Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 351 (2006) 39–51
Theoretical Computer Science
www.elsevier.com/locate/tcs

A certified, corecursive implementation of exact real numbers

Alberto Ciaffaglione*, Pietro Di Gianantonio

Dipartimento di Matematica e Informatica, Università di Udine, Italy

Abstract

We implement exact real numbers in the logical framework `Coq` using streams, i.e., infinite sequences, of digits, and characterize constructive real numbers through a minimal axiomatization. We prove that our construction inhabits the axiomatization, working formally with coinductive types and corecursive proofs. Thus we obtain reliable, corecursive algorithms for computing on real numbers.

Keywords: Exact computation; Streams of digits; Lazy functional algorithms; Coinductive type theories; Interactive theorem proving; Program and system verification; Logical Frameworks; Coq

1. Introduction

Computer programs for scientific, numerical applications are largely used in practice, as e.g. in numerical analysis, computational geometry and hybrid systems, but seldom their reliability is addressed formally. A rigorous approach to this kind of software is nowadays crucial in many disciplines, such as mathematics, physics, informatics, engineering, aeronautics, because these employ widely computer systems for performing high-precision numerical data elaboration. Therefore, a first step towards the development of trusted information technology is the availability of *dependable* implementations of the real numbers. Programming languages typically provide *floating-point* real numbers (i.e., approximations of the reals through rational numbers), but these "machine" numbers fail to form a field, already by failing to be closed under the field operations. Computer algebra systems (as `Maple`, or `Mathematica`) represent the reals again via floating-point techniques, or by means of high-precision formal calculi, i.e., formalisms carrying out symbolic calculations. Anyway, it is possible to exhibit counterexamples enlightening the unreliability of such an approach w.r.t. the computational practice.

The crucial point is that both in programming languages and computer algebra systems it is not possible to address *formally* the verification of software. A different technology, providing the possibility of carrying out *mathematical proofs*, is supplied by Logical Frameworks; we are interested in this work in Logical Frameworks based on Type Theory (`LF`s) and their implementations. Typically, `LF`s supply only discrete primitive numerical datatypes, as naturals, integers and possibly rational numbers. The goal of the present work is *to construct* the real numbers in `LF`s. This construction will be done in an *intuitionistic*, i.e., constructive, logic, which is quite well-suited w.r.t. the computational perspective.

* Corresponding author.
E-mail address: ciaffagl@dimi.uniud.it (A. Ciaffaglione).

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.09.061

We intend to adhere to the constructive setting à la Brouwer and Bishop: constructive mathematics is *algorithmic* mathematics.

In order to settle our work we should choose a suitable representation for the reals. Several solutions have been proposed in order to overcome the unreliability of the floating-point practice: interval arithmetic, stochastic arithmetic, multiple-precision arithmetic, exact arithmetic, which we are interested about in the present paper. Exact representation and computation, which originates from seminal ideas of Brouwer and Turing, is gaining growing interest in recent years [19,21]: roughly speaking, it allows to avoid completely the round-off practice, thus permitting to obtain results of desired precision from computations, without having to carry out any independent error analysis. Among the "exact" alternatives for representing the reals, we prefer *digit expansions*, i.e., infinite sequences of positional digits. Hence, our goal is to provide the LFs with an exact implementation of the real numbers, using *streams* (i.e., infinite sequences) of digits, and such that the reals supply a concrete structure suitable both for reasoning and for calculating. To approach streams and exact computation formally in LFs, we need tools for defining and reasoning about infinite objects. The modern, canonical way for dealing with circular, nonwell-founded entities, is to adopt *coinduction*, which is partially supported by the current generation of LFs. Among the several frameworks, the Calculus of Inductive and Coinductive Constructions ($CC^{(Co)Ind}$) [10,9,13] is a type theory providing the user with coinductive definition and proof principles. $CC^{(Co)Ind}$ is implemented in the system Coq [15], the only proof assistant, up to our knowledge, providing the user with native tools for building infinitary derivations. These are carried out via the proof tactic Cofix, which permits to develop infinitely regressive proofs using the thesis as an extra hypothesis, provided its application is guarded by introduction rules [13].

Synopsis: We pick out the proof assistant Coq (V7.2) for experimenting an exact, corecursive implementation of the real numbers via streams of digits. In Section 2 we fix the representation and implement some fundamental functions. Then, in the core Sections 3 and 4, we introduce and justify a constructive axiomatization of the reals, and we use it for addressing the adequacy of the implementation internally in Coq, respectively. Conclusions, related work and directions for future research are in Section 5, while the full Coq development is available at [8]. The present document reports the full picture of our research, whose early steps were documented in [6] and [7].

2. Real numbers as streams of signed digits

In this work we represent real numbers using streams of digits, interpreted as infinite positional expansions, and we work with constructive logic, which is well-suited w.r.t. the computational perspective. It is well known that the standard positional notations *are not* computationally adequate w.r.t. the arithmetic operations. An usual solution for this problem is to adopt *redundant* digit notations, where a real number enjoys more than one representation (typically, infinitely many ones). The following are equivalent possibilities: any integral base with negative digits (Leslie 1817, Cauchy 1840); base $\frac{2}{3}$ with binary digits (Brouwer 1920, Turing 1937); digit notations with an irrational base; infinitely iterated Möbius transformations (Edalat and Potts 1997). We decide to adopt the first approach, i.e., a *signed-digit* notation. Starting from the standard binary notation, we add the negative digit -1 to the digits 0 and 1, and maintain 2 as the value for the base.

Definition 2.1 (*Ternary streams*). Let *str* be the set of the infinite sequences built of ternary digits:

$$str = \{a_1 : a_2 : a_3 : \ldots \mid \forall i \in \mathbb{N}^+.\, a_i \in \{0, 1, -1\}\}.$$

The elements of *str* represent the real numbers in the interval $[-1, 1]$ via the interpretation function $[\![\ ]\!] : str \to \mathbb{R}$, defined by:

$$[\![a_1 : a_2 : a_3 : \ldots]\!] = \sum_{i \in \mathbb{N}^+} a_i \cdot 2^{-i}.$$

To give an example, the number $\frac{1}{2}$ can be represented by the ternary stream 1:0:0:..., or by the stream 0:1:1:..., as well as by the stream $1{:}1{:}-1{:}-1{:}\ldots$. With the above representation we can treat just numbers in the interval $[-1, 1]$. To dispose of arbitrarily large reals, it is necessary to use an exponent-mantissa notation: namely, it is possible to encode real numbers by pairs formed by a natural number and a stream. With this approach, $\langle n, x\rangle$ represents the

number $2^n \cdot [\![x]\!]$. For lack of space, we treat in this paper only the stream representation of the numbers $[-1, 1]$. This is the most interesting part of the work: in fact, all notions and properties can be then straightforwardly extended to the exponent-mantissa representation, see [5] for details.

In order to complete our construction it is necessary to provide an *order* relation and a *field* structure: actually, the real line is completely determined by the binary *strict order* relation ($<$) and the arithmetic operations of *addition* and *multiplication*. We considered several different possible characterizations for order, addition and multiplication along our research: in the end, we chose to describe not only the order, but also the operations using relations and not functions. This choice is due to the fact that relations are simpler to specify and work with. An intuitive motivation is that functions are requested to be "productive"—i.e., they must supply a method to effectively produce the result, given the input; on the contrary, a relation just specifies the constraints that the output has to satisfy w.r.t. the input. Thus it is a simpler task to prove the formal properties of the relations. Anyway, we will introduce the functions as well and we will prove they are coherent w.r.t. the relations. One can interpret this fact saying that the implementation (described by algorithms, i.e., functions) satisfies the specification (described by relations).

We have devised that the length and the complexity of the formal proofs about the relations are greatly affected by the pattern of their specifications: very often the proofs are carried out by structural (co)induction on the derivations, thus the number of cases to consider grows together with the number of constructors of the relation involved. In order to simplify the proofs, we have formalized the (co)inductive relations using at most two constructors, thus reducing the cases to address.

Relations: The strict order relation ($<$) is defined by *induction*: this is possible because, given two streams, we can semi-decide whether the first is smaller than the second just by examining a finite number of digits. The binary strict order relation on streams is defined in terms of an auxiliary ternary relation $less\_aux \subseteq (str \times str \times Z)$, whose intended meaning is: $less\_aux(x, y, i) \Leftrightarrow ([\![x]\!] < [\![y]\!] + i)$.

This auxiliary relation permits to simplify the management of the order: the use of the integer parameter i allows to obtain simpler proofs, because the extensive case analysis on the ternary digits is replaced by automated proofs over integers. The main binary relation on streams $less \subseteq (str \times str)$ is defined fixing the value of the integer parameter to 0.

Definition 2.2 (*Order*). The relation $less\_aux \subseteq (str \times str \times Z)$ is defined by induction:

$$(\textit{less-base})\ \frac{big \leqslant i}{less\_aux(x, y, i)} \quad \text{where } big > 2$$

$$(\textit{less-ind})\ \frac{less\_aux(x,\ y,\ (2i + b - a))}{less\_aux(a : x,\ b : y,\ i)}.$$

The strict order relation on streams, $less \subseteq (str \times str)$, is defined by: $less(x, y) \triangleq less\_aux(x, y, 0)$.

This definition, parametric on the value *big*, requires some additional explanations. It is easy to see, referring to the intended meaning, that $less\_aux(x, y, i)$ is valid for any value of the parameter i greater than 2: a natural choice for the constant *big* would be the integer 3, but it turns out that any greater value gives rise to an equivalent definition. Moreover, we have experimented that greater values simplify several proofs built by structural induction on the judgment *less_aux*: in fact, the (*less-base*) rule has a stronger premise as the value of *big* grows, and so it provides a stronger inductive hypothesis. Up to our experience there is no canonical choice for *big*; however we state that the value 32 is sufficiently large to simplify all the proofs we need to construct and that larger values do not give rise to any extra simplification. So we propose 32 as convenient choice for the constant *big*. It is immediate to see that the base rule is sound. The induction rule can be informally justified by means of a simple calculation: $[\![a : x]\!] < ([\![b : y]\!] + i) \Leftrightarrow a/2 + ([\![x]\!]/2) < b/2 + ([\![y]\!]/2) + i \Leftrightarrow [\![x]\!] < [\![y]\!] + 2i + b - a$.

Differently from the order, the arithmetic relations can be naturally defined by *coinduction*, because the process of adding and multiplying two real numbers is in general not terminating. Coinductive relations give rise to assertions that have to be proved by an infinite application of the corresponding constructors [9,13]. The relations of addition and multiplication share the following pattern: $relation(operand_1,\ operand_2,\ result)$. We start from addition: as done for the order relation, we define first an auxiliary relation on streams. The relation $add\_aux \subseteq (str \times str \times str \times Z)$ has intended meaning: $add\_aux(x, y, z, i) \Leftrightarrow ([\![x]\!] + [\![y]\!]) = ([\![z]\!] + i)$.

Definition 2.3 (*Addition*). The relation $add\_aux \subseteq (str \times str \times str \times Z)$ is defined by coinduction:

$$(add\text{-}coind)\ \frac{add\_aux(x,\ y,\ z,\ (2i + c - a - b)) \quad (-big < i < big)}{add\_aux(a : x,\ b : y,\ c : z,\ i)}.$$

The addition relations on streams, $add \subseteq (str \times str \times str)$, is defined by: $add(x, y, z) \triangleq add\_aux(x, y, z, 0)$.

Note that there is no base rule proving in one step a statement in the form $add\_aux(x, y, z, i)$. A proof of $add\_aux(x, y, z, i)$ has to be an infinite object obtained by applying infinitely the constructor (add-coind). Coinductive predicates are, for many aspects, similar to lazy data types (such as streams themselves), whose elements are obtained by applying infinitely the concatenation constructor. The side-condition $(-big < i < big)$ has been introduced in order to make the relation *add_aux* not total—otherwise, given any 4-tuple x, y, z, i, there would exists an instance of the add-coind rule deriving $add\_aux(x, y, z, i)$, and therefore it would be possible to construct an infinitary proof of $add\_aux(x, y, z, i)$ for any x, y, z, i. Similarly to the order, values of *big* greater than 3 give rise to equivalent definitions, but larger values lead to simpler proofs. The coinductive rule (*add-coind*) can be informally justified by the calculation: $([\![a : x]\!] + [\![b : y]\!]) = ([\![c : z]\!] + i) \Leftrightarrow a/2 + ([\![x]\!]/2) + b/2 + ([\![y]\!]/2) = c/2 + ([\![z]\!]/2) + i \Leftrightarrow [\![x]\!] + [\![y]\!] = [\![z]\!] + 2i + c - a - b$.

As far as the multiplication is concerned, we define a preliminary multiplication *function* between signed digits and streams $times_{\mathrm{d,str}} : \{0, -1, 1\} \times str \to str$, with the obvious behaviour: $[\![times_{\mathrm{d,str}}(a, x)]\!] = a \cdot [\![x]\!]$. As usual, the multiplication is reduced to a series of additions; in this way, we can define directly a ternary multiplication relation on streams (*mult*).

Definition 2.4 (*Multiplication*). The function $times_{\mathrm{d,str}} : \{-1, 0, 1\} \times str \to str$ is defined by corecursion:

$$times_{\mathrm{d,str}}(a, (b : x)) \triangleq (a \cdot b) : (times_{\mathrm{d,str}}(a, x)).$$

The multiplication relation on streams $mult \subseteq (str \times str \times str)$ is defined by coinduction:

$$(mult\text{-}coind)\ \frac{mult(x, y, w) \quad add(0 : times_{\mathrm{d,str}}(a, y),\ 0 : w,\ z)}{mult(a : x,\ y,\ z)}.$$

The coinductive rule (*mult-coind*) can be informally justified by: $([\![a : x]\!] \cdot [\![y]\!]) = [\![z]\!] \Leftrightarrow (a/2 + [\![x]\!]/2) \cdot [\![y]\!] = [\![z]\!] \Leftrightarrow (a \cdot [\![y]\!])/2 + ([\![x]\!] \cdot [\![y]\!])/2 = [\![z]\!]$.

Exact arithmetic algorithms: In the following we define the main functions on reals; then, in Section 4, we will prove that they are coherent with respect to the specifications defined by relations above.

Addition: The addition of streams is defined via an auxiliary function $+_{\mathrm{aux}} : (str \times str \times [-2, 2]_Z) \to str$, where we denote with $[-2, 2]_Z$ the set of integers $\{-2, -1, 0, 1, 2\}$. Using $+_{\mathrm{aux}}$, one can easily define the function on streams $(+_{\mathrm{str}})$. The behaviours are: $[\![+_{\mathrm{aux}}(x, y, i)]\!] = ([\![x]\!] + [\![y]\!] + i)/4$ and $[\![+_{\mathrm{str}}(x, y)]\!] = ([\![x]\!] + [\![y]\!])/2$. Notice that, since a single stream can represent only the real numbers in the interval $[-1, 1]$, the result of the addition between streams has to be normalized (divided) by a factor 2. We remark that our algorithm has linear complexity, as it just examines one digit of the (stream) arguments to generate a digit of output.

Definition 2.5 (*Addition function*). The function $+_{\mathrm{aux}} : (str \times str \times Z) \to str$ is defined by corecursion:

$$\begin{aligned} +_{\mathrm{aux}}(a : x_0,\ b : y_0,\ i) \triangleq\ & \texttt{let } j := (2i + a + b) \texttt{ in} \\ & \texttt{Cases } j \texttt{ of} \\ & \quad j \geqslant 2 \Rightarrow (1 : +_{\mathrm{aux}}(x_0,\ y_0,\ j - 4)) \\ & \quad j \in [-1, 1] \Rightarrow (0 : +_{\mathrm{aux}}(x_0,\ y_0,\ j)) \\ & \quad j \leqslant -2 \Rightarrow (-1 : +_{\mathrm{aux}}(x_0,\ y_0,\ j + 4)) \end{aligned}$$

The addition function on streams, $+str : (str \times str) \to str$, is defined by: $+_{\mathrm{str}}(a : x_0,\ b : y_0) \triangleq +_{\mathrm{aux}}(x_0,\ y_0,\ a + b)$.

Multiplication: The multiplication algorithm is defined in terms of the addition one. Also for multiplication it is convenient to use the auxiliary functions $\times_{\mathrm{aux}} : (str \times str \times str \times [-2, 2]_Z) \to str$ and $\times_4 : str \to str$, with behaviour: $[\![\times_{\mathrm{aux}}(x, y, z, i)]\!] = (([\![x]\!] \cdot [\![y]\!]) + [\![z]\!] + i)/4$, and $[\![\times_4(x)]\!] = [\![x]\!] \cdot 4$, in the case that $[\![x]\!]$ is contained in the

interval $[-\frac{1}{4}, \frac{1}{4}]$. For lack of space we do not specify the (trivially definable) $\times_4$ function, whose formal definition can be found in [5]. Note that our multiplication algorithm has quadratic complexity on the number of generated digits.

Definition 2.6 (*Multiplication function*). The function $\times_{\text{aux}} : (str \times str \times str \times [-2, 2]) \to str$ is defined by corecursion:

$$\begin{aligned}\times_{\text{aux}}(a : x_0,\ y,\ c : z_0,\ i) \triangleq\ & \texttt{let}\ (d : e : w) := +_{\text{aux}}(\times_{\text{d,str}}(a, y),\ z_0,\ i)\ \texttt{in}\\ & \texttt{let}\ j := (2d + e + c + i)\ \texttt{in}\\ & \texttt{Cases}\ j\ \texttt{of}\\ & \quad j \geqslant 3\ \Rightarrow\ (1 : \times_{\text{aux}}(x_0,\ y,\ w,\ j - 4))\\ & \quad j \in [-2, 2]\ \Rightarrow\ (0 : \times_{\text{aux}}(x_0,\ y,\ w,\ j))\\ & \quad j \leqslant -3\ \Rightarrow\ (-1 : \times_{\text{aux}}(x_0,\ y,\ w,\ j + 4))\end{aligned}$$

The multiplication function on streams, $\times_{\text{str}} : (str \times str) \to str$, is defined by: $\times_{\text{str}}(x, y) \triangleq \times_4 (\times_{\text{aux}}(x, y, \overline{0}, 0))$.

An informal proof of correctness for $\times_{\text{aux}}$ must consider all the possible cases for the test on j. The first case is justified by the following chain of equalities:

$$\begin{aligned}&([\![a : x_0]\!] \cdot [\![y]\!] + [\![c : z_0]\!] + i)/4\\ &\quad = ((a + [\![x_0]\!]) \cdot [\![y]\!] + c + [\![z_0]\!] + 2i)/8\\ &\quad = ([\![x_0]\!] \cdot [\![y]\!] + ([\![z_0]\!] + a \cdot [\![y]\!] + i) + i + c)/8\\ &\quad = ([\![x_0]\!] \cdot [\![y]\!] + ([\![d : e : w]\!] \cdot 4) + i + c)/8\\ &\quad = [\![x_0]\!] \cdot [\![y]\!] + [\![w]\!] + (2d + e + i + c))/8\\ &\quad = \tfrac{1}{2} + \tfrac{1}{2} \cdot ([\![x_0]\!] \cdot [\![y]\!] + [\![w]\!] + j - 4)/4.\end{aligned}$$

The other cases can be treated similarly. To complete the proof of correctness, one should also prove that in each recursive call the integer argument i is always contained in the interval $[-2, 2]$, which is carried out by case analysis.

Negation and reciprocal: The negation of a stream can be trivially defined by negating, one by one, the single digits of the stream. More complex is the definition of the reciprocal. For evaluating the reciprocal, we have to introduce the division function on streams div, and, in turn, an auxiliary function $test : (str \times str) \to Z$. The expression $test(x, y)$ returns an integer approximant of the value $(2^5 \cdot [\![x]\!] - 2^3 \cdot [\![y]\!])$, the expression $div(x, y)$ returns the value $[\![x]\!]/[\![y]\!]$ on the hypotheses that $[\![x]\!]/[\![y]\!]$ is representable (i.e., it belongs to the interval $[-1, 1]$) and $[\![y]\!]$ belongs to the interval $[\frac{1}{4}, 1]$.

Definition 2.7 (*Division function*). The test function on streams $test : (str \times str) \to Z$ is defined by:
$test(a : b : c : d : e : x_4,\ f : g : h : y_2) \triangleq 16a + 8b + 4(c - f) + 2(d - g) + (e - h)$.
The division function on streams $div : str \to str$ is defined by corecursion:

$$\begin{aligned}div(x,\ y) \triangleq\ & \texttt{let}\ x := (a : b : c : x_2),\ j := (4a + 2b + c),\ i := test(x,\ y)\ \texttt{in}\\ & \texttt{if}\ (j \geqslant 0)\quad (*\ [\![x]\!] \geqslant -1/8\ *)\\ & \quad \texttt{then if}\ (i \geqslant 0)\quad (*\ 4[\![x]\!] - [\![y]\!] \geqslant -1/4\ *)\\ & \quad\quad \texttt{then}\ 1 : div(\times_4(+_{\text{str}}(x, -(0 : y))),\ y)\\ & \quad\quad \texttt{else}\ 0 : div(\times_4(+_{\text{str}}(0 : x, 0 : x)),\ y)\\ & \quad \texttt{else if}\ (i \geqslant 0)\quad (*\ 4[\![x]\!] - [\![y]\!] \geqslant -1/4\ *)\\ & \quad\quad \texttt{then}\ 0 : div(\times_4(+_{\text{str}}(0 : x, 0 : x)),\ y)\\ & \quad\quad \texttt{else}\ -1 : div(\times_4(+_{\text{str}}(x, 0 : y)),\ y)\end{aligned}$$

Limit: To complete our construction we need to define a limit function that, taken as input a Cauchy sequence with an exponential convergence rate, returns its limit. To this end, we employ a normalization function $norm : (digit \times str) \to str$. The expression $norm(a, x)$ returns a stream such that, whenever possible, the following equality holds: $[\![x]\!] = [\![a : norm(a, x)]\!]$ (its actual definition can be found in [5]). Given a sequence $\langle x_n \rangle_{n \in \mathbb{N}}$ having an exponential convergence rate: $\forall n.\ |[\![x_n]\!] - [\![x_{n+1}]\!]| \leqslant 2^{-(n+4)}$, its limit is constructed by generating the first digit looking at the first three digits of x_0, and applying corecursively the method to the subsequence $\langle x_{i+1} \rangle_{i \in \mathbb{N}}$, point-wise modified by the function $norm$.

Definition 2.8 (*Limit function*). The limit function on stream sequences $lim : (\mathbb{N} \to str) \to str$ is defined by corecursion:

$$\begin{aligned} lim(\langle x_n\rangle_{n\in\mathbb{N}}) \triangleq\ & \texttt{let}\ x_0 := (a : b : c : y),\ j := (4a + 2b + c)\ \texttt{in} \\ & \texttt{Cases}\ j\ \texttt{of} \\ & \quad j \in [3, 7] \Rightarrow 1 : (lim(\lambda n.\ (norm(1,\ x_{n+1})))) \\ & \quad j \in [-2, 2] \Rightarrow 0 : (lim(\lambda n.\ (norm(0,\ x_{n+1})))) \\ & \quad j \in [-7, -3] \Rightarrow -1 : (lim(\lambda n.\ (norm(-1,\ x_{n+1})))) \end{aligned}$$

Equivalence: In constructive analysis, it is possible to describe the equivalence relation on real numbers by means of the order relation: $equal_{\text{ind}}(x, y) \triangleq \neg less(x, y) \wedge \neg less(y, x)$. It is interesting to notice that the equivalence can also be defined directly via a coinductive predicate. In this case, it is convenient to introduce first an auxiliary relation $equal\_aux \subseteq (str \times str \times Z)$, which has intended meaning: $equal\_aux(x, y, i) \Leftrightarrow (\llbracket x \rrbracket = \llbracket y \rrbracket + i)$

$$(equal\text{-}coind)\ \frac{equal\_aux(x,\ y,\ (2i + b - a)) \quad (-big < i < big)}{equal\_aux(a : x,\ b : y,\ i)}$$

Then, the equivalence relation on streams $equal \subseteq (str \times str)$ is defined by: $equal(x, y) \triangleq equal\_aux(x, y, 0)$. In our formalization of the real numbers we have used both definitions, and the proof that they coincide is in [5].

Adequacy: There are two main approaches that can be used for justifying the construction presented above: the first one can be called external-semantic, while a second one is internal-axiomatic. According to the semantic approach, we justify the predicates of order, addition and multiplication by proving that their specification is sound and complete with respect to an external model of the reals $\mathbb{R}$. A proof of correctness along these lines can be found in [6]. Following the second approach, we first present an axiomatization of the constructive real numbers, then we prove that our implementation provides a model for these axioms. These are the subjects of Sections 3 and 4.

3. A constructive axiomatization

We fix here a characterization of the constructive real numbers through a novel, minimal axiomatization. In the literature there are two alternative, equivalent axiomatizations. One is proposed by Bridges [2], and a second one by the FTA group [12,11]. A comparison between our axiomatization and the latter can be found in [7]; here we just remark that the main objective of our approach is to have a minimal set of axioms to be formalized in the logical framework we use.

In order to state our axioms we should dispose of a logical system that accommodates the second-order quantification—to axiomatize the completeness—and the Axiom of Choice—for defining the "reciprocal" function on reals different from zero. The proof assistant `Coq` provides such a logical system.

Sets, *functions*, *predicates and axioms*: We postulate the constructive real numbers as the mathematical objects satisfying four groups of axioms. The basic notions are the following:

- a representation set R, with two elements 0_R (zero) and 1_R (one);
- a binary relation $<$ (strict order) over R;
- two binary operations $+$ (addition) and $\times$ (multiplication) over R.

It is then convenient to introduce two relations and two functions:

- a binary relation $\sim$ (equivalence) over R tells that two different elements represent the same number, thus capturing the redundancy of the representation;
- two functions $inj : \mathbb{N} \to R$ ($inj(n) = n$) and $exp : \mathbb{N} \to \mathbb{N}$ ($exp(n) = 2^n$) are used in the archimedeanity and completeness axioms;
- a ternary relation $near \subseteq R \times R \times \mathbb{N}$ ($near(x, y, n) \Leftrightarrow |x - y| \leqslant 2^{-n}$) describes the Euclidean metric.

Axioms: As standard in constructive approaches to analysis [20], the set of real numbers is defined as the quotient of a set of representations. Our axiomatization is parametric with respect to the set $\mathbb{N}$ of the natural numbers, that we

suppose to be given. In our formalization in Coq, $\mathbb{N}$ is taken as the set of the *inductive* natural numbers. Finally, we claim that constructive real numbers are captured by the following axiomatization.

Definition 3.1 (*Axioms for constructive real numbers*).

$$\textit{Constants} : R,\ \{0_R, 1_R\} \in R,\ < \subseteq R \times R,\ + : R \times R \to R,\ \times : R \times R \to R$$

Defs :	$\sim\ \subseteq R \times R$	$(x \sim y) \triangleq \neg(x < y) \wedge \neg(y < x)$
	$inj : \mathbb{N} \to R$	$inj(0) \triangleq 0_R,\ inj(n+1) \triangleq inj(n) + 1_R$
	$exp : \mathbb{N} \to \mathbb{N}$	$exp(0) \triangleq 1,\ exp(n+1) \triangleq exp(n) \cdot 2$
	$near \subseteq R \times R \times \mathbb{N}$	$near(x, y, n) \triangleq \forall \varepsilon \in R.\ (1_R < \varepsilon \times inj(exp(n))) \Rightarrow (x < y + \varepsilon) \wedge (y < x + \varepsilon)$
Axioms :	$+$-*associativity*	$\forall x, y, z \in R.\ (x + (y + z)) \sim ((x + y) + z)$
	$+$-*unit*	$\forall x \in R.\ (x + 0_R) \sim x$
	negation	$\forall x \in R.\ \exists y \in R.\ (x + y) \sim 0_R$
	$+$-*commutativity*	$\forall x, y \in R.\ (x + y) \sim (y + x)$
	$\times$-*associativity*	$\forall x, y, z \in R.\ (x \times (y \times z)) \sim ((x \times y) \times z)$
	$\times$-*unit*	$\forall x \in R.\ (x \times 1_R) \sim x$
	reciprocal	$\forall x \in R.\ (0_R < x) \Rightarrow \exists y \in R.\ (x \times y) \sim 1_R$
	$\times$-*commutativity*	$\forall x, y \in R.\ (x \times y) \sim (y \times x)$
	distributivity	$\forall x, y, z \in R.\ (x \times (y + z)) \sim (x \times y) + (x \times z)$
	nontriviality	$0_R < 1_R$
	$<$-*asymmetry*	$\forall x, y \in R.\ (x < y) \Rightarrow \neg(y < x)$
	$<$-*co-transitivity*	$\forall x, y, z \in R.\ (x < y) \Rightarrow (x < z) \vee (z < y)$
	$+$-*reflects-* $<$	$\forall x, y, z \in R.\ (x + z < y + z) \Rightarrow (x < y)$
	$\times$-*reflects-* $<$	$\forall x, y \in R.\ (x \times z < y \times z) \Rightarrow (x < y) \vee ((y < x) \wedge (z < 0_R))$
	archimedeanity	$\forall x \in R.\ \exists n \in \mathbb{N}.\ x < inj(n)$
	completeness	$\forall f : \mathbb{N} \Rightarrow R.\ \exists x \in R.$ $(\forall n \in \mathbb{N}.\ near(f(n), f(n+1), n+1)) \Rightarrow (\forall m \in \mathbb{N}.\ near(f(m), x, m))$

Arithmetic operations: As the reader can see, the properties required for the arithmetic operations are just those characterizing a classical Abelian field: in [2], this set of properties is named "Heyting field". Notice, however, a slight simplification: it is sufficient to assume the existence of the reciprocal only for positive reals. We do not assume the existence of the "negation" $(-)$ and "reciprocal" $(^{-1})$ functions. The main reason for this choice is that the reciprocal function cannot be defined in Coq, where functions have to be totally specified. Moreover, in a constructive setting, functions have to be continuous w.r.t. the Euclidean topology; however, it is not possible to make continuous by extension the reciprocal function. Thus we assume the existence, for each real x, of its negation, and, if $0 < x$, its reciprocal *elements*. In this way we must postulate the axiom of choice for extracting effectively the negation and the reciprocal of a number x. The necessity of the axiom of choice can be seen as a weakness of the axiomatization; however, there is no simple way to avoid it: in fact, without choice, the reciprocal function could not be defined in Coq (whereas the negation function and the limit functional can be defined).

Order relation: First notice that the classical Law of Trichotomy $(x < y) \vee (x = y) \vee (y < x)$ fails to be a constructive property [2]: its substitute, in the constructive setting, is the property $(x < y) \Rightarrow (x < z) \vee (z < y)$, named $<$-*co-transitivity*.

We remark that it is sufficient to define only the relation of order, because in constructive mathematics the order is universally considered the most fundamental relation for the real numbers. In our approach, in fact, the equivalence is a derived notion. We are able to derive all the basic properties relating the equivalence to the operations from the two reflection axioms: $+$-reflects-$<$ and $\times$-reflects-$<$. The fact that the equivalence is preserved by the basic notions (order, addition and multiplication), is an immediate corollary of the two reflection axioms and the $<$-*co-transitivity* one.

Archimedeanity and completeness: The Archimedean axiom links the real numbers to the natural numbers, stating that reals are standard with respect to naturals. The completeness property is postulated asking for the existence of the limit for any Cauchy sequence $\langle s_n \rangle_{n\in\mathbb{N}}$ with an exponential convergence rate ($\forall n \in \mathbb{N}.\ |s_n - s_{n+1}| \leqslant 2^{-(n+1)}$). Many alternative choices for capturing the completeness might be stated, and our axiom could appear weak at a first glance. Indeed, it is necessary to know the convergence rate of a Cauchy sequence S in order to evaluate constructively its limit: from such a convergence rate, it is then possible to extract (constructively) a subsequence of S having an exponential convergence rate. Therefore, starting from our axiom, we are able to derive the alternative completeness properties found in the literature [2,11]. Our choice is motivated by simplicity reasons.

Axioms at work: Most of the elementary mathematical theory can be easily derived from our axiomatization. Namely, it is possible to prove that the order is transitive, that the operations of addition and multiplication preserve, and reflect, the relations of order and equivalence. Such a development of the basic arithmetic theory for the constructive reals has been formally carried out using the proof assistant `Coq` [15], and has been presented in [7].

4. Consistency

In this Section we document the certification of the implementation presented in Section 2, using the axiomatization 3.1: first we discuss the formalization in `Coq`, then we present the development of the formal proofs. For lack of space, we address only the axioms at the level of streams; this is the most interesting part of the work, as all proofs can be straightforwardly extended to the exponent-mantissa representation, see [5] for details.

Formalization in Coq: We represent signed-digits and streams in the specification language of $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ through concrete sets:

```
Inductive digit: Set :=  mino: digit | zero: digit | one: digit.
CoInductive str: Set :=  cons: digit -> str -> str.
```

The specification of order, addition and multiplication predicates requires to introduce the function `code`, which maps the constructors of digits into corresponding built-in integer values, thus allowing to automate integer calculations. The encoding of the main predicates *less*, *add* and *mult* is carried out formalizing the specifications of Section 2. We report below just the "auxiliary" level code for addition:

```
CoInductive add_aux: str -> str -> str -> Z -> Prop :=
add_coind: (x,y,z:str) (a,b,c:treat) (i:Z)
           (`-big < i`) -> (`i < big`) ->
           (add_aux x y z `2*i-(code a)-(code b)+(code c)`) ->
           (add_aux (cons a x) (cons b y) (cons c z) i).
```

Then we encode the exact algorithms. The construction of streams is carried out through corecursive functions, which allow to build terms inhabiting coinductive sets (streams, in the case) and can have arbitrary domains. A corecursive function is checked by `Coq` and accepted if and only if the recursive call is *guarded by constructors* [9,13]: the guardedness condition guarantees the *strong normalization* property in the logical framework. Circular, nonwell-founded terms, such as streams, must be constructed *lazily*, i.e., they can be expanded just when they occur as arguments of a case-analysis construct. We formalize the addition of streams as follows (the remaining definitions are similar and not problematic: see [8] for the code):

```
CoFixpoint r_plus_aux: str -> str -> Z -> str :=  [x,y:str; i:Z]
Cases x of (cons a x0) => Cases y of (cons b y0) =>
let j = `2*i + (code a) + (code b)` in Cases `j + 1`
of (NEG _) => (cons mino (r_plus_aux x0 y0 `j + 4` ))
|  (ZERO)  => (cons zero (r_plus_aux x0 y0  j))
|  (POS _) => Cases `j - 1`
   of (POS _) => (cons one  (r_plus_aux x0 y0 `j - 4`))
   |  _  => (cons zero (r_plus_aux x0 y0  j)) end end end end.
```

We detail a full example about a corecursive proof in Coq. We assume the corecursive functions odd, even (taken an input stream, they return, respectively, streams built by the elements in odd and even positions), merge (given two streams, it renders the stream built taking elements alternatively in the arguments), and the coinductive point-wise equality on streams:

```
CoInductive Eq_str: str->str->Prop :=  eq_c: (x,y:str) (a:digit)
            (Eq_str x y) -> (Eq_str (cons a x) (cons a y)).
```

The system Coq mechanizes the *guarded induction principle* of Coquand and Giménez [9,13], which is associated to coinductive *predicates*. This is a proof schema for carrying out infinitely regressive proofs: it permits to use the thesis as an auxiliary hypothesis, provided it is applied within introduction rules; this principle is implemented by means of the tactic Cofix. We prove, by guarded induction, through Cofix, that every stream is point-wise equal to its transformation through the combination of the above functions, namely that $\forall x \in str.\ \mathtt{Eq\_str}(\mathtt{merge}(\mathtt{odd}(x), \mathtt{even}(x)), x)$. First we assume the thesis among the hypotheses, then we destruct the argument-stream x, expand the definitions, consume input digits through the constructor eq_c, and conclude applying the coinduction hypothesis, as follows ($\cong$ stands for Eq_str):

$$\begin{array}{rcl}
\mathtt{merge}(\mathtt{odd}(x), \mathtt{even}(x)) & \cong & (x) \\
\mathtt{merge}(\mathtt{odd}(a:b:z), \mathtt{even}(a:b:z)) & \cong & (a:b:z) \\
\mathtt{merge}(a:\mathtt{odd}(z), b:\mathtt{even}(z)) & \cong & (a:b:z) \\
(a:\mathtt{merge}(b:\mathtt{even}(z), \mathtt{odd}(z))) & \cong & (a:b:z) \\
(a:b:\mathtt{merge}(\mathtt{odd}(z), \mathtt{even}(z))) & \cong & (a:b:z) \\
\mathtt{merge}(\mathtt{odd}(z), \mathtt{even}(z)) & \cong & (z) \\
 & Q.E.D &
\end{array}$$

Certification of the exact algorithms: We prove in Coq that the implementation of the real numbers through streams, introduced in Section 2, is a model for the axiomatization given in Definition 3.1. The importance of this result is twofold: we address the internal adequacy of our construction of the reals, and we show that the axioms are consistent. As previously explained, it is convenient to prove first the coherence between the exact, corecursive algorithms and the arithmetic predicates, and then to show that the axioms are inhabited by carrying out proofs about the predicates.

From now on we will denote with the symbol "$\sim$" the equivalence between streams introduced in Section 2 (remember that the coinductive equivalence *equal* and the inductive one $equal_{\mathrm{ind}}$ coincide). Hence we state and prove that the results computed by the arithmetic functions are admissible for the predicates, and predicates are well-defined with respect to the equivalence. These are key properties for proving, in turn, that the axiomatization 3.1 is inhabited by our coinductive model of the reals. All the properties we list below are proved formally in the Coq system [8].

Proposition 4.1 (*Addition functions and predicates*).

(i) $\forall x, y \in str, \quad \forall a, b, c, d \in \{0, 1, -1\}, \ \forall i, j \in Z.$
$(2a + b + 2c + d = 4i + j) \wedge (-2 \leqslant j \leqslant 2) \Rightarrow add\_aux(a : b : x,\ c : d : y,\ +_{\mathrm{aux}}(x, y, j),\ i);$
(ii) $\forall x, y \in str.\ add(0 : x,\ 0 : y,\ +_{\mathrm{str}}(x, y));$
(iii) $\forall x, y, z, w \in str.\ add(x, y, z) \wedge add(x, y, w) \Rightarrow (z \sim w).$

Proof. (i) By coinduction. (ii) Corollary of the point (i): it suffices to consider $x = a : x_0$ and $y = b : y_0$, so $add(0 : x, 0 : y, +_{\mathrm{str}}(x, y)) = add\_aux(0 : a : x_0, 0 : b : y_0, +_{\mathrm{aux}}(x_0, y_0, a + b), 0)$. (iii) By coinduction. □

Proposition 4.2 (*Multiplication functions and predicates*).

(i) $\forall x \in str,\ \forall i \in Z.\ less\_aux(norm(i, x),\ x,\ i) \Rightarrow less\_aux(\overline{1},\ x,\ i);$
(ii) $\forall x \in str,\ \forall i \in Z.\ less\_aux(x,\ norm(-i, x),\ i) \Rightarrow less\_aux(x,\ \overline{-1},\ i);$
(iii) $\forall x \in str.\ \neg less(x, 0 : 0 : \overline{-1}) \wedge \neg less(0 : 0 : \overline{1}, x) \Rightarrow \neg less(x, 0 : 0 : \times_4(x)) \wedge \neg less(0 : 0 : \times_4(x), x);$
(iv) $\forall x, y, z, w \in str.\ add(x, y, z) \wedge \neg less(z, w) \wedge \neg less(w, z) \Rightarrow add(x, y, w);$
(v) $\forall x, y, v, w, z \in str,\ \forall d, e \in \{0, 1, -1\},\ \forall i, j, k, l \in Z.$
$(-2 \leqslant i, j \leqslant 2) \wedge (k + i + 2d + e = 4l + j) \wedge add\_aux(v, w, z, k) \Rightarrow$
$add\_aux(\times_{\mathrm{aux}}(x, y, v, i),\ d : e : w,\ \times_{\mathrm{aux}}(x, y, z, j),\ l);$

(vi) $\forall x, y \in str.\ mult(x,\ y,\ \times_{str}(x, y))$;
(vii) $\forall x, y, z, w \in str.\ mult(x, y, z) \wedge mult(x, y, w) \Rightarrow (z \sim w)$.

Proof. (i) By structural induction on $less\_aux(norm(i, x),\ x,\ i)$. (ii) By structural induction on $less\_aux\ (x,\ norm\ (-i, x),\ i)$. (iii) By points (i) and (ii). (iv) Exploiting the equivalence between *equiv* and $equiv_{\text{ind}}$, and by coinduction. (v) By Lemmas 4.1(i), 4.3(v) and 4.3(vi). (vi) By points (iii), (iv), (v) and Lemma 4.3(vi). (vii) By coinduction. □

The two propositions can also be seen as the formal proof about the reliability of algorithms performing exact real number computation. Thus, `Coq` code implementing addition and multiplication is *certified*, and `Haskell` running code can be immediately obtained from the system.

Certification of the implementation: Most of the proofs of the Axiomatization 3.1 follow a similar pattern: first we prove the basic facts about the auxiliary predicates (*less_aux*, *add_aux*), then we deduce the corresponding results about streams (*less*, *add*, *mult*). The main difficulty is to address the "auxiliary" level: the two tactics we have mainly used are `Cofix` and `Omega`.

The tactic `Cofix`, as discussed above in the Section, is a built-in tool for proving coinductive assertions. The tactic `Omega` proves automatically assertions in Presburger's arithmetic, and is very useful to avoid repeated case analysis on the values of the ternary digits. The use of this tactic and the introduction of the auxiliary predicates allow for a great simplification of the proofs: almost all the propositions are proved invoking at most 50 strategies. We detail below the proofs for the different families of axioms; it is worth noticing that the presentation is quite technical.

Order and addition: A preparatory lemma about the predicate *less_aux* is used, in turn, for deducing those properties that play the role of the axioms at the level of streams. The treatment of addition is similar to the order.

Lemma 4.1 (*Order-auxiliary*). *Let be* $x, y \in str, i, j, k \in Z, big = 32$:
(i) $less\_aux(x, y, i) \Rightarrow (-1 \leqslant i)$;
(ii) $less\_aux(x, y, i) \wedge (i \leqslant j) \Rightarrow less\_aux(x, y, j)$;
(iii) $(2 < big - i) \Rightarrow less\_aux(x, y, big - i)$;
(iv) $(2 < i) \Rightarrow less\_aux(x, y, i)$;
(v) $less\_aux(x, y, i) \wedge less\_aux(y, x, j) \Rightarrow (0 < i + j)$;
(vi) $less\_aux(x, y, k) \wedge (k \leqslant i + j) \Rightarrow less\_aux(x, z, i) \vee less\_aux(z, y, j)$.

Proof. (i) By structural induction on $less\_aux(x, y, i)$. (ii) By structural induction on $less\_aux(x, y, i)$. (iii) By the above induction principle and point (ii). (iv) Directly by point (iii). (v) By structural induction on $less\_aux(x, y, i)$ and point (i). The intended meaning is the following: $([\![x]\!]_{\text{str}} < [\![y]\!]_{\text{str}} + i) \wedge ([\![y]\!]_{\text{str}} < [\![x]\!]_{\text{str}} + j) \Rightarrow (0 < i + j)$. (vi) By structural induction on $less\_aux(x, y, k)$ and point (iv). The intended meaning is: $([\![x]\!]_{\text{str}} < [\![y]\!]_{\text{str}} + k) \wedge (k \leqslant i + j) \Rightarrow ([\![x]\!]_{\text{str}} < [\![z]\!]_{\text{str}} + i) \vee ([\![z]\!]_{\text{str}} < [\![y]\!]_{\text{str}} + j)$. □

Lemma 4.2 (*Order: streams*). *Let be* $x, y, z \in str$:
(i) $less(x, y) \Rightarrow \neg less(y, x)$;
(ii) $less(x, y) \Rightarrow less(x, z) \vee less(z, y)$.

Proof. (i) By Lemma 4.1(v). (ii) By Lemma 4.1.(vi). □

Lemma 4.3 (*Addition-auxiliary*). *Let be* $x, y, z, w_1, w_2, v \in str, i, j, k \in Z$:
(i) $add\_aux(x, y, z, i) \wedge (j - big \leqslant -3) \Rightarrow (j - big \leqslant i)$;
(ii) $add\_aux(x, y, z, i) \Rightarrow (-3 \leqslant i)$;
(iii) $add\_aux(x, y, z, i) \wedge (3 \leqslant big + j) \Rightarrow (i \leqslant big + j)$;
(iv) $add\_aux(x, y, z, i) \Rightarrow (i \leqslant 3)$;
(v) $add\_aux(x, y, w1, i) \wedge add\_aux(w1, z, v, j) \wedge add\_aux(y, z, w2, i + j - k) \Rightarrow add\_aux(x, w2, v, k)$;
(vi) $add\_aux(x, y, z, i) \Rightarrow add\_aux(y, x, z, i)$;
(vii) $add\_aux(x, z, w1, i) \wedge add\_aux(y, z, w2, i + j - k) \wedge less\_aux(w1, w2, j) \Rightarrow less\_aux(x, y, k)$.

Proof. (i) By integer induction on j. (ii) Directly by point (i). (iii) By integer induction on j. (iv) Directly by point (iii). (v) By coinduction and points (ii), (iv). The intended meaning is:

$$(\llbracket x\rrbracket_{\mathrm{str}} + \llbracket y\rrbracket_{\mathrm{str}} = \llbracket w_1\rrbracket_{\mathrm{str}} + i) \wedge (\llbracket w_1\rrbracket_{\mathrm{str}} + \llbracket z\rrbracket_{\mathrm{str}} = \llbracket v\rrbracket_{\mathrm{str}} + j) \wedge (\llbracket y\rrbracket_{\mathrm{str}} + \llbracket z\rrbracket_{\mathrm{str}} = \llbracket w_2\rrbracket_{\mathrm{str}} + (i + j - k)) \Rightarrow (\llbracket x\rrbracket_{\mathrm{str}} + \llbracket w2\rrbracket_{\mathrm{str}} = \llbracket v\rrbracket_{\mathrm{str}} + k).$$

(vi) By coinduction. (vii) By structural induction on $less\_aux(w1, w2, j)$, Lemma 4.1(iv) and points (ii), (iv). □

Lemma 4.4 (*Addition: streams*). *Let be* $x, y, z, w_1, w_2, v \in str$:
(i) $add(x, y, w1) \wedge add(w1, z, v) \wedge add(y, z, w2) \Rightarrow add(x, w2, v)$;
(ii) $add(\overline{0}, x, x)$;
(iii) $add(x, -_{str}(x), \overline{0})$;
(iv) $add(x, y, z) \Rightarrow add(y, x, z)$;
(v) $add(x, z, w1) \wedge add(y, z, w2) \wedge less(w1, w2) \Rightarrow less(x, y)$.

Proof. (i) By lemma 4.3(v). (ii) By coinduction. (iii) By coinduction. (iv) By Lemma 4.3(vi). (v) By Lemma 4.3(vii). □

Multiplication: Since the multiplication predicate is defined in terms of the addition one, we get rid of the "auxiliary" level (see Definition 2.4): therefore, we cannot use the `Omega` tactic for carrying out formal proofs about the multiplication. A suitable proof technique is to derive first a suite of auxiliary properties for the addition, then to reduce to such properties via the following preparatory lemma, relating the multiplication to the addition.

Lemma 4.5 (*Multiplication-auxiliary*). *Let be* $x, y, z, w_1, w_2, v \in str$, $a, b, c \in \{0, 1, -1\}$, $i, j, k \in Z$, $m \in \mathbb{N}$, *and* $big = 32$:
(i) $add\_aux(times_{\mathrm{d,str}}(a, \overline{1}), x, x, a)$;
(ii) $mult(x, b : y, c : z) \wedge mult(x, y, w) \Rightarrow add(0 : times_{\mathrm{d,str}}(b, x), 0 : w, c : z)$;
(iii) $add(x, y, z) \Rightarrow add(times_{\mathrm{d,str}}(a, x), times_{\mathrm{d,str}}(a, y), times_{\mathrm{d,str}}(a, z))$.

Proof. (i) By coinduction. (ii) By coinduction. (iii) By coinduction. □

Lemma 4.6 (*Multiplication: streams*). *Let be* $x, y, z, w_1, w_2, u, v \in str$:
(i) $mult(x, y, w1) \wedge mult(w1, z, v) \wedge mult(y, z, w2) \Rightarrow mult(x, w2, v)$;
(ii) $mult(x, \overline{1}, x)$;
(iii) $\neg less(y, x) \wedge \neg less(x, -y) \wedge \neg less(y, 0 : 1 : \overline{0}) \Rightarrow (\times_{\mathrm{str}}(y, div(x, y)) \sim x)$;
(iv) $mult(x, y, z) \Rightarrow mult(y, x, z)$;
(v) $add(y, z, u) \wedge mult(x, y, w1) \wedge mult(x, z, w2) \wedge add(0 : w_1, 0 : w_2, v) \Rightarrow mult(x, 0 : u, v)$;
(vi) $mult(x, z, w1) \wedge mult(y, z, w2) \wedge less(w1, w2) \Rightarrow less(x, y) \vee (less(y, x) \wedge less(z, \overline{0}))$.

Proof. (i) By coinduction and Proposition 4.2(vi). (ii) By coinduction and Lemma 4.5(i). (iii) By coinduction. (iv) By coinduction, Proposition 4.2(vi) and Lemma 4.5(ii). (v) By coinduction, Lemma 4.1(ii) and Lemma 4.5(iii). (vi) By Lemma 4.1(ii), Proposition 4.2(vi) and Lemma 4.4(v). □

Archimedeanity and completeness: It is quite simple to show that the Archimedean axiom is inhabited, while the consistency of the completeness axiom can be established addressing the limit function introduced in Section 2.

5. Conclusion, related and future work

We have built the real numbers in `Coq` using corecursive streams and constructive logic. Then we have proved that our model inhabits a second order axiomatization, which we have proposed and motivated. Hence, streams of signed-digits can be used as a concrete implementation and for addressing formally the reliability of exact algorithms on reals. This

fact is very important from the point of view of the software engineering, because it allows for the development of certified programs working on the reals.

The first full-scale attempt to formalize the analysis is due to Jutting [17], who used the Automath system. Since then, several efforts about formalizations of the real numbers in logical frameworks have been carried out. These works differ depending on the fact that reals are axiomatized or constructed, and on the logical setting used.

The main contribution based on classical logic is probably by Harrison [14], who constructs the real numbers in HOL by a technique closely related to Cantor's method, and then develops a significant part of the mathematical analysis, up to integration of functions of a single real variable.

Constructive real numbers, in the Bishop style [1], have been formalized by various authors: Chirimar and Howe [4] introduce the reals in the Nuprl system and perform a proof of their completeness; Jones [16] uses Lego for studying the completion of general metric spaces; Cederquist [3] uses Half for proving the Hahn–Banach theorem.

Other systems used to develop significant part of the analysis (typically starting from a suitable axiomatization of the real numbers) are Mizar, IMPS, PVS and Isabelle. Focusing specifically to `Coq`, the system is equipped with a library `Reals` [18], which is a classical axiomatization: real numbers are assumed to be a commutative, ordered, Archimedean and complete field. As far as we know, besides the one presented in this paper, just another construction of the reals in `Coq` do exist, developed in the context of the FTA project [12] and documented in [11]. In that contribution, one model for the FTA axiomatization is constructed in `Coq` using Cauchy sequences of rational numbers. Since that model is not computationally efficient, another attempt of the same authors is in progress at the time of writing, through continued fractions.

In the present investigation, we have largely used coinductive tools. Our formal development points out the importance of coinductive principles in theoretical computer science, and shows that they are the most natural and powerful ones for dealing with circular, nonwell-founded entities. We see a lack of these technologies in the current generation of proof assistants and theorem provers. `Coq` is, up to our knowledge, the only proof assistant embarking a proof tactic specific for dealing with coinductive assertions: this pragmatism is extremely successful in the present case, but, more generally, it is still hard working with coinductive definitions and judgments in `Coq` in a "guarded" way.

We are interested to achieve in the future the following goals:

- to design and implement more advanced exact algorithms working on corecursive streams, starting from the standard analytic functions sin, cos, exp and log;
- to extract, test and run exact algorithms in lazy functional programming languages (e.g. `Haskell`);
- to construct alternative, hopefully more efficient, models of the constructive real numbers (e.g. through the implementation of *integers* in `Coq`).

We are also looking for scientific exchange and cooperation with research programs close to ours. Many noncommercial packages for exact real number computation have been implemented in a variety of programming languages: exact arithmetic is reliable and often effective, but less efficient than floating-point practice. Along this direction, probably, the best solution would be the integration of logical frameworks and computer algebra systems with engines for exact numerical evaluation of symbolic expressions.

Finally, our effort has been fruitful also in order to devise a suitable characterization of the constructive reals: we have synthesized an original, minimal axiomatization, equivalent to the alternative ones in the literature by Bridges [2] and FTA [11]. A possible direction for future work is to consider an axiomatization for the constructive reals not requiring the axiom of choice. In this perspective, it would be interesting to consider also a constructive axiomatization obtained by Dedekind cuts: Cauchy sequences and Dedekind cuts provide actually equivalent constructions for the reals only in the case the axiom of choice is available [20]. Results in this sense would help to characterize the fundamental differences between the two constructions.

References

[1] E. Bishop, Foundations of Constructive Analysis, McGraw-Hill, New York, 1967.
[2] D. Bridges, Constructive mathematics: a foundation for computable analysis, Theoret. Comput. Sci. 219, 1999.
[3] J. Cederquist, A pointfree approach to constructive analysis in type theory, Ph.D. Thesis, Göteborg University, 1997.
[4] J. Chirimar, D.J. Howe, Implementing constructive real analysis: preliminary report, Proc. Constructivity in Computer Science, Lecture Notes in Computer Science, Springer, Berlin, Vol. 613, 1992.

[5] A. Ciaffaglione, Certified reasoning on real numbers and objects in co-inductive type theory, Ph.D. Thesis, Dipartimento di Matematica e Informatica, Università di Udine, Italy and INPL-ENSMNS, Nancy, France, 2003.
[6] A. Ciaffaglione, P. Di Gianantonio, A co-inductive approach to real numbers, in: Proc. TYPES, Lecture Notes in Computer Science, Springer, Berlin, Vol. 1956, 2000.
[7] A. Ciaffaglione, P. Di Gianantonio, A tour with constructive real numbers, in: proc. TYPES, Lecture Notes in Computer Science, Springer, Berlin, Vol. 2277, 2001.
[8] A. Ciaffaglione, P. Di Gianantonio, The Web Appendix of this paper, Dipartimento di Matematica e Informatica, Udine, Italy, <http://www.dimi.uniud.it/~ciaffagl/reals.html>, 2004.
[9] T. Coquand, Infinite objects in type theory, Proc. TYPES, Lecture Notes in Computer Science, Springer, Berlin, Vol. 806, 1993.
[10] T. Coquand, G. Huet, The calculus of constructions, Inform. Control, 76, 1988.
[11] H. Geuvers, M. Niqui, Constructive reals in Coq: axioms and categoricity, in: Proc. TYPES, Lecture Notes in Computer Science, Springer, Berlin, Vol. 2277, 2001.
[12] H. Geuvers, R. Pollack, F. Wiedijk, J. Zwanenburg, The Fundamental Theorem of Algebra Project, Computing Science Institute, Nijmegen, The Netherlands, <http://www.cs.kun.nl/~freek/fta/index.html>, 2000.
[13] E. Giménez, Codifying guarded definitions with recursion schemes, Proc. TYPES, Lecture Notes in Computer Science, Springer, Berlin, Vol. 996, 1994.
[14] J.R. Harrison, Theorem proving with real numbers, Ph.D. Thesis, University of Cambridge, 1996.
[15] INRIA, The Coq Proof Assistant, <http://coq.inria.fr/doc/main.html>.
[16] C. Jones, Completing the rationals and metric spaces in Lego, in: Logical Frameworks, 1991.
[17] L.S. Jutting, Checking Landau's Grundlagen in the automath system, Ph.D. Thesis, Eindhoven University of Technology, 1977.
[18] M. Mayero, Formalisation et automatisation de preuves en analyses réelle et numérique, Ph.D. Thesis, Univ. Paris VI, 2001.
[19] P.J. Potts, A. Edalat, M.H. Escardo, Semantics of exact real arithmetic, in: Proc. LICS, 1997.
[20] A.S. Troelstra, D. van Dalen, Constructivism in Mathematics, North-Holland, Amsterdam, 1988.
[21] K. Weihrauch, Computable Analysis, An Introduction, Springer, Berlin, 2000.