Affine functions and series with co-inductive real numbers

Yves Bertot

Affine functions and series with co-inductive real numbers

Yves Bertot

November 2005

Abstract

We extend the work of A. Ciaffaglione and P. Di Gianantonio on mechanical verification of algorithms for exact computation on real numbers, using infinite streams of digits implemented as co-inductive types. Four aspects are studied: the first aspect concerns the proof that digit streams can be related to the axiomatized real numbers that are already axiomatized in the proof system (axiomatized, but with no fixed representation). The second aspect re-visits the definition of an addition function, looking at techniques to let the proof search mechanism perform the effective construction of an algorithm that is correct by construction. The third aspect concerns the definition of a function to compute affine formulas with positive rational coefficients. This should be understood as a testbed to describe a technique to combine co-recursion and recursion to obtain a model for an algorithm that appears at first sight to be outside the expressive power allowed by the proof system. The fourth aspect concerns the definition of a function to compute series, with an application on the series that is used to compute Euler's number e. All these experiments should be reproducible in any proof system that supports co-inductive types, co-recursion and general forms of terminating recursion, but we performed with the Coq system [12, 3, 14].

1 introduction

Several proof systems provide data-types to describe real numbers, together with basic operations and theorems that express that the whole data-type is an ordered, complete, and archimedian field [18, 19]. In the Coq system, several approaches have been taken; depending on whether developers wanted to adhere to pure constructive mathematics or more classical approaches. In the classical approach, the type of real numbers is merely "axiomatized", the existence of the type and the elementary operations is assumed and the properties of these operations are asserted as axioms. This approach has been used extensively to provide a large collection of results, going all the way to the description of trigonometric functions, calculus and the like. However, because the type of

real is axiomatized, there is no "physical representation of numbers" and the basic operations correspond to no algorithm.

In an alternative approach, a type of *constructed* numbers may be defined as special kind of data-type and the basic operations may be described as algorithms manipulating elements of this data-type. A. Ciaffaglione and P. Di Gianantoni [7] showed that a well-known representation of real numbers as infinite sequences of redundant digits could easily be implemented inside theorem provers with co-inductive types. We say the digits are redundant because several representation are possible for every number. In the case of [7] the representation is simply inspired from the usual binary representation of of fractional numbers and made more redundant by adding the possibility to use a negative digit.

Our approach is different in the sense that the redundant digit that is added in our representation should not be interpreted as a negative digit, but as an extra positive digit, with a meaning that is intermediary between the existing 0 and 1. However, the notation is the same in spirit, Niqui [27] actually shows that both approaches are special cases of a more general family of representations.

Once we have chosen the way to represent real numbers as a data-type, we proceed by establishing a relation between this data-type and the axiomatized type of real numbers. This is a departure from the prescription of pure constructive mathematics, because we rely on the axioms of that theory to state the correctness of our algorithms. We hope that this approach gives a more direct path to mechanically verified statements about our algorithms.

Once we have provided the basic data-type and its relation with the axiomatized theory of real numbers, we proceed by defining an addition function. We rely on the axiomatized theory to provide an automatic construction of the addition algorithm: this way we only provide guidelines for the construction of the algorithm, without actually describing all 25 cases in the function. The proof of correctness then consists in showing that there is a morphism between the data-type and the axiomatized type. Our contribution in this part is to show how to use the proof search engine to construct a well formed addition function.

We then describe how we can compute affine formulas combining two real values with rational coefficients. For these more general operations, we need to combine co-recursion and well-founded recursion, We show that the function responsible for producing the infinite stream of digits representing the result can be decomposed in two recursive functions. One of the function is guarded co-recursive function as proposed in [14], the second function is well-founded recursive function [3]. Each function satisfies a different form of constraint: the co-recursive function does not need to be terminating, but it must produce at least a digit at each recursive call, while the well-founded function does not need to produce a digit at each recursive call, but it must terminate. Our main contribution in this part is to show that functions that appear at first sight to be outside the expressive power of guarded co-recursion can actually be modeled and proved correct.

In a fourth part, we describe how to compute infinite sums, when enough knowledge is provided to ensure that these infinite sums are converging. We

show how to avoid having to consider the infinity of terms that are parts of the sum. In particular, we exhibit a framework that can be re-used from one series to the other. As applications, we show how to compute the infinite stream representing Euler's number e and to multiply two real numbers represented as infinite streams of digits. In particular, the algorithm we obtain can be executed directly using the reduction mechanism provided in Coq to compute e to a great precision in a reasonable time. For the multiplication, performance is less impressive for execution inside the proof system.

Our work stops here, although the experiments described in this paper seem to open the door to a more complete study of real functions, especially analytic functions with the help of power series.

2 Related work

For numerical computation, real numbers are usually represented as approximations using floating point numbers. These floating point numbers are composed of a mantissa and an exponent, so that the value of the least significant bit in the mantissa varies with the exponent. Still both the exponent and the mantissa have a fixed size, so that there is only a finite number of floating point numbers and real values must be rounded to find the closest floating point numbers. Floating-point based computations are thus only approximation and errors stemming from successive rounding operations may accumulate to the point that some computations can become grossly wrong [25].

In spite of their limitations, Floating-point numbers are used extensively: most processors provide directly an implementation of the elementary operations (addition, subtraction, multiplication, division) according to a standard that gives a precise mathematical meaning to the rounding operations [21]. This standard provides the basis to implement computations with a guaranteed precision [25, 17], sometimes with correctness proofs that can be verified with the help of computer-aided proof tools [9, 20, 6, 31, 5]. In particular, some approaches, named *expansions* make it possible to increase drastically the number of representable number by extending the length of the mantissa [6].

An alternative research direction focusses on computations with number representations that alleviate the need for rounding and associated errors. Among the possible approaches, the best well-known are based on continued fractions [15, 32] or representations with floating points in a fixed base [24]. In the latter case, the representation is very close to the floating-point representations with rounding modes, but the size of the mantissa is not fixed to a given size: it is rather considered as an infinite piece of data, but at any time only of part of it is known and the infinite part is only materalized as the need arises. Exact arithmetics is studied by a large community [22, 26], in particular in the realm of functional programming [23, 4, 1, 13].

Formal proofs around computations on infinite data-structures are a priviledged ground for the use of co-inductive types [8, 14]. First experiments on the topic of exact real number computations using co-inductive types were per-

formed by Ciaffaglione and Di Gianantonia [7] who showed that one could represent infinite sequences of digits with co-inductive types and the basic operations of arithmetics (addition, multiplication, comparison) with simple co-recursive functions, as long as the set of digits was extended to allow for enough redundancy. Niqui [28] also studies the problems of modeling real number arithmetic for use in formal proofs, providing a single point of view to account for continued fractions and infinite sequences of digits. Our approach is very similar to Ciaffaglione and Di Gianantonio's, it only differs in the collection of digits that we consider.

Since they are expected to return infinite data-structures as results, co-recursive functions usually are recursive functions that exhibit a form of infinite recursion. However, the context of formal proofs still requires that general recursion should be avoided. For this reason, co-recursive functions are constrained to produce at least a fragment of the result data at each recursive call. This constraint seems to forbid most of the general recursive algorithms one can envision on infinite data-structures. However, we have shown in [2] that co-recursion could be combined with general forms of well-founded recursion to define functions where some recursive calls are productive while others are not. Di Gianantonio and Miculan also proposed an approach for the same kind of combination, but we believe the work presented here is independent from their results.

3 Redundant digit representation for real numbers

We are all used to the notation with a decimal point to represent real numbers. For instance, we usually write a number between 0 and 1 as a text of the form $0.1354647\cdots$ and we know that the sequence of digits must be infinite for some numbers, actually all those that are not of the form $\frac{a}{10^b}$, where a and b are positive integers. It is a bit less natural, but still easy to understand, that all numbers can be represented by an infinite sequence: for those that have a finite representation, it suffices to add an infinite sequence of zeros. More over the number 1 can also be represented by the sequence $0.999\cdots$.

When we know a prefix of one of these infinite sequences, we actually know the number that is represented up to a certain precision. If the prefix has length n, we actually know precisely the bounds of an interval of length $\frac{1}{10^n}$ that contains the number. We are accustomed to reasoning with these prefixes of infinite sequences and we expect tools to return correct prefixes of an operation's output when this operation has been fed with correct prefixes for the inputs.

In the conventional representation, the number 10 plays a special role: it is the *base*. We can change the base and use digits that are between 0 and the base. For instance, we can use 2 as the base, so that the digits are only 0 and 1. The number $\frac{1}{2}$ can then be represented by the sequence $0.1000\cdots$ and the number 1 can be represented by the sequence $0.1111\cdots$. For a sequence

$0.d_1d_2\cdots$, the number being represented is:

$$\sum_{i=0}^{\infty} \frac{d_i}{2^i}.$$

the following equalities hold:

$$\begin{aligned} 0.0s &= \frac{0.s}{2} \\ 0.1s &= \frac{0.s+1}{2} \end{aligned}$$

The last important property is that a prefix with n digit gives an interval of width $\frac{1}{2^n}$ that contains the number represented by the infinite sequence. In the rest of this paper, we will carry on with this representation using base 2 (but the set of digits will change).

For the computation of basic operation, this conventional representation is not really well adapted. Here is an example that exhibits the main flaw of this representation. The numbers $\frac{1}{3}$ and $\frac{1}{6}$add up to give $\frac{1}{2}$. However, the infinite sequences for these numbers are given in the following equations:

$$\begin{aligned} \frac{1}{3} &= 0.01010101\cdots \\ \frac{1}{6} &= 0.00101010\cdots \\ \frac{1}{2} &= 0.10000000\cdots = 0.01111111\cdots \end{aligned}$$

The following reasoning steps justify the first equation:

$$0.01010101\cdots = \sum_{i=1}^{\infty} \frac{1}{2^{2i}} = \frac{1}{1-\frac{1}{4}} - 1 = \frac{1}{3}$$

Similar proofs can be used to justify the other equations. If s is a prefix of $0.01010101\cdots$, s is also the prefix of all numbers between $s00\cdots$ and $s111\cdots$. These two numbers are rational numbers of the form $\frac{a}{2^b}$ and none these numbers can be equal to $\frac{1}{3}$. Thus, we actually have an interval of possible values that contains both values that are smaller and values that are larger than $\frac{1}{3}$. The same property occurs for the representation of $\frac{1}{6}$. When considering the sum of values in the interval around $\frac{1}{3}$ and values in the interval around $\frac{1}{6}$, the results are in an interval that contains both values that are smaller and values that are larger than $\frac{1}{2}$. However, numbers of the form $0.1\cdots$ can only be larger than $\frac{1}{2}$ and numbers of the form $0.0\cdots$ can only be smaller than $\frac{1}{2}$. Thus, even if we know the inputs with a great precision, we must indefinitely delay the decision and require more precision on the input before choosing the first digit of the result: we need to know the inputs with infinite precision before stating the first digit of the result.

We solve this problem by adding a third digit in the notation. This digit provides a way to express that the interval given by a prefix has $\frac{1}{2}$ in its interior. This new digit adds more redundancy in the representation. We now have three digits, even though we still work in base 2.

- A digit `L` is used like the digit 0. If x is an infinite sequence of digits representing the number v, the sequence `L`x represents $v/2$.
- A digit `R` is used like the digit 1. If x is an infinite sequence of digits representing the number v, the sequence `R`x represents $v/2 + 1/2$.
- A digit `C` is used with the following meaning: if x is an infinite sequence of digits representing the number v, the sequence `C`x represents $v/2 + 1/4$.

From now on, we consider only numbers in the interval $[0, 1]$ and we drop the first characters "0." when writing a sequence of digits. We will confuse a sequence of digits and the real number that it represents. In the same spirit, we will use the same notation for a digit and the function it represents. For instance, the function `L` is the function that maps x to $x/2$. Last, we will associate the digits `L`, `R`, `C` to the intervals $[0, \frac{1}{2}]$, $[\frac{1}{2}, 1]$, and $[\frac{1}{4}, \frac{3}{4}]$, respectively. These interval will be called *basic intervals*.

The redundancy of the new digit gives a very simple property: a number that can be written `CL`x can also be written `LR`x and a number that can be written `CR`x can also be written `RL`x. This property will be re-used several times in this paper.

3.1 Formal details of infinite sequences and real numbers

In our formalization, we benefit from theories that state the main properties of natural numbers (type `nat`), integers (type `Z`), and real numbers (the type is usually written `R`, but in this article we shall write it as `Rdefinitions.R` to avoid ambiguity with the "digit" `R`). The two integer types come with addition, subtraction, and multiplication, while the type of real numbers is also equiped with division. The integer types are actually described as inductive types and the basic operations are implemented as recursive functions. For the real numbers, the existence of the type, two constants 0 and 1, the operations, comparison predicates, and the properties of these operation (associativity, distributivity, inverse, etc.) are assumed. Among the assumed features, there is an axiom that expresses that the type is complete, with the following statement.

$$\forall E : \texttt{Rdefinition.R} \to \texttt{Prop}, \texttt{bound}\ E \to (\exists x, E\ x) \to \overline{\exists} m, \texttt{is\_lub}\ E\ m.$$

Three concepts appearing in this statement need some clarification. First, `bound` is a predicate that expresses that there exist an upper bound to the real numbers that satisfy E. Second, `is_lub` expresses that m is the least upper bound of E. Third, the existential quantification $\overline{\exists}$ is a special quantification that comes with the equivalent of the axiom of choice. This means that whenever we have exhibited a property E and prove that it is bound, we can construct a function

that returns its least upper bound. The form of this completeness axiom implies that we are not working with constructive mathematics. To be more precise, we provide a constructive theory of real numbers, but all the justifications of correctness, which rely on the axiomatized real numbers, are non-constructive.

We do not use the completeness axiom directly, but a function `growing_cv` that takes as argument an infinite sequence, a proof that this sequence is growing, and a proof that this sequence is bounded, and returns a value v and a proof that v is the limit of the sequence.

Aside from its treatement of limits of sequences, the axiomatization of real numbers also provide a few decision procedures. The decision procedure `field` [11] solves equalities between rational expression, occasionally leaving proofs obligation to make sure denominators are non zero. The decision procedure `fourier` determines when a collection of inequations concerning affine formulas with rational coefficients is satisfiable.

The type of digits is described as an enumerated type:

```
Inductive idigit : Set := L | R | C.
```

The type of infinite sequences of digits is based on a polymorphic type of streams, which is defined as a co-inductive type:

```
CoInductive stream (A:Set) : Set :=
  Cons : A -> stream A -> stream A.

Implicit Arguments Cons.
Infix "::" : Cons : stream_scope.
Open Scope stream_scope.
```

Thus the type of infinite sequences of digits is the type `stream idigit`. We can use co-recursion to construct simple values in this type. For instance, 0 and 1 are represented by the infinite sequences `LLL...` and `RRR...`:

```
Cofixpoint zero : stream idigit : L::zero.

Cofixpoint one : stream idigit := R::one.
```

To relate infinite streams of digits with real numbers we define a a co-inductive relation.

```
CoInductive represents: stream idigit -> Rdefinitions.R -> Prop:=
  reprL : forall s r, represents s r -> (0 <= r <= 1)%R ->
          represents (L::s) (r/2)
| reprR : forall s r, represents s r -> (0 <= r <= 1)%R ->
          represents (R::s) ((r+1)/2)
| reprC : forall s r, represents s r -> (0 <= r <= 1)%R ->
          represents (C::s) ((2*r+1)/4).
```

This relation really states that infinite streams are only used to represent numbers between 0 and 1 and it confirms the correspondance between `L`, `R`, and `C` on

the one hand and the functions $x \mapsto \frac{x}{2}$, $x \mapsto \frac{x+1}{2}$, and $x \mapsto \frac{2x+1}{4}$ on the other hand.

An alternative approach to relate sequences of digits and real numbers is to build a function that maps an infinite sequence to a real value. Every prefix of an infinite sequence corresponds to an interval that contains all the values that could have the same prefix. As the prefix grows, the new intervals are included in each other, and the size is divided by 2 at each step. We defined a function `bounds` to compute the interval corresponding to the prefix of a given length for a given sequence. Actually, this function takes as arguments a digit sequence and a number n and it computes the bounds of an interval that contains all the real numbers whose representation shares the same prefix of length n. This function is programmed as a recursive function that is structural recursive in n. The notion of *structural recursion* is similar to *primitive recursion*. It simply means that this function could easily be defined in any theorem proving assistant that provides a notion of natural numbers and recursive definition on these numbers. In the Coq system, a structural recursive function also has the property that it is easily re-used in proof search tools. The definition is based on the following equations:

$$\begin{aligned}
\texttt{bounds}(\ldots, 0) &= [0, 1] \\
\texttt{bounds}(\texttt{L}x, n+1) &= [\frac{a}{2}, \frac{b}{2}] \quad \text{where} \quad \texttt{bounds}(x, n) = [a, b] \\
\texttt{bounds}(\texttt{C}x, n+1) &= [\frac{a}{2} + \frac{1}{4}, \frac{b}{2} + \frac{1}{4}] \\
\texttt{bounds}(\texttt{R}x, n+1) &= [\frac{a}{2} + \frac{1}{2}, \frac{b}{2} + \frac{1}{2}]
\end{aligned}$$

In practice, we do not manipulate real numbers in this function, but integers. The result of the function is a triplet (a, b, k) such that the interval is $[\frac{a}{2^k}, \frac{b}{2^k}]$.

We then define a function that maps a stream of digits to a sequence of real numbers, which are the lower bounds of the intervals. This function is called `si_un` and is defined by the following text:

```
Definition si_un (s:stream idigit) (n:nat) : Rdefinitions.R :=
   let (p,k) := bounds n s in let (a,b) := p in
    (IZR a/IZR(2^k))%R.
```

We then show that for every stream `s`, the sequence `si_un s` is a growing sequence.

The Coq non-constructive formalization of real number nevertheless provides a function that maps any Cauchy sequence of real values to its limit. Actually, any sequence digits also represents an infinite sequence of intervals, where each interval is included in the previous one and has its size equal to half the size of the previous one. This is enough to express that the digit sequences defines a Cauchy sequence (for example by taking the intervals lower bounds). Using this approach we have defined a function `real_value` and we proved basic theorems about this function:

```
Theorem represents_real_value :
  forall s, represents s (real_value s).

Theorem represents_equal :
  forall s r, represents s r -> real_value s = r.
```

3.2 Addition

It is well-known that addition of two infinite sequences of redundant digits can be described as a simple automaton that inputs digits from both arguments and produces digits as regularly. Two approaches can be taken: either this automaton is understood as a program that keeps a carry as it progresses in processing the inputs, or it is a can be viewed a program that performs a little look-ahead before outputing the result and processing the rest, maybe with a slight modification of the first digit in both inputs. The algorithm we describe follows the second approach.

Our modelization contains two parts. The first part is a function that computes the arithmetic mean of two real values (in other words, the half sum). The second function is a function that computes the double of real number. Actually, our algorithms only take inputs in [0,1] and produce outputs in the same interval, so that the second fonction only returns a meaningful result when the input is smaller than 1/2.

For the half-sum, the structure of the algorithm is as follows: if the inputs have the form $d_1 d_2 x$ and $d_3 d_4 y$, then the algorithms outputs a digit d and calls itself recursively with the new inputs $d_5 x$ and $d_6 y$. Our formalization of the algorithm does not prescribe precisely the correct values for d, d_5, and d_6 for each possible choice of d_1, ..., d_4. Instead, we write a program that tries possible values and chooses only the ones that make the algorithm satisfy the specification of an addition function.

Here is an example, suppose that $d_1 = \mathtt{L}$ and $d_3 = \mathtt{R}$, in this case we can choose $d = \mathtt{C}$ and $d_5 = d_2$ and $d_6 = d_4$, because the following equalities hold, using the interpretations of digits as functions:

$$\begin{aligned}
\mathtt{half_sum}(\mathtt{L}d_2x, \mathtt{R}d_4y) &= \frac{\mathtt{L}d_2x + \mathtt{R}d_4y}{2} \\
&= \frac{\frac{d_2x}{2} + \frac{d_4y}{2} + \frac{1}{2}}{2} \\
&= \frac{d_2x}{4} + \frac{d_4y}{4} + \frac{1}{4} \\
&= \frac{\frac{d_2x+d_4y}{2}}{2} + \frac{1}{4} \\
&= \mathtt{C}(\mathtt{half_sum}(d_2x, d_4y))
\end{aligned}$$

In this case, it is not necessary to scrutinize d_2 and d_4 to decide the value of d and the arguments for the recursive call.

Here is a second example where d_5 and d_6 are modified with respect to d_2 and d_4. We suppose $d_1 = \mathtt{C}$, $d_2 = \mathtt{L}$, and $d_3 = \mathtt{L}$. In this case, the following equalities hold:

$$\begin{aligned}
\mathtt{half_sum}(\mathtt{CL}x, \mathtt{L}d_4y) &= \frac{\frac{x}{4} + \frac{1}{4} + \frac{d_4y}{2}}{2} \\
&= \frac{\frac{x}{2} + \frac{1}{2} + \frac{d_4y}{2}}{2} \\
&= \mathtt{L}(\mathtt{half_sum}(\mathtt{R}x, d_4y))
\end{aligned}$$

In this case, it is not necessary to scrutinize d_4, but the value d_5 is modified with respect to d_6.

In general, there could be 81 cases in the definition of the function `half_sum`, because we scrutinize two digits on both sides, but as we saw in the previous two examples there are cases when some digits do not need to be scrutinized. In practice, there are only 25 cases and we describe in the next section the approach we took to generate these cases automatically.

3.3 Automatic generation of function code

In the process of defining the function we need to consider *real number patterns*. Real number patterns are sequences of digits where the tail is replaced with a variable. These patterns occur naturally when we define a function by cases. For instance, Lx is a pattern, CLy is a pattern, etc. To every pattern we can associate a length, which is the number of digits before the variable occurs. Patterns can be manipulated in the tactic programming language of most theorems (in Isabelle [30], the tactic programming language is ML [29]). In the Coq system, the tactic programming language is Ltac [10].

Combining the computation of length and bounds for a real number pattern, we can associate an interval to each pattern. This interval is the smallest interval that contains all real numbers represented by the real-number pattern, when the variable in this pattern is replaced by an arbitrary real number in [0,1]. We can then combine the intervals associated to two input real number patterns to know the interval containing all possible values for the half sum of the corresponding real numbers.

For instance, if we have two input real number patterns CLx, and Ly, we can associate the interval $[\frac{1}{4}, \frac{1}{2}]$ to the first pattern and the interval $[0, \frac{1}{2}]$ to the second pattern and we can infer that the half-sum of a number represented by the first pattern and a number represented by the second pattern will necessarily be inside $[\frac{1}{8}, \frac{1}{2}]$.

Thus, for a given pair of input patterns, we are able to determine an interval that contains the half-sum of any pair of numbers represented by these patterns. If this interval is inside one of the intervals associated to the digits `L`, `C`, or `R`, then the output can start with the corresponding digit.

In our example, the interval $[\frac{1}{8}, \frac{1}{2}]$ is inside the interval $[0, \frac{1}{2}]$ and the output can start with the digit `L`.

In general, the result interval is not inside an interval associated to a digit and it is necessary to collect more information about the input. This is done by replacing the variable in one of the input real number patterns with a more specific pattern. In practice, this means replacing the current pair of input patterns with three new patterns.

For instance, when the two input patterns are $\mathtt{L}x$ and $\mathtt{C}y$, our approach makes it possible to infer that the result in between $\frac{1}{8}$ and $\frac{5}{8}$. Our procedure proceeds by inserting in the half-sum algorithm a case analysis on x, thus relacing $\mathtt{L}x$ with $\mathtt{LL}x'$, $\mathtt{LC}x'$, and $\mathtt{LR}x'$. We can then restart our bounds computation for the pairs of patterns $(\mathtt{LL}x', \mathtt{C}y)$, $(\mathtt{LC}x', \mathtt{C}y)$, and $(\mathtt{LR}x', \mathtt{C}y)$. With a new bounds computation, we obtain that the digit `L` is a good result digit for the first case and the digit `C` is a good result digit for the last case.

When the result interval does not fit a digit interval, we have a choice of which of the two patterns in the input pair to replace with three more specific patterns. We programmed our search procedure to make sure it makes the shortest pattern more specific, when such a shortest pattern exists. For the half-sum function, this process stops when the two input patterns have length 2.

Once we have computed the first digit of the output, we still have to compute the arguments of the recursive calls. We know that these recursive calls will contain the variables that occur in the input patterns, but the digits that appear in front of these variables may change. To compute the change, we compare the lower bound of the interval associated to the pair of input patterns *with a length of 1* with the lower bound of the chosen digit. The difference of bounds may be either 0, $\frac{-1}{8}$, or $\frac{1}{8}$.

For instance, if the input patterns are $\mathtt{CL}x$ and $\mathtt{L}y$ and the output digit is `L`, then the lower bound of the result is $\frac{1}{8}$, while the lower bound of the digit is 0.

When the difference is non zero, this difference needs to be compensated in the recursive call. We can compensate a difference by adding or removing $\frac{1}{2}$ from the inputs of the recursive call. There are several way in which this possible: if an input has the form $\mathtt{L}x$, then replacing this input with $\mathtt{C}x$ adds $\frac{1}{4}$, to the argument of the recursive half-sum, replacing this input with $\mathtt{R}x$ adds $\frac{1}{2}$. Because the function that we compute is a half-sum, a compensation of $\frac{1}{2}$ appears as compensation of $\frac{1}{4}$ in the result of the recursive call. Because, the recursive calls appears behind a digit in the overall result, this compensation then amounts to a compensation of $\frac{1}{8}$. The procedure sometimes needs to modify both arguments to obtain the right compensation, because a compensation of $\frac{1}{2}$ cannot be obtained from an input whose first digit is `C`.

For instance, if the inputs are $\mathtt{LC}x$ and $\mathtt{CC}y$ then the result is inside $[\frac{1}{4}, \frac{1}{2}]$ and the output digit can be chosen as `L`. However, the bounds of the result for a pair of streams $\mathtt{L}\ldots\ \mathtt{C}\ldots$ are

$$\frac{1}{8}, \frac{5}{8}$$

. The difference between the lower bounds is $\frac{-1}{8}$. The arguments to the recursive call before compensation are $\mathtt{C}x$ and $\mathtt{C}y$. To compensate the difference we add $\frac{1}{2}$

to the arguments of the recursive call, by adding $\frac{1}{4}$ to each of the two arguments, replacing them with $\mathtt{R}x$ and $\mathtt{R}y$. This is also justified by the following equations:

$$\begin{aligned}\frac{1}{2}(\mathtt{LC}x + \mathtt{CC}y) &= \frac{1}{8}x + \frac{1}{8}y + \frac{1}{4} \\ &= \frac{1}{2}(\frac{1}{4}x + \frac{1}{4} + \frac{1}{2}) \\ &= \mathtt{L}(\frac{\frac{x}{2} + \frac{1}{2} + \frac{y}{2} + \frac{1}{2}}{2}) \\ &= \mathtt{L}(\frac{1}{2}(\mathtt{R}x + \mathtt{R}y))\end{aligned}$$

Altogether these processes are enough to define the half-sum function. To have a complete addition function, we need to add a function that multiplies its argument by 2. This function is based on the following remarks.

- The double of a number of the form $\mathtt{L}x$ is simply x,
- The double of a number of the form $\mathtt{R}x$ is either 1 or outside the interval [0,1],
- The double of a number of the form $\mathtt{C}x$ is a number of the form $\mathtt{R}x'$ where x' is the double of x (hence the algorithm exhibits a co-recursive call).

3.4 Formal details for addition

Multiplication by 2 is represented by the following theorem:

```
Cofixpoint mult2 (n:stream idigit) : stream idigit :=
  match n with L x => x | C x => R::mult2 x | R x => one end.
```

We used this function for some of our correctness statements.
For multiplication by 2, we only used the `represents` relation:

```
Theorem mult2_correct :
  forall x u, (0 <= u <= 1/2)%R -> represents x u ->
  represents (mult2 x)(2*u)%R.
```

The formal description of the half sum function is done in *definition by proof mode*: the whole function type is manipulated like a logical goal and the function body is constructed by *tactics* that are normally used only to build proofs.

The function type is

```
stream idigit -> stream idigit -> stream idigit.
```

Proof mode term construction consists in providing a "hole" that needs to be filled up with this type. A first tactic `cofix` can be used to express that the result will be a recursive value. A second tactic `intros` can be used to express that the value will also be a function, at the same time giving names to the function's formal parameters. A tactic `case` is often used to express that the

various possible cases for one of the inputs will be analysed. This tactic produces several "holes", each of them to be filled with the value corresponding to one of the cases. In the middle process, we usually have a collection of holes to fill in and for each of them we know the type of the value that should be returned. In our case, the type of the result is always `stream idigit`; this type does not give any information about which case is currently considered. To provide more information we define an artificially dependent type, which we call `help_type`:

```
Definition help_type (d1 d2:stream idigit) := stream idigit.
```

This help type makes it possible to take better benefit from the behavior of the `case` tactic. If the current hole should be filled with a value of type `stream idigit` and we use a tactic `case d` where `d` is digit, then the hole is replaced with a term of the form

```
match d with L => ?1 | R => ?2 | C => ?3 end.
```

The expressions `?1`, `?2` and `?3` are holes which need to be replaced with values of type `stream idigit` and which should be filled in by different values, but the expected type does not help in knowing which hole corresponds to which input. If we replace the type for the initial hole by a type of the form

```
help_type (d::tl1)(d'::tl2)
```

and we use the same tactic `case d`, then the hole is replaced with term of the following form:

```
match d return help_type (d::tl1)(d'::tl2) with
 L => ?1 | R => ?2 | C => ?3
end
```

Then the expected types for the three new holes are different. The first one should have type `help_type (L::tl1)(d'::tl2)`, and so on. Thus, subsequent tactic can know which is the case being considered simply by observing the expected type. Of course, because `help_type` is actually a constant function, all types are the same, but during the proof process they are annotated with information about the case being considered[1].

We use two help types, the second one is given by the following definition:

```
Definition help_type2 (d1 d2:stream idigit)(d3:idigit) :=
   stream idigit.
```

We then follow the steps described in the previous section. The computation of the bounds is described by a function named `bounds`, this function returns a triplet (a, b, c) which represents the interval $[\frac{a}{c}, \frac{b}{c}]$. Three auxiliary functions `ub`, `lb`, and `den` are defined to return a, b, c separately. The process of producing the first digit of the output or decomposing arguments to more specific patterns is described in a function named `produce_add_cases`:

[1] This technique was taught to the author during a friendly conversation with J.C. Filliâtre and L. Théry. The latter indicated that an early inspiration could be found in P. Crégut developments for the `omega` tactic.

```
Ltac produce_add_cases :=
 match goal with |- help_type ?u ?v =>
  let bb1 := lb u in let bb2 := lb v in
  let tb1 := ub u in let tb2 := ub v in
  let d1 := den u in let d2 := den v in
  let test1 := eval compute in
    (bb1*d2 + bb2*d1 ?= d1*d2)%Z in
  ...
  match test1 with
    Gt => apply (Cons R); change (help_type2 u v R)
  | Lt =>
      ...
        match test4 with
          Lt => try (mk_cases; produce_add_cases)
        | _ => apply (Cons C); try change (help_type2 u v C)
        end
      ...
  end
end.
```

As we see in this outline of the function's body, when the various tests made on the bounds of the result do not make it possible to choose an output digit, one of the inputs is decomposed further by the tactic `mk_cases` and the testing phase is re-started through a recursive call to `produce_add_cases`. To understand this script, one should also understand that the semi-colon acts as a *span* tactic combinator: when tactics t_1 and t_2 are combined in the expression $t_1;t_2$, all the goals generated by t_1 are processed by t_2. In this case, we know that `mk_cases` produces three goals, which are processed using by three recursive calls to `produce_add_cases`.

The process of computing the difference of bounds is described by a function named `compute_borrow`, and the process of building the right arguments for the recursive calls is described in a function named `mk_borrow`.

The whole definition of `half_sum` can then be condensed in a few lines, thanks to the semi-column combinator. One should be aware that `produce_add_cases` actually produces 25 goal, each of which is processed by the tactic `match goal ...` that follows.

```
Definition half_sum (x y:stream idigit) : stream idigit.
cofix.
intros x y; change (help_type x y).
produce_add_cases;
match goal with |- help_type2 (?x::?tl1) (?y::?tl2) ?z =>
 let r := eval compute in (compute_borrow x y z) in
 match r with 0 => exact (half_sum tl1 tl2)
   | _ => mk_borrow half_sum r tl1 tl2
 end
```

```
end.
Defined.
```

While this definition produces a function that is correct by construction, we still need to produce a correctness theorem. This is also done with a systematic approach. All 25 cases need to be checked one-by-one, but the proofs are easily boiled down to easy arithmetical verifications that can be performed with existing tactics for comparisons of affine formulas (the tactic `fourier`) and equalities of fractional formulas in a field (the tactic `field`) [11].

We believe that our technique of definition can easily be reproduced for different sets of digits or for other simple binary operations like subtraction.

3.5 subtraction

In this section we discuss several approaches to subtraction. One first approach uses a few intermediary function. The first intermediary value mimicks the opposite function. Of course the opposite function cannot be defined from [0,1] to [0,1], but the result can be translated so that we can easily compute the function that maps x to $1 - x$. Here is the general definition, where we name the function `minus_aux`:

$$\begin{aligned}
\texttt{minus_aux}(\texttt{L}(x)) &= \texttt{R}(\texttt{minus_aux}(x)) \\
\texttt{minus_aux}(\texttt{R}(x)) &= \texttt{C}(\texttt{minus_aux}(x)) \\
\texttt{minus_aux}(\texttt{R}(x)) &= \texttt{L}(\texttt{minus_aux}(x))
\end{aligned}$$

These equations are justified through simple computations. For instance, the last equation is justified with the following reasoning steps:

$$\begin{aligned}
\texttt{minus_aux}(\texttt{R}(x)) &= 1 - (\frac{x}{2} + \frac{1}{2}) \\
&= \frac{1}{2} - \frac{x}{2} \\
&= \frac{1}{2}(1 - x) \\
&= \texttt{L}(\text{minus_aux}(x)
\end{aligned}$$

Combining `minus_aux` with addition, we can easily compute the binary function that maps x and y to $1+x-y$. Of course, this function returns a meaningful result only when x is smaller than y. This limitation is a problem.

We should remember that addition was first defined as half sum, and combining `minus_aux` with `half_sum` actually gives a function that maps $\frac{1+x-y}{2}$. Now, if we really want to have a subtraction, we can remove the $\frac{1}{2}$ offset, but this is only valid if the result is larger than $\frac{1}{2}$. We use an other auxiliary function, which we name `minus_half` and is defined by the following equations:

$$\begin{aligned}
\texttt{minus_half}(\texttt{R}x) &= \texttt{L}x \\
\texttt{minus_half}(\texttt{L}x) &= \texttt{zero} \\
\texttt{minus_half}(\texttt{C}x) &= \texttt{L}(\texttt{minus_half}(x))
\end{aligned}$$

The first of these equations is trivial to justify. The second is justified by the fact that the only value inside $[0, \frac{1}{2}]$ for which $x - \frac{1}{2}$ belongs to $[0, 1]$ is $\frac{1}{2}$ and the result is 0 in this case. The third equation is justified by the following reasoning steps:

$$\begin{aligned} \texttt{minus\_half}(\texttt{C}x) &= \frac{x}{2} + \frac{1}{4} - \frac{1}{2} \\ &= \frac{x}{2} - \frac{1}{4} \\ &= \frac{x - \frac{1}{2}}{2} \end{aligned}$$

$$\frac{\texttt{L}}{(}\texttt{minus\_half}(x))$$

4 Parameterized affine operations

In this section, we study another approach to addition, with the encoding of a more general function that computes affine formulas in two real values with rational coefficients. More precisely, we want to compute the value

$$\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'}$$

When a, b, c are positive integers, a', b', c' are non-negative integers (a, b, c may be null, but not the others), and x and y are real numbers, given as infinite sequences of digits.

4.1 Algorithm main structure

Choosing the digits of the result is based on the following remarks:

1. Even without observing x and y, we already know that they are numbers between 0 and 1. The result then lies in the interval whose bounds are

$$\frac{c}{c'} \qquad \text{et} \qquad \frac{ab'c' + a'bc' + abc'}{a'b'c'}.$$

2. As soon as we can state that the result is inside of one of the intervals denoted by `L`, `R`, or `C`, it is possible to produce a digit and perform a co-recursive call with a new affine formula.

3. If the extremes are badly placed, we can choose an interval associated to a digit that is sure to contain the result. In this case, we scrutinize x and y and observe their first digit. As a result, we obtain a new estimate of the interval that may contain the result, whose size is half the previous size. We can then perform a recursive call with a new affine formula. In the long run, we are forced to get to a situation where a digit can be chosen and a co-recursive call can be performed. In fact, this condition is guaranteed as soon as the distance between extrema is shorter than 1/4.

Let us study two examples. In the first example, suppose that the property $c/c' \geq 1/2$ holds. We know that the result is larger than $1/2$ and we ca produce a `R` digit. The following computation takes place:

$$\begin{aligned} \frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'} &= \mathtt{R}(2 \times (\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'}) - 1) \\ &= R(\frac{2a}{a'}x + \frac{2b}{b'}y + \frac{2 * c - c'}{c'}) \end{aligned}$$

There is a recursive call with a new affine formula, where all the coefficient are positive integers or non-negative integers as required.

In a second example, suppose that the properties $x = \mathtt{L}x'$ and $y = \mathtt{R}y'$ hold. The following computation can take place:

$$\begin{aligned} \frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'} &= (\frac{a}{a'}\frac{x'}{2}) + (\frac{b}{b'}\frac{y'+1}{2}) + \frac{c}{c'} \\ &= \frac{a}{2a'}x' + \frac{b}{2b'}y' + \frac{bc' + 2b'c}{2b'c'} \end{aligned}$$

Here again, we can have a recursive call with a new affine formula, no digit has been produced (therefore the recursive call cannot be a co-recursive call) but the distance between the extrema in the new formula is $a/2a' + b/2b'$, the half of $a/a' + b/b'$, which was the distance between extrema for the initial affine formula.

4.2 Formal details for affine formulas

We define a data-type to represent affine formulas, with 6 integers and 2 real numbers given as infinite digit sequences.

```
Record affine_data : Set :=
 {m_a : Z; m_a' : Z; m_b : Z; m_b' : Z; m_c : Z; m_c' : Z;
  m_x : stream idigit; m_y : stream idigit}.
```

We define a predicate `positive_coefficients` on this data-type to state that the integers are positive or non-negative as required. We then define a function called `axbyc` to compute the affine formula. Its type is the following one:

```
axbyc : forall x: affine_data,
         positive_coefficients x -> stream idigit.
```

This function relies an an auxiliary function that we called `axbyc_rec`. This auxiliary function performs the recursive calls in the phase 3 in the previous section. It is a well-founded recursive function, which returns elements in yet another data-type called `decision_data`. This new data-type cobines an `affine_data` value, a proof that it has positive coefficients, a proof that this new affine formula satisfies the required properties to produce a digit for the result, and a proof that the new affine formula represents the same real number as the initial one.

The data-type `decision_data` is described as inductive type with three constructors, `caseR`, `caseL`, `caseC`, whose meaning should be self-explanatory. The type of the `axbyc_rec` function is the following one:

```
axbyc_rec : forall x, positive_coefficients x -> decision_data x
```

With the help of this function the main function is described in a few lines:

```
CoFixpoint axbyc (x:affine_data)
   (h:positive_coefficients x):stream idigit :=
  match axbyc_rec x h with
    caseR y Hpos Hc _ =>
     R::(axbyc (prod_R y) (A.prod_R_pos y Hpos Hc))
  | caseL y Hpos _ _ =>
     L::(axbyc (prod_L y) (A.prod_L_pos y Hpos))
  | caseC y Hpos H1 H2 _ =>
     C::(axbyc (prod_C y) (A.prod_C_pos y Hpos H2))
  end.
```

The functions `prod_R`, `prod_L`, `prod_C` perform the affine formulas transformations that correspond to the digit being produced. The theorems `..._pos` provide proofs that the recursive calls are always performed with arguments that have positive coefficients.

Using the `real_value` function, we also defined a function `af_real_value` that maps an affine formula to the real number it represents. This function is instrumental to state the correctness theorem for our `axbyc` function.

```
axbyc_correct:
  forall x, forall H :positive_coefficients x,
   (0 <= af_real_value x <= 1)%R ->
   real_value (axbyc x H) = af_real_value x.
```

While this theorem relies on an equality between real values, we actually proved a statement based on the `represents` relation and this proved was based on a co-recursive proof:

```
axbyc_correct_aux :
  forall x:affine_data, forall H :positive_coefficients x,
  (0 <= (af_real_value x) <= 1)%R ->
   represents (axbyc x H) (af_real_value x).
```

This proof relies on a case analysis of the result of the function call `axbyc_rec x H`. Three cases appear, corresponding to the three constructors of the type `decision_data` and each case corresponds to one of the constructors of the `represents` relation. Once the constructor have been applied, the proof can easily be concluded with the help of a co-recursive hypothesis.

5 Computing series

A series is an infinite sum of values. Knowing how to compute series can help in computing famous constants like e (Euler's number) and to implement the multiplicatioin of two real numbers. It turns out our approach of multiplication, based on series yields an implementation of multiplication that is very close to the implementation in [7].

5.1 Algorithm main structure

In our current approach, we only consider series where the limit is between 0 and 1. Such a series is usually written $\sum_{i=0}^{\infty} a_i$. Studying series is very close to studying converging sequences, since it is enough to consider the sequence $u_n = \sum_{i=0}^{n} a_i$. Each element of the sequence can then be approximated as combination of finite sums; to get an approximation of the limit, it is then necessary to study how well one given finite sum approximates the limit.

Computing the n first digit of a sequence representing the limit means computing the limit up to a given precision. Let's say we want to represent this precision with a given ϵ. If we know that a given element u_n is closer than $\frac{\epsilon}{2}$ to the limit and that we can compute u_n to a precision better than $\frac{\epsilon}{2}$, then we are able to compute the limit to the required precision. To compute the first n digits of $\sum_{i=0}^{\infty} a_i$, it is then enough to compute the first $n+1$ digits of $\sum_{i=0}^{k} a_i$ for some k, and this in turn requires computing the first p digits of each of the terms a_i, where p is a number larger than n.

In practice, we restrict again our study to series that converge regularly, the behavior at infinity being controlled by a bound function μ:

$$\forall m.n \leq m \Rightarrow |\sum_{i=m}^{\infty} a_i| < \mu(n).$$

We actually formalize the computation of a function f that has the following informal specification:

$$f(x, y, n) = x + y \times \sum_{i=n}^{\infty} a_i.$$

Intuitively, y represents the inverse of the precision that is been reached in the computation. The number x represents a partial sum multiplied by y. If we know both x and $y \times \mu(n) < \frac{1}{8}$ with enough accuracy, we able to choose the first digit of $x + y\sum_{i=n}^{\infty} a_i$. We can then perform the following computation

$$f(x, y, n) = d::(2x - c, 2y, n),$$

where d is one of the digit and c is the offset corresponding to this digit, $c = 1$ if $d = \texttt{R}$, $c = 1/2$ if $d = \texttt{C}$. If $y \times \mu(n)$ is not small enough, we perform the following computation:

$$f(x, y, n) = f(x + y \times \sum_{i=n}^{\phi(y,n)} a_i)$$

where $\phi(y, n)$ is chosen so that $y \times \mu(\phi(y, n)) < \frac{1}{16}$. If such a value cannot be found, the series is probably not converging.

In details our algorithms works as follows:

1. We compute a value $\phi(y, n)$, larger than n so that $y \times (\phi(y, n))$ is smaller than $\frac{1}{16}$,

2. We compute the number $v = x + y \times \times \sum_{i=n}^{\phi(y,n)-1} a_i$ by repeated binary additions. We then perform a case analysis on this number.

3. If v has the form $\texttt{RR}v'$, $\texttt{RC}v'$, $\texttt{RLC}v''$, or $\texttt{RLR}v''$. In this case, we are certain that $v + \sum_{i=\phi(y,n)}^{\infty} a_i$ is larger than $1/2$ and the result can have the form $\texttt{R}(f(\texttt{R}v', 2y, \phi(y, n)))$, $\texttt{R}(f(\texttt{C}v', 2y, \phi(y, n)))$, $\texttt{R}(f(\texttt{LC}v', 2y, \phi(y, n)))$, or $\texttt{R}(f(\texttt{LR}v', 2y, \phi(y, n)))$, respectively.

4. If v has the form $\texttt{CC}v'$, $\texttt{CLC}v''$, $\texttt{CLR}v''$, $\texttt{LL}v'$, $\texttt{LC}v'$, $\texttt{LRL}v''$, $\texttt{LRC}v''$ then this case can be treated like the previous case, producing a first digit that is the same as the first digit of v, with a recursive call to f with v without its first digit, $2y$, and $\phi(y, n)$ as arguments.

5. if v has the form $\texttt{RLL}v''$, then v could also be represented using $\texttt{CRL}v''$ and this case is already considered above. The same goes for the cases LRR, CLL, and CRR, using CLR, LRL, and RLR as respective alternatives.

Computing $phi(y, n)$ and $sum_{i=n}^{\phi(y,n)-1} a_i$ depends on the series being studied. This may rely on a well-founded recursive function. Each recursive call maintains the invariant $y \times \mu(n) < 1/8$ and multiplies y by 2.

5.2 Series with positive terms

When we know that the a_i terms are all positive, we do not need to use $\frac{1}{16}$ to bound the infinite remainder of the sum. The computation technique can be simplified as follows:

1. We compute a vale $\phi(y, n)$ so that $y \times \phi(y, n)$ is smaller than $\frac{1}{8}$,

2. We compute $v = x + y \times \sum_{i=n}^{\phi(y,n)-1} a_i$ and perform a case analysis on its first digits:

3. if v has the form $\texttt{R}v'$, we are sure that the result is larger than $1/2$, the result is $\texttt{R}(f(v', 2y, \phi(y, n)))$,

4. if v has the form $\texttt{C}v'$ but not $\texttt{CR}v'$, then the result is $\texttt{C}(f(v', 2y, \phi(y, n)))$,

5. if v has the form $\texttt{L}v'$, but not $\texttt{LR}v'$, then the result is $\texttt{L}(f(v', 2y, \phi(y, n)))$,

6. if v has the form $\texttt{CR}v''$ or $\texttt{LR}v''$, we can use the equivalences with $\texttt{RL}v''$ and $\texttt{CL}v''$, respectively to switch to one of the previous cases.

5.2.1 Formal details of positive series

We programmed the general treatement of positive series in higher-order function that is easy to re-use from one series to the other.

```
Definition series_body (A:Set)
  (f : stream idigit -> A -> stream idigit)
  (x : stream idigit) (a:A) : stream idigit :=
 let (d,x'') := x in
 match d with
   R =>  R::f x'' a
 | L =>
   match x'' with R::x3 => C::f (L::x3) a | _ => L::f x'' a end
 | C =>
   match x'' with R::x3 => R::f (L::x3) a | _ => C::f x'' a end
  end.
```

The type `A` should contain enough data to recover the values y, and n from our informal presentation.

5.3 Application to computing e

The number e is defined by the formula

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}.$$

Of course, this number is larger than 2, but we only want to compute its fractionary part, so that we actually compute $\sum_{k=2}^{\infty} \frac{1}{k!}$. The following properties are easy to obtain, by rememebering that $n!n^{k-n} < k!$ for every $k \geq n$.

$$0 < \sum_{k=n}^{\infty} \frac{1}{k!} < \frac{1}{(n-1)!(n-1)}.$$

Let $\nu(n)$ be the value $\frac{1}{(n-1)!(n-1)}$. As soon as $2 < n$, we have $\nu(n+1) < \frac{\nu(n)}{2}$. This implies the following property:

$$\forall n, y. 0 < y \wedge 2 < n \wedge \phi(y, n) < \frac{1}{4} \Rightarrow \phi(y, n+1) < \frac{1}{8}.$$

Thus, it is never necessary to absorb more than one term from the infinite sum into x at each co-recursive call.

the type `A` that appears in our use of `series_body` is a triplet, the third component is the factorial of the first component minus one. This way, factorials are not recomputed from scracth at each recursive call. The other components are integer values for y and n in our informal presentation. Here, computing $\phi(y, n)$ is easy, because we know that this value is always n or $n+1$. To know which is the right value, we simply need to compare $\frac{1}{(n-1)!(n-1)}$ with $\frac{1}{8}$ but

we encode this comparison as a plain comparison between integers. When one term from the infinite sum must be absorbed in x, this is performed using the regular addition that we defined in the first part of this paper and a function to construct the infinite digit stream representation of a rational number, which we called `rat_to_stream`.

```
CoFixpoint e_series (v:stream idigit)(s :Z*Z*Z) :stream idigit :=
  let (aux, fact_nm1) := s in let (y, n) := aux in
  let (v', n', fact_nm1') :=
     if Z_le_gt_dec (8*y) (fact_nm1*(n-1))%Z then
        mk_triple v n fact_nm1
     else
        mk_triple (v + (rat_to_stream y (fact_nm1 * n)))
          (n+1) (fact_nm1*n) in
     series_body  _ e_series v' (2*y, n', fact_nm1')%Z.
```

We proved that this co-recursive function provides a correct representation of the infinite sum. The theorem statement is as follows:

```
Theorem e_correct1 :
 forall v vr r n p fact_pm1,
   fact_pm1 = (Z_of_nat (fact (Zabs_nat (p-1)))) ->
   4 * n <= fact_pm1 * (p-1) -> 2 <= p -> 1 <= n ->
   represents v vr ->
   infinit_sum (fun i => (1/INR(fact (i+Zabs_nat p)))%R) r ->
   (vr + (IZR n)*r <= 1)%R ->
   represents (e_series v n p fact_pm1) (vr+(IZR n)*r)%R.
```

This statement is a bit difficult to read, because there are three types of numbers and explicit conversion from one to the other: the factorial function is given as a function from `nat` to `nat` (`nat` is a type of positive numbers, 0 included), `IZR` and `INR` inject positive integers and integers into the type `R` of real numbers, and `Zabs_nat` returns the natural number corresponding to a relative integers (computing an absolute value). In this statement the formula

```
infinit_sum (fun i => (1/INR(fact (i+Zabs_nat p)))%R) r
```

means "`r` is the limit of the infinite sum $\sum_{i=0}^{\infty} \frac{1}{(i+p)!}$". the definition of e is then given by the following commands:

```
Definition head := (rat_to_stream 1 2)+(rat_to_stream 1 6).

Definition number_e_minus2 : stream idigit :=
  e_series head 1 4 6.
```

The correctness theorem then makes it possible to

```
Theorem e_correct :
  forall r,
  infinit_sum (fun i => (1/INR(fact(i+Zabs_nat 2)))) r ->
  represents number_e_minus2 r.
```

This statement really expresses that `number_e_minus2` is a representation of $\sum_{i=2}^{\infty} \frac{1}{i!}$.

We can combine the `bounds` function and the value `number_e_minus2` to build the numerator and denominator of rational that bound e to a given precision. Actually, given n we compute $(a, b, k$ so that

$$\frac{a}{2^k} \leq e - 2 \leq \frac{b}{2^k} \qquad \frac{b}{2^k} - \frac{a}{2^k} = \frac{1}{2^n}.$$

For $n = 320$, this computation takes approximately a minute with the standard version of Coq[2], but in formal verification of hand-made proofs we are not likely to need this level of precision.

The code can also be extracted both to Ocaml and to Haskell. Running the Ocaml extracted code shows that computation is significantly faster than for computation inside the Coq system. In a minute, we can compute 2000 redundants digits of $e - 2$.

5.4 Multiplication as a special case of series

When u is the infinite sequence $d_0 d_1 \ldots$ and if α is the function such that $\alpha(\mathtt{L}) = 0$, $\alpha(\mathtt{C}) = \frac{1}{4}$, and $\alpha(\mathtt{R}) = \frac{1}{2}$, then uu' is a series:

$$uu' = \sum_{i=0}^{\infty} \frac{\alpha(d_i)u'}{2^i}.$$

This is a series where all terms are positive. Moreover, two simplifications can be made with respect to the general approach. First, while y is multiplied by 2 at every recursive call, a_i contains a divisor that is also multiplied by 2, so that the two multiplications by 2 cancel out. Second, it is reasonable to simply consume one element from the infinite sum at each recursive call, with taking into account the value of u'. If this approach is foolowed, the argument y is not necessary anymore: only the digits d_i and u' are needed. We can re-use the general function `series_body` in the following manner:

```
CoFixpoint mult_a (x:stream idigit)(p:stream idigit*stream idigit)
  : stream idigit :=
  let (u,v) := p in
  match u with
    L::u' => series_body _ mult_a x (u',v)
  | C::u' => series_body _ mult_a (x+(L::L::v)) (u',v)
  | R::u' => series_body _ mult_a (x+(L::v)) (u',v)
  end.
```

The function `mult_a` x (u, u') computes $x + uu'$ when $uu' < \frac{1}{4}$. We start by dviding u' by 4 and then we multiply the result by 4. Here is a naive implementation:

[2]Coq version `8.0pl2`, Intel Pentium(R) M 1700Mhz.

```
Definition mult (x y:stream idigit) : stream idigit :=
  mult2(mult2(mult_a zero (x,L::L::y))).

Infix "*" := mult : stream_scope.
```

The following theorem can then proved and verified formally:

```
mult_correct
   : forall (x y : stream idigit) (vx vy : Rdefinitions.R),
     represents x vx -> represents y vy -> represents (x*y)(vx*vy)
```

In this statement the notation `x * y` represents our multiplication as an operation on infinite digit streams, while the notation `vx * vy` represents the multiplication of real numbers, as they are axiomatized in the `Coq` system.

We can also try this multiplication directly inside Coq, for instance, we can compute $(e-2)^2$. With the current standard version of Coq[3] no effort is made to exploit possible sharing in the lazy computation of values, so that the same value may be computed several times. For this reason, we cannot compute this number to a high precision as easily as for $e-2$. For example, our few experiments showed that it takes approximately 10 seconds to compute an approximation with an accuracy of 30 digits and a minute to compute the approximation with an accuracy.

6 Conclusion

The work described in this paper is both an extension and a departure from the work of Ciaffaglione and di Gianantonio. We chose to formalise numbers between 0 and 1. However, a sensible description of real numbers should represent the full real line. Of course, it is possible to obtain a real number by multiplying a number between 0 and 1 by a relative integer, but this naive approach does not preserve the property of redundancy that is necessary for most computations. multiplication by relative integer does not provide the capability to represent an interval around 0. An solution is to consider numbers of the form $a + bx$ where a and b are relative integers.

Other authors take a different approach: instead of adding a digit to represent an interval around $\frac{1}{2}$, they add a digit that should be read as -1. In effect, we still have three overlapping intervals, $[-1, 0]$, $[-\frac{1}{2}, \frac{1}{2}]$, $[0, 1]$. The full line is then represented by multiplying a number between -1 and 1 by a power of 2. This representation is actually closer to useual representation of floating point numbers. It should be easy to redo the experiment described in this paper to adapt it to this new interpretation of digits. In fact, other digit systems, with more or less digits can also be studied. However, there are advantages in manipulating numbers that are positive and in having only to prove that numbers are positive as we did, and we suspect that some proofs may become harder for other digit systems.

[3] Coq version 8.0pl2.

Now, that we have a multiplication for our representation of real numbers, we can consider the task of implementing analytic functions, division. It is also natural to generalize our work to Möbius transformations

$$\frac{axy + bx + cy + d}{exy + fx + gy + h}.$$

However, we suspect that the proofs of correctness for these transformations are less easy to automatize, because they do not rely only on affine formulas (which are easily treated with the Fourier-Motzkin decision procedure.

One of the interesting features of this experiment is that some series can be computed directly inside the theorem prover. This is an important feature, if a proof requires that we produce an accurate approximation of the series value. For instance, we have been able to compute an approximation of e to a few digits in a matter of seconds.

References

[1] A. Bauer, M.H. Escardó, and A. Simpson. Comparing functional paradigms for exact real-number computation. In *Automata, languages and programming*, volume 2380 of *Lecture Notes in Comput. Sci.*, pages 489–500. Springer, 2002.

[2] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications, TLCA 2005*, pages 102–115. Springer-Verlag, 2005.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[4] Hans-Juergen Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic: A case study in higher order programming. In *LISP and Functional Programming*, pages 162–173, 1986.

[5] Sylvie Boldo. *Preuves formelles en arithmétiques àvirgule flottante*. PhD thesis, École Normale Sup'erieure de Lyon, November 2004.

[6] Sylvie Boldo, Marc Daumas, Claire Moreau-Finot, and Laurent Théry. Computer validated proofs of a toolset for adaptable arithmetic. *Journal of the ACM*, 2002. Submitted.

[7] 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.

[8] Thierry Coquand. Infinite objects in Type Theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78. Springer Verlag, 1993.

[9] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, September 2001.

[10] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France)*, volume 1955 of *LNCS/LNAI*, pages 85–95. Springer-Verlag, November 2000.

[11] David Delahaye and Micaela Mayero. Field: une procédure de décision pour les nombres réels en coq. In *Proceedings of JFLA'2001*. INRIA, 2001.

[12] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.

[13] Abbas Edalat and Peter John Potts. A new representation for exact real numbers. In Stephen Brookes and Michael Mislove, editors, *Electronic Notes in Theoretical Computer Science*, volume 6. Elsevier Science Publishers, 1998.

[14] 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.

[15] Ralph W. Gosper. HAKMEM, Item 101 B. `http://www.inwap.com/pdp10/hbaker/hakmem/cf.html#item101b`, feb. 1972. MIT AI Laboratory Memo No.239.

[16] Tanja Grubba, Peter Hertling, Hideki Tsuiki, and Klaus Weihrauch, editors. *CCA 2005 - Second International Conference on Computability and Complexity in Analysis, August 25-29, 2005, Kyoto, Japan*, volume 326-7/2005 of *Informatik Berichte*. FernUniversität Hagen, Germany, 2005.

[17] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. The mpfr library. available at `http://www.mpfr.org`.

[18] John Harrison. Hol light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[19] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[20] John Harrison. Formal verification of IA-64 division algorithms. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[21] IEEE. IEEE standard for binary floating-point arithmetic. *SIGPLAN Notices*, 22(2):9–25, 1987.

[22] Branimir Lambov. Reallib: an efficient implementation of exact real arithmetic. In Grubba et al. [16], pages 169–175.

[23] Valérie Ménissier-Morain. *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse, Université Paris 7, dec 1994.

[24] Valérie Ménissier-Morain. Conception et algorithmique d'une représentation d'arithmétique réelle en précision arbitraire. In *Proceedings of the first conference on real numbers and computers*, 1995.

[25] Jean-Michel Muller. *Elementary Functions, Algorithms and implementation*. Birkhauser, 1997.

[26] Norbert Th. Müller. Implementing exact real numbers efficiently. In Grubba et al. [16], page 378.

[27] Milad Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud University, Nijmegen, September 2004.

[28] Milad Niqui. *Formalising Exact Arithmetic, Representations, Algorithms, and Proofs*. PhD thesis, University of Nijmegen, September 2004. ISBN 90-9018333-7.

[29] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[30] Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[31] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[32] Jean E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, aug 1990.