} \frac{1}{1 - \frac{-x'}{k_1}} = \frac{1}{\beta^{n-1}} \frac{1}{k_1} \sum_{i=0}^{\infty} \left(\frac{-x'}{k_1}\right)^i$$

    As $2 \leq |k_1|$, $|\frac{-x}{k_1}| \leq \frac{1}{2}$ so it is straightforward that the series $\sum_{i=0}^{\infty} \left(\frac{-x}{k_1}\right)^i$ converges in $[-2, 2]$. The division by k_1 makes the series converging in [-1, 1]. Therefore a representation of the result could be the stream of this series where $n-1$ zeros were added in front of it to compute the remaining division by β^{n-1}.
  - If $k_1 = 0$ then $\frac{1}{\beta^n \times \frac{0+x'}{\beta}} = \frac{1}{\beta^{n-1} \times x'}$. This yields a recursive call on $n-1$ and x'.
  - If $k_1 = 1$ then we need to read a second digit of $x = k_1 :: k_2 :: x''$.

* If $k_2 < 0$ then $k_2+\beta$ is in the set of signed digits of β and the prefixes $1::k_2$ and $0::(k_2+\beta)$ are equivalent i.e. $\frac{1}{\beta}+\frac{k_1}{\beta^2} = \frac{0}{\beta}+\frac{k_1+\beta}{\beta^2}$. Therefore we can proceed as in a previous case.
* If $k_2 > 0$ then $k_2 - \beta$ is in the set of digits and the prefixes $1::k_2$ and $2::(k_2 - \beta)$ are equivalent. Therefore we can proceed as in a previous case.
* Otherwise $k_2 = 0$ then we cannot choose another representation of x but we can use again a series to compute the inverse :

$$\frac{1}{\beta^n \times (1::0::x'')} = \frac{1}{\beta^n \frac{1+\frac{0+x''}{\beta}}{\beta}} = \frac{1}{\beta^{n-1}} \frac{1}{1-\frac{-x''}{\beta}} = \frac{1}{\beta^{n-1}} \sum_{i=0}^{\infty} \left(\frac{-x''}{\beta}\right)^i$$

Here again we rely on a converging series. But it is converging in $[-\frac{\beta}{\beta-1}, \frac{\beta}{\beta-1}]$ which is larger than [-1, 1].
We can then distinguish two cases.
* If $2 \leq n$ then we can write

$$\frac{1}{\beta^{n-1}} \sum_{i=0}^{\infty} \left(\frac{-x''}{\beta}\right)^i = \frac{1}{\beta^{n-2}} \frac{1}{\beta} \sum_{i=0}^{\infty} \left(\frac{-x''}{\beta}\right)^i$$

Thanks to the division by the base we can make the series converge inside $[-\frac{1}{\beta-1}, \frac{1}{\beta-1}]$ which is inside [-1, 1]. Thus the inverse will be the stream beginning by $n-2$ zeros followed by the stream of the series.
* Otherwise $n = 1$ then we can compute

$$\frac{1}{\beta^0} \sum_{i=0}^{\infty} \left(\frac{-x''}{\beta}\right)^i = \beta \frac{1}{\beta} \sum_{i=0}^{\infty} \left(\frac{-x''}{\beta}\right)^i$$

In this case we can also compute the stream of the series divided by the base. And if the x satisfies $\frac{1}{\beta^n} \leq |x|$ then this stream should be in $[-\frac{1}{\beta}, \frac{1}{\beta}]$. Therefore the result of the multiplication by the base will be guaranteed and we obtain the inverse.

- If $k_1 = -1$ then we can proceed symmetrically as $k_1 = 1$

In three cases this first step requires to describe how to compute a converging series

$$\frac{1}{k} \sum_{i=0}^{\infty} x^i \quad \texttt{with} \quad 2 \leq |k| \leq \beta \quad \texttt{and} \quad |x| \leq \frac{1}{k}.$$

So we first should define the function $f(x, j, n, r) = r + \beta^j \sum_{i=n}^{\infty} x^i$. As we saw, to compute this series we need to find a $p \geq n$ such that $|\beta^j \sum_{i=p}^{\infty} x^i| \leq \frac{\beta-2}{2\beta^2}$. So our first intuition was that finding a suitable p and computing $r + \beta^j \sum_{i=n}^{p-1} x^i$ iteratively should be too expensive. So we tied to find a different way to compute the series that need less computation for each step.

We noticed that for each $l \geq 1$,

$$\begin{aligned}\frac{1}{k}\sum_{i=0}^{\infty} x^i &= \frac{1}{k}(1 + \ldots + x^{l-1} + x^l(1 + \ldots + x^{l-1}) + \ldots) \\ &= \frac{1}{k}\sum_{i=0}^{l-1} x^i \sum_{i=0}^{\infty}(x^l)^i.\end{aligned}$$

In this way we can find a l such that the computation of the series could be simplified. If we choose it even, then the terms of the series $\sum_{i=0}^{\infty}(x^l)^i$ will be all positive and if we choose it larger than $\log_k \beta$ then we will have $|x^l| \leq \frac{1}{\beta}$, since $|x| \leq \frac{1}{k}$.

It will be now easy to define the function that for $0 \leq x \leq \frac{1}{\beta}$ and y in [-1, 1], computes :

$$\begin{aligned}f(\beta, n, x, y, r) &= r + \beta^n \times y \sum_{i=n}^{\infty} x^i \\ &= r + \beta^n \times y \times x^n + \beta^n \times y \sum_{i=n+1}^{\infty} x^i.\end{aligned}$$

As we described in the technique, to split the series in $n+1$ we need :

$$\begin{aligned}|\beta^n \times y \sum_{i=n+1}^{\infty} x^i| &\leq \frac{\beta-2}{2\beta^2} \\ \beta^n \times |y| \times |\frac{x^{n+2}}{1-x}| &\leq \frac{\beta-2}{2\beta^2} \\ \beta^n \frac{x^{n+2}}{1-x} &\leq \frac{\beta-2}{2\beta^2} \\ \frac{x}{\beta(1-x)} &\leq \frac{\beta-2}{2\beta^2} \\ 2x\beta^2 &\leq \beta(1-x)(\beta-2) \\ 2\beta &\leq (\beta-1)(\beta-2) \\ 4 &< \beta\end{aligned}$$

So at each step we can split the series in this way if we work with a base greater than 4. We use extra parameters to keep the precomputed value $\beta^n \times y \times x^n$. In this way, our implementation requires one multiplication and one addition for each step. The initialization requires to compute x^l and $\sum_{i=0}^{l-1} x^i$ such that l is even and $|x^l| \leq \frac{1}{\beta}$. And by composing all this functions, we define the inverse $x, n \mapsto \frac{1}{\beta^n \times x}$ when x is outside $[-\frac{1}{\beta^n}, \frac{1}{\beta^n}]$.

But since we are handling streams that are lazily computed, numbers are not really precomputed and even if only an addition and a multiplication appear in

the recursive call, a lot of computations is actually done. We are now working in a

8 Formalization and proofs of correctness

Co-induction [7] in Coq provides a way to define types of potentially infinite objects. It allows us to implement this representation of real numbers and our algorithms as we described them. The type of infinite sequences of objects of any type `A` could be define as follow.

```
CoInductive stream (A:Set): Set :=
  | Cons : A → stream A → stream A.
```

`Cons` should not be understood as a way to construct an infinite stream from another since we cannot give an initial infinite stream, but as a way to decompose an infinite stream into a finite part and an infinite part that could be described again with `Cons` and so on. A co-inductive object is lazily evaluated each time one asks for a better description. The only way to force a step of evaluation of such an object in Coq is using pattern matching.

We can then define streams using co-recursive functions, for instance the stream of 0 which obviously represents the real number 0.

```
Cofix zero : stream ℤ :=
   Cons 0 one.
```

Coq prevents the users from defining non terminating functions. The way to certify that recursive functions always terminate in Coq is to only provide structural recursion on inductive types. For co-recursive functions it means that one can expect the evaluation of any finite part of the object described by the function to terminate. This is guaranteed if co-recursive calls are done after producing a part of the result i.e. inside a constructor of the co-inductive type.

Co-inductive predicates can be defined to describe an infinite behavior of such an object. We defined in this way a relation between a stream and the number it represents using the fact we explained before : $k::s = \frac{k+s}{\beta}$.

```
CoInductive represents (β : ℤ): stream ℤ →ℝ →Prop :=
  | rep : ∀ s r k, −β < k < β →  −1 ≤ r ≤ 1 →
    represents β s r →represents β (Cons k s) (k+r)/β.
```

It means that if k is in the set of signed digits of base β and s is the representation in this base of a number $r \in$ [-1, 1], then $\frac{k+r}{\beta}$ is the value represented by `Cons` k s in base β.

Then to show that an algorithm we define on our representation is computing a mathematics function, we prove that the predicate `represents` is some kind of a morphism between the algorithm and the function. For instance, the theorem that the stream `zero` represents the real value 0 is

```
Theorem zero_correct :
∀ β, represents β zero 0.
```

Proving a co-inductive predicate in Coq means constructing a co-recursive function whose type is the predicate. The tactic `cofix` helps to construct a proof-term by adding the goal to the hypothesis. But since applying since applying this hypothesis correspond to a co-recursive call, one has to prove a finite part of the theorem before. So when saving the proof, Coq system checks if this guarded condition holds to validate the proof.

In `http://www-sop.inria.fr/marelle/Nicolas.Julien/exactreals.tgz` our development in Coq can be found.

9 Benchmarks

The goal of formalizing the base in this library was to use fast operations on integers. We present here some benches inside the proof system Coq. The digits are implemented with the library `BigZ` [8] which provides fast operations on non-bounded integers using an implementation of native integers [13] in Coq.

We compared the time of computation of the same number in different bases. For each base we computed the number of digit needed for the same precision : n digits in base β give a precision of the number of $\frac{1}{\beta^n}$. For instance knowing 10 digits in base 2^{10} is equivalent to knowing 1 digit in base 2^{100}. These computations were made on a computer with two processors P4 3.40GHz and 1GB of memory.

- Computation of $\frac{3}{7} + \frac{5}{9}$

Used Base	LCR (2)	2^{31}	2^{62}	2^{124}	2^{248}	2^{496}
Number of digit	248000	16000	8000	4000	2000	1000
Time (s)	Computation failure	3.600	1.896	1.036	0.584	0.348

- Computation of $\frac{3}{7} \times \frac{5}{9}$

Used Base	LCR (2)	2^{31}	2^{62}	2^{124}	2^{248}	2^{496}
Number of digit	7440	240	120	60	30	15
Time (s)	77	7.032	2.364	0.852	0.384	0.220

  It is clear here that the use of big bases improves computations. The reason is that since the complexity of operation we use should not be affected when the size of the base is increasing, reducing the number of digit reduces the number of recursive steps.
- Computation of $\frac{\pi}{4}$

Used Base	LCR (2)	2^{31}	2^{62}	2^{124}	2^{248}	2^{496}
Number of digit	3472	112	56	28	14	7
Time (s)	Computation failure	14.6	5.57	3.49	4.03	6.93

  This computation of $\frac{\pi}{4}$ is realized with more than one thousand of decimals : $\frac{1}{2^{3472}} = \frac{1}{4\times(2^{10})^{347}} \leq \frac{1}{10^{1041}}$. We can see here that even if the efficiency is firstly growing with the base, with very large bases it is decreasing. A reason could be that since we need to compute only a few digits with big bases, the efficiency of computation on integers starts to matter. Indeed for the test

on adding or multiplying all computed integers were lower than the square of the base. Here as contrary we used integers for intermediate computation that can be much larger.

– Computation of $\frac{1}{\beta^1 \times \left(\frac{3}{7}+\frac{5}{9}\right)}$

Used Base	2^{31}	2^{62}	2^{124}	2^{248}	2^{496}
Number of digit	16	8	4	2	1
Time (s)	0.056	0.196	0.740	3.176	15.3

Surprisingly we can observe an inverse behavior here. A possible explanation is that in the trick we use, we split the series in part of $2 \times \log_k \beta$. for computing each digit which is much more than needed. And the bigger the base, the larger the difference between what we compute and what we need. We have not yet redesigned our procedure to compute the inverse to take into account the new technique we propose for series. It may be a strong improvement.

10 Conclusion and future work

Our work contributes mainly to the problem of computing converging series. Previous work only gave insights for the computation of series with all positive terms, but we identified the problems that may be encountered when terms may have different signs. We propose a new approach to decompose each step of the computation, embodied in our function `make_digit` and we show that series that are proved to converge in a shorter interval than [-1,1] are easier to handle and we propose a technique to map all cases to the easy ones. Properties and tactics are also described to simplify the certification of implementation of new series.

An important point is that this library is compatible with the reduction mechanism of Coq. It means that all operations we provide can be evaluated inside the proof assistant and computation on real numbers can be used as genuine proofs. This could be helpful for theorems relying on computations [10]. Moreover by formalizing the base, the operations are much more efficient than previous work on base two.

We are now working on improving the efficiency of the inverse thanks to our better description of the computation of series. We also would like to define analytic functions using their taylor series. Better understanding the issues of computing converging power series would be helpful.

References

1. Algirdas A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961.
2. Yves Bertot. Affine functions and series with co-inductive real numbers. *Mathematical Structure in Computer Sciences*, 17(1), 2007.
3. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

4. Alberto Ciaffaglione and Pietro Di Gianantonio. A coinductive approach to real numbers. In Th. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types 1999 Workshop, Lökeberg, Sweden*, number 1956 in LNCS, pages 114–130. Springer-Verlag, 2000.
5. Coq development team. *The Coq Proof Assistant Reference Manual, version 8.0*, 2004.
6. Abbas Edalat and Reinhold Heckmann. Computing with real numbers. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 193–267. Springer, 2000.
7. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for proofs and Programs*, volume 996 of *LNCS*, pages 39–59. Springer Verlag, 1994.
8. Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application for certifying large prime numbers. In U. Furbach and N. Shankar, editors, *3rd International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 423–437. Springer-Verlag, 2006.
9. Jean-Michel Muller. *Elementary Functions, Algorithms and implementation*. Birkhauser, 1997.
10. Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130, pages 21–35. Springer-Verlag, 2006.
11. Milad Niqui. Coinductive correctness of homographic and quadratic algorithms for exact real numbers. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs International Workshop, TYPES 2006, Nottingham, UK, April 18–21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 203–220. Springer-Verlag, 2007.
12. Russell O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 17:129–159, 2007.
13. Arnaud Spiwack. Ajouter des entiers machine coq. `http://arnaud.spiwack.free.fr/papers/nativint.pdf`, 2006.
14. Jean E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, aug 1990.