ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D. J. Duke , University of York, UK and A.S. Evans, University of Bradford,
UK (Eds)**

2nd BCS-FACS Northern Formal Methods Workshop

Proceedings of the 2nd BCS-FACS Northern Formal Methods
Workshop, Ilkley, 14-15 July 1997

The Real Numbers in $\mathbb{Z}$

W. R. Oliveira and R. S. M. Barros

Springer

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

The Real Numbers in Z

Wilson Rosa de Oliveira*

Departamento de Informática, Universidade Federal de Pernambuco,
Recife-PE, Brazil

Roberto Souto Maior de Barros

Departamento de Informática, Universidade Federal de Pernambuco,
Recife-PE, Brazil

Abstract

Exact real number computation is a fast growing field with applications varying from debugging to specification of numerical to program. We present a specification of the real numbers represented as infinite lists of signed digits in Z. The expressive power and closeness to usual set theoretical mathematical notation gives us a clean and readable specification which is further directly implementable. A comparison with other formal methods is given together with a partial proof that the object being specified is actually the real numbers.

1 Introduction

In contrast to seminumerical computation, numerical computation has been relatively neglected by the formal methods researchers despite the fact that numerical programs are being used in vital activities.

This paper presents a formal specification of the *complete ordered field* of the real numbers $\mathbb{R}$ written in Z [1, 2], from which programs for manipulating *exact real numbers* can be extracted.

Exact real numbers computation is a fast growing field of theoretical computer science which aims at the implementation of the computable reals and of the operations on them [3, 4] by directly manipulating a *representation* for the reals and not their approximations, as it is done in floating point arithmetic. This is achieved by representing the (computable) reals as functions [3] or as infinite lists [4]. In the next section we argue that the most natural representation, namely the decimal expansions, are not adequate for computational purposes.

Starting with the seminal work of Alan Turing in 1937 [5] the notion of computable real numbers has since been extensively studied, particularly by the constructivist mathematicians. This subject has usually been referred to as recursive analysis, constructive analysis, and/or computable analysis.

Obviously, one missing theme on these early studies is that of implementation. Recently, perhaps influenced by the relative success of the functional programming languages, implementation related investigations have emerged – see for example the work of Gianantonio [6, 7] for an updated survey and results. These works can all be classified as using the denotational semantics style of software specification, since they are based on λ -calculus (functional languages) and its models.

The formal specification language chosen to be used in this work is Z, for a number of reasons. Firstly, it is an established language, probably the most widely used formal specification language, with an extensive literature available, and has been adopted in many projects both in academia and in industry [8].

Secondly, Z is a very general and flexible language, and permits the adoption of different levels of abstraction, even within the same specification document. This gives the specifier the necessary freedom to adopt the most appropriate level of abstraction for each part of the specification.

*Supported by the Brazilian Research Council CNPq-RHAE, grant no. 610293192.3.

Finally, we chose to specify the reals by giving a model which could later be proved to be a valid representation of the reals, and Z is model-oriented. Moreover, it has been claimed that, in general, human beings tend to find model-oriented methods easier to understand than their property-oriented counterparts [9, 10, 11].

It should be stressed that the specification here presented can as well be applied to conventional floating point numerical programs either to prove properties or even to debugging.

Attempts of specifying the real numbers using other formal methods, for instance *algebraic specification*, *higher-order logic*, and *type theory*, have been made.

In [12] an extension of the theory of continuous algebraic data types specification is proposed using quasi-metric spaces *in lieu* of partial orders (as originally in [13]). The specification of the computable reals as an abstract data type (ADT) appears in [14]. This approach has all the advantages of the usual ADT specifications and the operations on the reals are defined on the finite parts of its inputs - the infinite is taken care of by means of topological completion. The disadvantage is the (non-standard) mathematical machinery used.

In [15], a construction of the real numbers using the HOL theorem-prover [16] is described. Despite its simplicity the system has two drawbacks which come from (1) the logic underlying HOL which is not constructive and (2) the fact that HOL functions are always total. The definitional way the reals are introduced is not constructive, i.e. there is no mention of representation. This could be easily amended, perhaps not logically coherently, since we can make constructive mathematics in a classical logic. A consequence of the second drawback is that it can be proved that $0/0 = 0$ and, as a result, the following true fact of analysis is not deductible in the system:

$$\forall x : \mathbb{R} \bullet ((\tan(x) = 0) \Rightarrow (\exists n : \mathbb{Z} \bullet x = n * \pi))$$

In [17] some elementary theorems of constructive analysis are proved (or implemented) in the Extended Calculus of Constructions using the LEGO system. The reals are obtained as completion of the rationals and then all operations on the reals are obtained *via* completion. The mathematical machinery is very heavy with no indication of the actual need for it.

The only previous work on the specification of the real numbers in Z that we are aware of is a paper by Sam Valentine [18]. Concisely, Valentine's work is *not* aimed at exact real number computation but to approximate real computation via interval computation. Considerations of computability of the operations involved is not an issue in that work, or so it seems, since the inequality relation defined there is obviously not computable.

Our main goal is to provide a formal specification which is simpler, sufficiently general, and easier to understand. In contrast to the other approaches cited above the mathematics employed is fairly standard and at the level of undergraduate students. Moreover, actual programs can be derived from the specifications and this process is fairly simple. In fact, an example implementation of the specifications presented in this paper has already been done [19].

The organization of the paper is as follows. In Section 2, we motivate the representation we have chosen. We do this by considering it in its simplest guise which is better for presentation and the reader understanding.

Next, we describe the guise we actually use, which is a bit more complicated but gives rise to more efficient algorithms.

Section 4 then presents the formal specification in Z of our representation of $\mathbb{R}$.

Then, the specification of a complete ordered field is presented in Section 5. This specification is mathematically known as being the real numbers up to isomorphisms. In this section we also give some indication of how to prove that our representation satisfies this specification.

Finally, Section 6 draws some conclusions and indicates suggestions for further work.

2 Motivations

One of the most widely used representation of the reals in both practical and theoretical computer science is the signed digit representation (*q.v.* [3, 4, 7, 20, 21]). The reason comes from

- its simplicity when compared with others representations such as Cauchy sequence representation, Continuous Fractions, etc. [6];

- computability of the arithmetic operations: addition is *not* computable on the decimal notation (see below); and
- easy implementation in lazy functional languages.

Here a real number is represented by an infinite list or string of bounded integers (or signed digits). The reals represented in this way are sometimes called **Lazy Reals** since they can be implemented in a lazy functional language as a lazy list.

Let us starting by recalling the somehow surprising argument that in *not* all the representations the arithmetic operations are computable. Consider the usual decimal (radix ten) representation of the reals and suppose we want to multiply a given decimal real by three. If one is given the number $x = 0.333 \dots$, there is no bound on the number of input digits which we must read in order to output a single digit. If we guess that x is really $1/3$ than the result is obviously known to be 1.0, but what if in the fifth digit we had a 2? The result now would be 0.99996... Or what if the stream of digits would be long but finite? Thus we can easily convince ourselves, by this informal argument, that multiplication is not computable in the decimal representation. There is no bound in the input look ahead and one property of a computable process is that one only needs a finite portion of the input in order to produce finite portions of the result.

Similar reasoning brings about the fact that all the other arithmetic operations are not computable and that this is not only a problem of the decimal representation but also of any other integer positive radices!

The solution for that is, amongst others [4, 22, 6], to use *signed digits* where negative digits compensates for erroneous bounded guesses. This is a variant of the redundant balanced radix notation [23]. In this representation we allow digits to be negative as well as positive. The redundant balanced radix notation is also known as signed digits representation. In its simplest form, which we use in this section, it is also known as *tritstreams* representation or modified binary representation, since the only digits allowed are in the set $V = \{-1, 0, 1\}$. We shall, w.l.o.g, limit ourselves to the discussion of reals in $[-1, 1]$.

An obvious disadvantage of this representation (and of any representation which solves the computability of the operations problem) is that it is not at all natural for human beings. Thus the need for an abstract specification which would hide the representation from the user [12].

The rule to convert a string of signed digits is analogous to the one standardly used in, say, converting binary to decimal numbers. The formulae for converting a real number $a_n \dots a_1 a_0 . a_{-1} a_{-2} \dots$ in a radix r to decimal notation is given by:

$$\sum_{i=-\infty}^{\infty} a_i \times b^i$$

Observe that the negative digits subtract from the total value. And that is how we compensate possibly erroneous guesses in a computation.

In the example above (assuming we are using digits in $\{-9, \dots, -1, 0, 1, \dots, 9\}$) we could have, for instance, safely guessed 1 as the first digit of the result just by reading the first digit after the dot in $0.333 \dots$. The reader may check, as an exercise, that if the input becomes say $0.3(-9)(-9) \dots (= 0.2$ in decimal), the output would be $1.(-4)000 \dots = 0.6$. If otherwise the input becomes $0.3999 \dots = 0.4$, the output would be $1.2000 \dots = 1.2$. All the intermediate cases treated in similar way.

Our aim is not to fully study the signed digits representation in all of its variations (on the radices, etc). For this the reader may consult [4]. In [24] an analysis of the best radices which renders the arithmetic operations fully parallel is made. We use the results presented by Avizienis [24] in order to choose the radix 4 (four) in next section.

By restricting ourselves to numbers in the interval $[-1, 1]$ we can represent them as signed binary digits (or signed bits¹), *i.e.* our string will be composed of digits in $V = \{-1, 0, 1\}$. We can still make another abbreviation representing a real number as an infinite word in the alphabet $V = \{-, 0, +\}$.

Let V^ω be the set of streams (infinite strings) on $V = \{-, 0, +\}$. The real number represented by the stream $\alpha = a_1 a_2 \dots$ is defined as $\llbracket \alpha \rrbracket_R$ where

$$\llbracket \alpha \rrbracket_R = \sum_{i \geq 0} a_i \times 2^{-i}$$

Note that a real number has infinitely many representations. We now know what streams denote. Considerations of computability as above leads us to ask the following question: what do the finite strings

¹ Knuth [23], in another context we must say, calls these signed bits *trits* hence *tritstreams* is also a possible name.

Figure 1: The real domain.

(partial streams) denote? A reasonable answer is that $\alpha = a_1 \dots a_n$ should represent the set of possible continuations of α ; hence, an interval. The possible continuations of α is contained in a precisely defined interval.

For example, all streams which have a representation in $[0, \frac{1}{2}]$ have at least one representation starting with $+ -$. In other words, all streams beginning with $+ -$ denote a real number in $[0, \frac{1}{2}]$. We therefore associate any finite string $\alpha = a_1 \dots a_n$ with the interval $[b, c]$ containing the real numbers that can be represented by the streams having α as initial substring. This defines a map $\llbracket \cdot \rrbracket_I$ from V^* to the closed intervals

$$\begin{aligned} \llbracket a_1 \dots a_n \rrbracket_I &= [(\sum_{i=1}^n a_i \times 2^{-i}) - 2^{-n}, (\sum_{i=1}^n a_i \times 2^{-i}) + 2^{-n}] \\ &= [\llbracket a_1 \dots a_n -^\omega \rrbracket_R, \llbracket a_1 \dots a_n +^\omega \rrbracket_R] \end{aligned}$$

$a_1 \dots a_n -^\omega$ and $a_1 \dots a_n +^\omega$ represents (respectively) the smallest and biggest number that can be represented by a stream starting with the string $a_1 \dots a_n$. And there are at most n different strings representing the same $\llbracket a_1 \dots a_n \rrbracket_I$

The reader may convince himself that the definition of $\llbracket \cdot \rrbracket_I$ is sound by proving that every number in the interval $\llbracket a_1 \dots a_n \rrbracket_I$ can be represented by a stream starting with $a_1 \dots a_n$.

Notation 2.1 We use $\llbracket \cdot \rrbracket$ to denote the function which behaves as $\llbracket \cdot \rrbracket_I$ on the strings and as $\llbracket \cdot \rrbracket_R$ on streams.

We have now at our disposal two ways of looking at the real domain. As the set of finite and infinite strings on V ; and as the set of closed intervals plus the real line. Actually, as there are many strings representing the same interval and infinitely many representing the same real, on the set $V^\infty (= V^* \cup V^\omega)$ we define the following equivalence relation:

$$\alpha \equiv \beta \text{ iff } \llbracket \alpha \rrbracket = \llbracket \beta \rrbracket$$

which says that two strings are equivalent when they give the same information. An information order $\sqsubseteq$ can then be defined on the equivalence classes. Indicating by $[\alpha]$ the equivalence class containing α , $[\alpha] \sqsubseteq [\beta]$ iff $\llbracket \beta \rrbracket \subseteq \llbracket \alpha \rrbracket$ (by abuse of notation we see a real number as a singleton set).

The picture in Figure 1 helps to visualize the system. The edges are labeled in V and the vertices are labeled with the intervals. Infinite points are not displayed. The concatenation of the labels of the edges in a path starting from the bottommost vertex gives the string which represents the interval at the end of the path.

Therefore, the longer the string the closer we are to a complete representation of a real number. If a closed interval is contained in another, the second is more defined than the first in the sense that the first is more precise or less ambiguous. This defines an information order on the closed intervals which essentially is the reverse inclusion.

The equivalence on strings can be described more operationally as:

$$0 + x = + - x \quad (1)$$

$$0 - x = - + x \quad (2)$$

which generalizes the obvious arithmetic fact that $0 + \frac{1}{4} = \frac{1}{2} - \frac{1}{4}$ and that $0 - \frac{1}{4} = -\frac{1}{2} + \frac{1}{4}$.

A simple observation is that those intervals which are in the range of $\llbracket \cdot \rrbracket$ are of the form $[\frac{(m-1)}{2^n}, \frac{(m+1)}{2^n}]$ where n is the length of the string, and m is the integer $\sum_{i=1}^n a_i \times 2^{n-i}$. Rationals of the form $m/2^n$, $m, n \in \mathbb{Z}$ are called *dyadic*. Hence, the order can equivalently be thought as the superset relation between intervals with dyadic rationals endpoints (*dyadic intervals*).

Now, what about the infinite strings? We may appropriately call them points since they intend to represent points in the real line.

We may think of them as the Dedekind cuts generated by the closed interval with dyadic rationals endpoints obtained from the finite sections of the infinite strings.

For example, for 0 we have 0^ω , $-+^\omega$ and $+ -^\omega$:

- $+ -^\omega$ corresponds to the ideal generated by $\{[0, \frac{1}{2^k}] \mid k \geq 0\}$
- 0^ω corresponds to the ideal generated by $\{[-\frac{1}{2^k}, \frac{1}{2^k}] \mid k \geq 0\}$
- $-+^\omega$ corresponds to the ideal generated by $\{[-\frac{1}{2^k}, 0] \mid k \geq 0\}$

The ideal generated by the second set is the one taken as the canonical representative for 0 on the quotient of the infinite strings by the equations 1 and 2 above. That is the approach taken in the next section.

3 Choosing a radix

In a *signed digit* (SD) number representation, given a radix r , each digit can assume one of the $2\alpha + 1$ values

$$\Sigma_r = \{-\alpha, \dots, -1, 0, 1, \dots, \alpha\}$$

where the maximum digit magnitude α must be within the region

$$\lceil \frac{r-1}{2} \rceil \leq \alpha \leq r-1.$$

The algebraic value of a $n + k + 1$ finite length SD number

$$y = (y_n \dots y_1 y_0 y_{-1} \dots y_{-k})$$

with $n + 1$ non-fractional and k fractional digits is

$$\llbracket y \rrbracket = \sum_n^{i=-k} y_i r^i.$$

The function $\llbracket \cdot \rrbracket$ can be easily extended to infinite length SD numbers by

$$\llbracket y \rrbracket = \sum_n^{-\infty} y_i r^i.$$

where $n + 1$ is, as before, the number of non-fractional digits. We can then think of a real number as a pair (n, y) where n is an integer and y is a stream of the form $y = y_n \dots y_1 y_0 y_{-1} \dots$.

In [24], it is argued that, in the finite case, by making the lower bound as tighter as

$$\lceil \frac{r+1}{2} \rceil \leq \alpha \leq r-1$$

one could eliminate the carry propagation in additions and subtractions, which then allows the implementation of fully parallel addition and subtraction. In this case the least radix with this property and being a power of two is $r = 4$ with the allowed digits in $\{-3, -2, -1, 0, 1, 2, 3\}$. In what follows negative digits will be displayed as a positive digit overbarred *e.g.* -3 as $\overline{3}$, -2 as $\overline{2}$, etc.

Examples 3.1

1. $\llbracket (3, \overline{1230\overline{2}} \dots) \rrbracket = \llbracket \overline{1230.\overline{2}} \dots \rrbracket = -1 \times 4^3 + 2 \times 4^2 + 3 \times 4^1 + 0 \times 4^0 + (-2) \times 4^{-1} + \dots$
2. $\llbracket (-1, 3^\omega) \rrbracket = \llbracket (0, 03^\omega) \rrbracket = \frac{3}{4} + \frac{3}{16} + \dots = 1$

3.1 Equivalences of numbers in this radix

Let r be the radix, $Succ_Alph$ be the alphabet with its highest digit removed, and Pos_Alph the positive digits of the alphabet. The following equation means that any sequence of digits $d1$ and $d2$ in the middle of the fractional part of a real number can be substituted by the successor of the first digit $d1$ and the digit whose value is the value of $d2$ minus the radix, respectively, without changing the value of the number.

$$\begin{aligned} \forall n1 : FNumber ; n2 : Number ; d1 : Succ_Alph ; d2 : Pos_Alph \bullet \\ n1 \frown \langle d1, d2 \rangle \frown n2 = n1 \frown \langle SUCC\ d1, SUCC^{-r}\ d2 \rangle \frown n2 \end{aligned}$$

For example, assuming $r = 4$, the given specification says there are 18 different equivalences of pairs in the radix. Some examples are: $\overline{3}1 = \overline{2}\overline{3}$, $\overline{3}2 = \overline{2}\overline{2}$, $\overline{2}1 = \overline{1}\overline{3}$, $\overline{1}2 = 0\overline{2}$, $03 = 1\overline{1}$, etc.

Notice that these equivalences also covers cases involving more than two digits. For example, $1000 \dots$ has the same value as $0333 \dots$, because $03 = 1\overline{1}$ and $\overline{1}3 = 0\overline{1}$. This is so because $0333 \dots = 1\overline{1}33 \dots = 10\overline{1}33 \dots = 100\overline{1}33 \dots = 1000 \dots \overline{1}33 \dots = 1000 \dots$

4 A formal specification of the real numbers

In this section, we present a formal specification of the real numbers written in Z.

4.1 The alphabets

Now, we formally define a number of alphabets. The first, $Mult_Alph$, is the extended alphabet which will be used in the intermediate results of the multiplication and division operations. It includes the digits in the interval $\overline{9}.9$ (from -9 to 9).

$$Mult_Alph == \begin{array}{c|c|c|c|c|c|c|c|c|c|c} \overline{9} & \overline{8} & \overline{7} & \overline{6} & \overline{5} & \overline{4} & \overline{3} & \overline{2} & \overline{1} & 0 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

The second, Ext_Alph , is the extended alphabet which will be used in the intermediate results of the addition and subtraction operations and includes the digits in the interval $\overline{6}.6$. It is defined as a subset of the alphabet $Mult_Alph$.

$$Ext_Alph == Mult_Alph \setminus \{\overline{9}, \overline{8}, \overline{7}, 7, 8, 9\}$$

The third, $Alphabet$, is the proper alphabet used to represent the fractional part of the real numbers. Similarly, it is defined as a subset of the extended alphabet.

$$Alphabet == Ext_Alph \setminus \{\overline{6}, \overline{5}, \overline{4}, 4, 5, 6\}$$

The fourth, $Rest_Alph$, is the restricted alphabeth. It is the subset of the alphabet formed by dropping the lowest and the highest numbers, i.e. $\overline{3}$ and 3, respectively.

$$Rest_Alph == Alphabet \setminus \{\overline{3}, 3\}$$

Finally, the carry alphabet *Carry_Alph* is also a subset of the others and contains only the digits $\overline{1}$, 0, and 1.

$$Carry_Alph == Rest_Alph \setminus \{\overline{2}, 2\}$$

4.2 General definitions

Now, we extend Z with the definition of infinite sequences. According to the specification below, infinite sequences of elements of type *X* are defined as total functions from the positive natural numbers to *X*.

$$ISeq\ X == \{ f : \mathbb{N}_1 \rightarrow X \}$$

This definition is similar to the standard definition of sequences, the only difference being the standard Z sequences are finite. For simplicity, we are going to use the same operations and the same notation without formally redefining them here. However, we would like to point out that there is no difficulty in providing such definitions, as most of them are very similar to their finite counterparts [1].

The fractional part of real numbers are then defined as infinite sequences of digits of the alphabet *Alphabet*.

$$\mid Number : ISeq\ Alphabet$$

The type *REAL* is then defined as a pair. The first element is an integer and represents the number of digits of the non-fractional part of the real number. The second is the fractional part of the number and its type is *Number*. In fact, this number contains all the digits of the real number and will be its fractional part only when the integer is -1.

$$REAL == \mathbb{Z} \times Number$$

The function *SUCC* is the usual successor function defined for the digits of the extended alphabet which have a successor in the same alphabet.

$$\left| \begin{array}{l} SUCC : (Ext_Alph \setminus \{6\}) \rightarrow (Ext_Alph \setminus \{\overline{6}\}) \\ \hline SUCC\ \overline{6} = \overline{5} \wedge SUCC\ \overline{5} = \overline{4} \wedge SUCC\ \overline{4} = \overline{3} \wedge \\ SUCC\ \overline{3} = \overline{2} \wedge SUCC\ \overline{2} = \overline{1} \wedge SUCC\ \overline{1} = 0 \wedge \\ SUCC\ 0 = 1 \wedge SUCC\ 1 = 2 \wedge SUCC\ 2 = 3 \wedge \\ SUCC\ 3 = 4 \wedge SUCC\ 4 = 5 \wedge SUCC\ 5 = 6 \end{array} \right.$$

We advise that only some of these auxiliary definitions are included here. Being more precise, we only include the ones needed for the specification of addition and subtraction. The others are also straightforward and are omitted for brevity.

4.3 Addition

The function *DIRECT_SUM*, presented below, specifies how to add two digits of the proper alphabet (*Alphabet*) returning a digit of the extended alphabet (*Ext_Alph*). This function implements the usual addition and is defined in terms of the successor function (*SUCC*) previously specified.

According to the specification, adding *zero* to a number *b* returns the same number, adding a positive number *a* to *b* is the same as adding *a* - 1 to *b* + 1, whereas adding a negative number *a* to *b* is the same as adding *a* + 1 to *b* - 1.

$\overline{DIRECT_SUM} : Alphabet \rightarrow Ext_Alph \rightarrow Ext_Alph$	$\forall a : Alphabet ; b : Ext_Alph \bullet$ $DIRECT_SUM\ a\ b =$ $(\text{if } a = 0$ $\text{then } b$ else $\text{if } a \in \{1, 2, 3\}$ $\text{then } DIRECT_SUM\ (SUCC^{-1}\ a)\ (SUCC\ b)$ $\text{else } DIRECT_SUM\ (SUCC\ a)\ (SUCC^{-1}\ b))$
---	---

Notice that, although the given specification of *DIRECT_SUM* defines the type of its second formal parameter as *Ext_Alph*, the first call of *DIRECT_SUM* *must* receive a member of the proper alphabet as its second actual parameter, because this is the precondition that guarantees the result of *DIRECT_SUM* will be a member of the extended alphabet. In other words, only the recursive calls of *DIRECT_SUM* may receive the digits $\bar{0}$, $\bar{5}$, $\bar{4}$, 4, 5, and 6 as its second actual parameter.

Notice however that this is *not* a problem as all the non-recursive invocations of *DIRECT_SUM* in the specifications presented here do satisfy this requirement.

The function *ADJUSTED_SUM* converts a digit of the extended alphabet to a digit of the restricted alphabet. There are three possibilities: the parameter *a* is already a digit of the restricted alphabet and the result is also *a*, the parameter is greater than two and the result is $a - 4$, or the parameter is less than $\bar{2}$ and the result is $a + 4$.

$\overline{ADJUSTED_SUM} : Ext_Alph \rightarrow Rest_Alph$	$\forall a : Ext_Alph \bullet$ $ADJUSTED_SUM\ a = (\text{if } a \in Rest_Alph$ $\text{then } a$ else $\text{if } a \in \{3, 4, 5, 6\}$ $\text{then } SUCC^{-4}\ a$ $\text{else } SUCC^4\ a)$
---	--

It is important to notice that 4 is the radix and that $a + 4$ and $a - 4$ are specified as four successive applications of the successor function and its inverse respectively.

The function *CARRY*, specified below, returns the carry which results from converting a digit of the extended alphabet to a digit of the restricted alphabet. The result will be *zero* if the parameter is already a digit of the *restricted* alphabet, *one* if it is bigger than *two*, and $\bar{1}$ otherwise.

$\overline{CARRY} : Ext_Alph \rightarrow Carry_Alph$	$\forall a : Ext_Alph \bullet$ $CARRY\ a = (\text{if } a \in Rest_Alph$ $\text{then } 0$ else $\text{if } a \in \{3, 4, 5, 6\}$ $\text{then } 1$ $\text{else } \bar{1})$
--	---

The next function, *FINAL_SUM*, adds a digit of the restricted alphabet *a* with a digit of the carry alphabet *b*. If the carry is *zero*, the result is *a*; If the carry is positive, the result is $a + 1$; otherwise, the result is $a - 1$. The specification of *FINAL_SUM* is presented below.

$FINAL_SUM : Rest_Alph \rightarrow Carry_Alph \rightarrow Alphabet$	$\forall a : Rest_Alph ; b : Carry_Alph \bullet$ $FINAL_SUM\ a\ b = (\text{if } b = 0$ $\quad \text{then } a$ $\quad \text{else}$ $\quad \quad \text{if } b = 1$ $\quad \quad \text{then } SUCC\ a$ $\quad \quad \text{else } SUCC^{-1}\ a)$
--	---

The recursive function *STRING_SUM* describes how to add two numbers (only the fractional parts) which have the same number of non-fractional digits. The result is the concatenation of the first digit of the result with the *STRING_SUM* of the tail of the two numbers.

Given an adjusted digit returned by the *ADJUSTED_SUM* function (*as1*) and the carry returned by the application of the *CARRY* function to the sum of the digits on the *next position to the right* (*t2*), the first digit of the result is the *FINAL_SUM* of *as1* and *t2* (*fs1*).

$STRING_SUM : Number \rightarrow Number \rightarrow Number$	$\forall x, y : Number \bullet$ $STRING_SUM\ x\ y =$ $\quad (\text{let } as1 == ADJUSTED_SUM$ $\quad \quad \quad (DIRECT_SUM\ (x\ 1)\ (y\ 1));$ $\quad \quad t2 == CARRY\ (DIRECT_SUM\ (x\ 2)\ (y\ 2));$ $\quad \quad fs1 == FINAL_SUM\ as1\ t2 \bullet$ $\quad \quad (fs1) \frown (STRING_SUM\ (tail\ x)\ (tail\ y)))$
--	--

Finally, the addition operation of two numbers of type *REAL* can be specified as the function *ADD* presented below. Notice that adding two real numbers (*m, x*) and (*n, y*) can only be done directly if *m = n*. Thus, if this is not the case, one of the two must be converted. The easiest way to achieve this is to convert the smaller of the two numbers by successively adding one to its order and concatenating zeros on the left hand side of its fractional part.

$ADD : REAL \rightarrow REAL \rightarrow REAL$	$\forall m, n : \mathbb{Z} ; x, y : Number \bullet$ $ADD\ (m, x)\ (n, y) =$ $\quad (\text{if } m \rangle n$ $\quad \quad \text{then } ADD\ (m, x)\ (n + 1, \langle 0 \rangle \frown y)$ $\quad \quad \text{else}$ $\quad \quad \quad \text{if } n \rangle m$ $\quad \quad \quad \text{then } ADD\ (m + 1, \langle 0 \rangle \frown x)\ (n, y)$ $\quad \quad \quad \text{else}$ $\quad \quad \quad \text{if } DIRECT_SUM\ (x\ 1)\ (y\ 1) \notin Rest_Alph$ $\quad \quad \quad \text{then } (m + 1, STRING_SUM\ (\langle 0 \rangle \frown x)\ (\langle 0 \rangle \frown y))$ $\quad \quad \quad \text{else } (m, STRING_SUM\ x\ y))$
--	---

Observe that this definition does indeed guarantee the corresponding digits on the two real numbers passed to *STRING_SUM* are of the same order of magnitude, i.e. corresponding digits can be added directly. Moreover, it also guarantees the result will not have an extra digit on the left hand side of the string, i.e. there will be no carry as the result of adding the first digit of the two strings. This completes the specification of the addition operation.

4.4 Subtraction

Now, we begin the specification of the subtraction operation. Firstly, we specify the function *COMPL*, below, which receives a digit of the alphabet and returns its complement to zero, i.e. it returns the digit of the alphabet such that the *DIRECT_SUM* of the two is zero. For example, *COMPL* 2 = 7 and *COMPL* 1 = 9.

$$\frac{\text{COMPL} : \text{Alphabet} \rightarrow \text{Alphabet}}{\forall a : \text{Alphabet} \bullet \text{DIRECT_SUM } a \text{ (COMPL } a) = 0}$$

In this representation, changing the signal of a real number is the same as changing the signal of each digit of its fractional part, which is specified as the recursive function *MINUS*, presented below. According to the specification, applying *MINUS* to a number involves the application of function *COMPL* to the first digit of the number followed by a recursive application of *MINUS* to the tail of the number.

$$\frac{\text{MINUS} : \text{Number} \rightarrow \text{Number}}{\forall x : \text{Number} \bullet \text{MINUS } x = \langle \text{COMPL } (x \text{ 1}) \rangle \frown \text{MINUS } (\text{tail } x)}$$

Finally, subtracting two real numbers is the same as adding the first number to the second number with its signal changed, which is specified as the function *SUBTRACT*.

$$\frac{\text{SUBTRACT} : \text{REAL} \rightarrow \text{REAL} \rightarrow \text{REAL}}{\forall m, n : \mathbb{Z}; x, y : \text{Number} \bullet \text{SUBTRACT } (m, x) \text{ (} n, y) = \text{ADD } (m, x) \text{ (} n, \text{MINUS } y)}$$

This completes the specification of the subtraction operation.

4.5 Multiplication

This section presents the specification, in Z, of the multiplication operation. The multiplication of two real numbers (m, b) and (n, d) is by definition:

$$r^m * \sum_{i=n}^{-\infty} b * d_i * r^i.$$

Developing this expression will result in

$$\sum_{k=0}^{\infty} \sum_{i,j | i+j=k} b_{m-i} * d_{n-j} * r^{m+n-k},$$

which underlies the basic idea behind the specification presented here.

Also, notice that this multiplication symbol (*) stands for the multiplication of finite signed digit numbers which can be easily specified [24].

Before proceeding to the multiplication per se, we present the function *DF_ADD*, which specifies how to add a digit of the restricted alphabet *Rest_Alph* to the first digit of the fractional part of a real number. Its result is the *DIRECT_SUM* of the two digits followed by the rest of the sequence.

$$\frac{\text{DF_ADD} : \text{Rest_Alph} \rightarrow \text{Number} \rightarrow \text{Number}}{\forall d : \text{Rest_Alph}; x : \text{Number} \bullet \text{DF_ADD } d \text{ } x = \langle \text{DIRECT_SUM } d \text{ (} x \text{ 0)} \rangle \frown (\text{tail } x)}$$

There is also a variation of this function called *DF_ADD_WC*, omitted here, which also allows for the cases where there is a carry. Both of these functions are used below.

The next function we define is *DF_MULT* which specifies how to multiply a digit of the restricted alphabet by the fractional part of a real number.

The functions *DIV* and *MOD* represent the usual integer division and the rest of the division of two integers. Their specifications, omitted here, are also fairly simple.

$$\begin{array}{|l}
 \hline
 DF_MULT : Rest_Alph \rightarrow Number \rightarrow Number \\
 \hline
 \forall d : Rest_Alph ; x : Number \bullet \\
 \quad DF_MULT \, d \, x = \\
 \quad \quad (\text{let } x0 == (x \, 1); \\
 \quad \quad \quad x1 == (x \, 2); \\
 \quad \quad \quad xt == tail (tail \, x); \\
 \quad \quad \quad z2 == d * x1 \, MOD \, 4; \\
 \quad \quad \quad z01 == d * x0 + (d * x1 \, DIV \, 4); \\
 \quad \quad \quad z1 == ADJUSTED_SUM (z01 \, MOD \, 4); \\
 \quad \quad \quad z0 == FINAL_SUM (z01 \, DIV \, 4) \\
 \quad \quad \quad \quad \quad \quad (CARRY (z01 \, MOD \, 4)) \\
 \quad \quad \bullet \langle z0 \rangle \cap (DF_ADD \, z1 \\
 \quad \quad \quad (DF_ADD_WC \, z2 (DF_MULT \, d \, xt))))
 \end{array}$$

The recursive function *STRING_MULT* describes how to multiply the fractional parts of two numbers of type *Real*. Each invocation of this function calculates two digits as results (*z0* and *z1*). The results of consecutive invocations will then be added using the addition operation defined previously (*STRING_SUM*).

$$\begin{array}{|l}
 \hline
 STRING_MULT : Number \rightarrow Number \rightarrow Number \\
 \hline
 \forall x, y : Number \bullet \\
 \quad STRING_MULT \, x \, y = \\
 \quad \quad (\text{let } md == (x \, 1) * (y \, 1); \\
 \quad \quad \quad z0 == md \, DIV \, 4; \\
 \quad \quad \quad z1 == md \, MOD \, 4; \\
 \quad \quad \quad part1 == \langle z0 \rangle \cap \langle z1 \rangle \cap (STRING_MULT (tail \, x) (tail \, y)); \\
 \quad \quad \quad part2 == STRING_SUM (DF_MULT (x \, 1) (tail \, y)) \\
 \quad \quad \quad \quad \quad \quad (DF_MULT (y \, 1) (tail \, x)) \\
 \quad \quad \bullet STRING_SUM \, part1 \, part2)
 \end{array}$$

Finally, the multiplication of two numbers of type *REAL* is specified as the function *MULT* presented below. Notice that, unlike the addition operation, the multiplication of two real numbers (*m, x*) and (*n, y*) can be done directly irrespective of the values of *m* and *n*. Also, the number of digits of the non-fractional part of the result is always *m + n*.

$$\begin{array}{|l}
 \hline
 MULT : REAL \rightarrow REAL \rightarrow REAL \\
 \hline
 \forall m, n : \mathbb{Z} ; x, y : Number \bullet \\
 \quad MULT \, (m, x) \, (n, y) = (m + n, STRING_MULT \, x \, y)
 \end{array}$$

This completes the specification of the multiplication operation.

4.6 Division

This section presents the specification, in Z, of the division operation. The division of two real numbers *x* and *y* is, by definition, a real number *z* such that the multiplication of *y* by *z* equals to *x*. However, the division operation is only applicable if *y* is *not* zero.

Here, we chose to present this higher-level specification of *DIVIDE* instead of the more operational specifications.² Notice that, in the model of the reals we use in this paper, a number is zero if its fractional part is one of the following: 0^ω , $\overline{1}3^\omega$, or $\overline{1}\overline{3}^\omega$.

$$\left| \begin{array}{l} \textit{DIVIDE} : \textit{REAL} \rightarrow \textit{REAL} \rightarrow \textit{REAL} \\ \hline \forall m, n : \mathbb{Z}; x, y : \textit{Number} \\ \quad | \quad y \notin \{0^\omega, \overline{1}3^\omega, \overline{1}\overline{3}^\omega\} \bullet \\ \quad \quad (m, x) = \textit{MULT} (n, y) (\textit{DIVIDE} (m, x) (n, y)) \end{array} \right|$$

This concludes our specification of division and this section.

5 The real numbers

Now, we present the properties that any model of the real numbers must satisfy. These are the properties we should prove to show that the specification given in section 4 is a valid representation of the real numbers.

The first equations given below say there must be representations for *zero* and for *one*. The third says zero and one are different numbers.

$$\begin{aligned} 0 &\in \mathbb{R} \\ 1 &\in \mathbb{R} \\ 0 &\neq 1 \end{aligned}$$

In the model specified in section 4, a representation for zero is $(0, 0^\omega)$ and a representation for one is $(0, 3^\omega)$, where ω stands for an infinite repetition of the last digit. None of the equalities in Subsection 3.1 relates pairs of zeroes with pairs of threes and thus $(0, 3^\omega) \neq (0, 0^\omega)$.

The next two equations say the addition operation is commutative and associative, respectively. The following two say *zero* is its neutral element and that subtraction is its inverse operation.

$$\begin{aligned} \forall x, y : \mathbb{R} \bullet x + y &= y + x \\ \forall x, y, z : \mathbb{R} \bullet x + (y + z) &= (x + y) + z \\ \forall x : \mathbb{R} \bullet 0 + x &= x \\ \forall x : \mathbb{R} \bullet (-x) + x &= 0 \end{aligned}$$

The first three can be easily obtained from the respective properties of the rational approximations. The last one follows from the definition of changing the signal given in the last section.

The next four equations say the multiplication operation is also commutative and associative, that *one* is its neutral element, and that division is its inverse operation.

$$\begin{aligned} \forall x, y : \mathbb{R} \bullet x * y &= y * x \\ \forall x, y, z : \mathbb{R} \bullet x * (y * z) &= (x * y) * z \\ \forall x : \mathbb{R} \bullet 1 * x &= x \\ \forall x : \mathbb{R} \mid x \neq 0 \bullet (1/x) * x &= 1 \end{aligned}$$

The only apparently non-trivial is the last equation. But it also follows easily from the specification of *DIVIDE*.

The next equation says the multiplication operation is distributive with respect to the addition operation.

$$\forall x, y, z : \mathbb{R} \bullet x * (y + z) = (x * y) + (x * z)$$

The order axioms can be readily checked from the definitions above. The reals are totally ordered.

$$\begin{aligned} \forall x, y : \mathbb{R} \bullet x = y \vee x < y \vee y < x \\ \forall x, y, z : \mathbb{R} \bullet (x < y \wedge y < z) &\Rightarrow (x < z) \\ \forall x : \mathbb{R} \bullet \neg (x < x) \end{aligned}$$

²The operational *DIVIDE* is presented as an appendix.

The order relation and the arithmetic operations interact.

$$\begin{aligned} \forall y, z : \mathbb{R} \bullet ((y < z) \Rightarrow \forall x : \mathbb{R} \bullet (x + y) < (x + z)) \\ \forall x, y : \mathbb{R} \bullet ((0 < x \wedge 0 < y) \Rightarrow (0 < x * y)) \end{aligned}$$

The completeness axiom below says that any non-empty set of reals which is bounded above has a least upper bound. The proof uses the fact the real line is the topological completion of the additive group of the rational numbers and is omitted here (but can be adapted from the one given in Bourbaki [25, Chapter IV, page 331, Proposition I]).

$$\begin{aligned} \forall S : \mathbb{P}_1 \mathbb{R} \mid (\exists M : \mathbb{R} \bullet (\forall x : S \bullet x \leq M)) \bullet \\ (\exists m : \mathbb{R} \bullet ((\forall x : S \bullet x \leq m) \wedge \\ (\forall m' : \mathbb{R} \mid m' < m \bullet (\exists x : S \bullet m' < x)))) \end{aligned}$$

6 Conclusions

In contrast to seminumerical computation, numerical computation has been relatively neglected by the formal methods researchers despite the fact that numerical programs are being used in vital activities on our day-life. We have proposed the use of the Z notation for the specifications of the reals based on its success as a powerful and simple to use specification language. We intend to apply our specification in the development of numerical software. The idea is to use the tools from the specification bag to the rigorous development and analysis of numerical programs.

The natural follow up of this paper is the specification of elementary real analysis such as integration, metric spaces, etc.

The design of a numerical programming environment is being envisaged.

An unfair and misleading criticism that is often made against our approach is to compare our specification, based in practical usability, to floating point numbers. We aim at exact real number computation and there is no place for floating point numbers here. We could say that there are, at least, three levels of abstraction here: (1) The Reals, (2) The Computable Reals, and (3) The Floating Point numbers. Level 2 is the implementation (in say Turing machines) of the mathematics of Level 1. Level 3 implements, in actual computers, the finite part of Level 2. Our specification aims at developing a theory based in the Z notation for the mathematics of Level 2.

References

- [1] Spivey J. M. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hemel Hempstead, UK, second edition, 1992.
- [2] Diller A. *Z: An Introduction to Formal Methods*. John Wiley & Sons Ltd., Chichester, UK, second edition, 1994.
- [3] Hans Boehm and Robert Cartwright. Exact real arithmetic: Formulating reals numbers as functions. Technical Report Rice COMP TR88-66, Department of Computer Science, Rice University, April 1988.
- [4] Hans Boehm, Robert Cartwright, Michael J. O'Donnell, and Mark Riggle. Exact real arithmetic: A case study in higher order programming. In *Proceedings of the Lisp and Functional Programming Conference*, pages 162–173, 1986.
- [5] Alan Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1936.
- [6] P. Di Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD thesis, Università Degli Studi di Pisa, Dipartimento di Informatica, 1993.
- [7] P. Di Gianantonio. Real numbers computability and domain theory. Submitted to Information and Computation. Available electronically at <http://www.cwi.nl/~pietro>, 1996.

- [8] Hall A. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.
- [9] Plat N. *Experiments with Formal Methods in Software Engineering*. PhD thesis, Delft University of Technology, Faculty of Technical Mathematics and Informatics, The Netherlands, 1993.
- [10] McParland P. J. *Software tools to Support Formal Methods*. PhD thesis, The Queen’s University of Belfast, Northern Ireland, October 1989.
- [11] Bloomfield R. E. and Froome P. K. D. The application of formal methods to the assessment of high integrity software. *IEEE Transactions on Software Engineering*, 20(9):988–993, September 1986.
- [12] W. R. de Oliveira and M. B. Smyth. Quasimetric Σ -algebras. In *SEMISH’94, XXI Seminario Integrado de Software e Hardware*, pages 547–561, Caxambu, Minas Gerais, Brazil, 31/07 to 05/08 1994. Sociedade Brasileira de Computação.
- [13] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.
- [14] W. R. de Oliveira and M. B. Smyth. Non-deterministic quasimetric algebras. In Preparation.
- [15] John Harrison. Constructing the real numbers in HOL. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 145–164, Leuven, Belgium, September 1992. North Holland.
- [16] Gordon M. J. C. and Melham T. J., editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, UK, 1993.
- [17] C. Jones. Completing the rationals and metric spaces in LEGO. In G. Plotkin G. Huet and C. Jones, editors, *Proceedings of the 2nd Workshop on Logical Frameworks*, Edimburgh, May 1992.
- [18] S. H. Valentine. Putting numbers into the mathematical toolkit. In Bowen J. P. and Nicholls J. E., editors, *Z User Workshop, London 1992*, Workshops in Computing Series, pages 9–36. Springer-Verlag, 1993.
- [19] Mlexener B. Romeiro. Implementando a aritmética real exata. Electronic notes, Departamento de Informática CCEN-UFPE, <http://www.di.ufpe.br/~mbr/relatorio.ps>, 1996.
- [20] K. Weihrauch. Constructivity, computability and computational complexity in analysis. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Fundamentals of Computation Theory*, volume 380 of *Lecture Notes in Computer Science*, pages 480–493, Szeged, Hungary, August 1989. Springer-Verlag.
- [21] K. I. Ko and H. Friedman. Computational complexity of real functions. *Theoretical Computer Science*, 20, 1982.
- [22] H. James Hoover. Feasible real arithmetic circuits. *SIAM Journal of Computing*, 19(1):182–204, February 1990.
- [23] Donald Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, volume I. Addison-Wesley, 1969. section 4.5.
- [24] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 1961.
- [25] N. Bourbaki. *Elements of Mathematics: General Topology*, volume I. Hermann/Addison-Wesley, 1966. Translation of 3rd. edition of *Topologie Générale*.

Appendix

This appendix includes the specification of the operational divide, which was omitted from the specification presented in Section 4.

The first function we specify in this appendix is *REMAINDER*. Given the two operands of the division operation (x and y) and the first digit of the corresponding division (q) as parameters, this function returns the remainder. In practice, this remainder is the new number that, in the next step, will be divided by y .

$\overline{REMAINDER : Number \rightarrow Number \rightarrow Alphabet \rightarrow Number}$	$\forall x, y : Number ; q : Alphabet \bullet$ $STRING_DIV \ x \ y \ q =$ $(\text{let } r == STRING_SUM \ x \ (MINUS \ (DF_MULT \ q \ y));$ $r1 == (r \ 1) \bullet$ $\text{if } r1 = 0$ $\text{then } tail \ r$ else $\text{if } r1 = 1$ $\text{then } \langle SUCC^4 \ (r \ 2) \rangle \frown (tail \ (tail \ r))$ $\text{else } \langle SUCC^{-4} \ (r \ 2) \rangle \frown (tail \ (tail \ r)) \)$
--	---

The recursive function *STRING_DIV* describes how to divide the fractional parts of two numbers of type *Real*. Notice that each invocation of *STRING_DIV* calculates one digit (q) of the final result. Also, only the first 2 digits of the two operands (x and y) are enough to find a value for q .

The following invocations of *STRING_DIV* will use the result of the previous application of function *REMAINDER* as its first parameter and the calculated digits will be concatenated to form the final result.

$\overline{STRING_DIV : Number \rightarrow Number \rightarrow Number}$	$\forall x, y : Number \bullet$ $STRING_DIV \ x \ y =$ $(\text{let } x12 == (x \ 1) * 16 + (x \ 2) * 4;$ $y12 == (y \ 1) * 4 + (y \ 2);$ $q == x12 \ DIV \ y12;$ $newx == REMAINDER \ x \ y \ q$ $\bullet \langle q \rangle \frown (STRING_DIV \ newx \ y) \)$
---	---

It is important to point out that the specification above is a simplified version of *STRING_DIV* which ignores the fact that we need conversion functions between elements of type *alphabet* and the integers. We decided to omit these details to make the specification easier to understand.

Finally, the division of two numbers of type *REAL* is specified as the function *DIVIDE* presented below. As in the multiplications, the division of two real numbers (m, x) and (n, y) can be done directly irrespective of the values of m and n . However, the number of digits of the non-fractional part of the result may vary.

$\overline{DIVIDE : REAL \rightarrow REAL \rightarrow REAL}$	$\forall m, n : \mathbb{Z} ; x, y : Number \bullet$ $DIVIDE \ (m, x) \ (n, y) =$ $(\text{let } q == STRING_DIV \ x \ y;$ $q1 == (q \ 1) \bullet$ $\text{if } q1 \in Rest_Alph$ $\text{then } (m - n, q)$ else $\text{if } q1 \succ 2$ $\text{then } (m - n + 1, \langle 1 \rangle \frown \langle SUCC^{-4} \ q1 \rangle \frown (tail \ q))$ $\text{else } (m - n + 1, \langle \bar{1} \rangle \frown \langle SUCC^4 \ q1 \rangle \frown (tail \ q)) \)$
--	---