Lazy functional algorithms for exact real functionals

23rd International Symposium, MFCS'98 Brno, Czech Republic, August 24–28, 1998 Proceedings

Simpson, A. (1998). Lazy functional algorithms for exact real functionals. In Brim, L., Gruska, J., & Zlatuska, J. (Eds.), Mathematical Foundations of Computer Science 1998. (pp. 456-464). (Lecture Notes in Computer Science). Springer Berlin / Heidelberg. doi: 10.1007/BFb0055795

Lazy Functional Algorithms for Exact Real Functionals

Alex K. Simpson

LFCS, Department of Computer Science, University of Edinburgh,
JCMB, King's Buildings, Edinburgh, EH9 3JZ, Scotland
<Alex.Simpson@dcs.ed.ac.uk>

Abstract. We show how functional languages can be used to write programs for real-valued functionals in exact real arithmetic. We concentrate on two useful functionals: definite integration, and the functional returning the maximum value of a continuous function over a closed interval. The algorithms are a practical application of a method, due to Berger, for computing quantifiers over streams. Correctness proofs for the algorithms make essential use of domain theory.

1 Introduction

In exact real number computation, infinite representations of reals are employed to avoid the usual rounding errors that are inherent in floating point computation [4–6, 17]. For certain real number computations that are highly sensitive to small variations in the input, such rounding errors become inordinately large and the use of floating-point algorithms can lead to completely erroneous results [1, 14]. In such situations, exact real number computation provides guaranteed correctness, although at the (probably inevitable) price of a loss of efficiency. How to improve efficiency is a field of active research [9].

Lazy functional programming provides a natural implementational style for exact real algorithms. One reason is that lazy functional languages support lazy infinite data structures, such as streams, which can be coveniently used to represent real numbers. The efficient management of such infinite data structures (for example, using call-by-need to avoid repeated computations) can be entrusted to the language implementer, leaving the programmer free to concentrate on the essentials of the algorithms being developed. Also, functional programming naturally supports the recursive definition of functions, which is the most useful method of defining exact functions on real numbers. Such considerations were important motivating factors in [4, 5, 17, 6, 7, 10].

One principal distinguishing feature of functional languages is their acceptance of functions as first-class values, and the associated possibility of passing functions as arguments to other function(al)s. In the context of exact real number computation, this raises the question of whether it is possible to write functional algorithms to implement useful functionals on real numbers. In [8], Edalat and Escardó show how to extend Real PCF [10] with primitive functionals for definite integration, and for the maximum value attained by a continuous function

over a closed interval. However, their operational semantics is nondeterministic, and requires a parallel evaluation strategy which is not readily supported within the context of the standard *sequential* functional languages. The problem of whether such algorithms are possible sequentially was originally posed in Di Gianantonio's PhD thesis [6], where it was conjectured that they are not.

In this paper we show that Di Gianantonio's conjecture is false. We provide sequential functional algorithms for the specific and useful functionals of integration and maximum. The algorithms rely on a clever, but little known, idea of Berger, who showed how to compute quantifiers over predicates on streams sequentially [2]. Berger's algorithms deserve to be better known, especially in the light of their possible applications.

The work of Berger, Di Gianantonio, Escardó and Edalat, referred to above, was carried out in the context of the minimal functional language PCF [15] (and extensions of it). It would be fully possible to write this paper in the same setting, but we prefer instead to adopt a less spartan approach. The goal of this paper is to describe and verify particular functional algorithms. We therefore use an easily readable, although not formally defined, functional pseudocode for expressing algorithms (just as an informal imperative pseudocode is used to specify algorithms throughout computer science). We also make use of a simple type discipline to specify the domains and codomains of functions.

Not only does the type discipline improve the readability of the code, it also serves a more significant purpose. The statements of correctness of the algorithms and their verification make essential use of a denotational semantics defined in terms of the type structure. Indeed it is a further benefit of using a functional language that a denotational semantics is easily obtained using standard constructions on complete partial orders. Because we have not formally defined the language, we cannot formally define its semantics either. Nonetheless, the denotational semantics of functional programming languages is now well enough understood that it is possible to use such semantics in an informal way with full mathematical rigour. Our approach is to use denotational semantics as one more mathematical tool for verifying informally specified algorithms, alongside all the other tools available from the body of mathematics as a whole.

Perhaps what is most interesting about the use of denotational semantics in this paper is that it goes beyond the mere existence of fixed-points and their basic properties. Instead, the correctness proofs make use of topological properties (moduli of continuity) of the denotations of higher-order functions. Understanding the denotational semantics is helpful even to appreciate the correctness of the algorithms informally. In order to verify the algorithms rigorously, some use of domain theory appears to be essential.

2 Types and their Denotations

In our functional pseudocode, we assume basic datatypes like `int`, the type of integers, `bool`, the type of booleans, as well as some convenient finite types:

```
type two = {0,1}
```

```
type three = {-1,0,1}
```

We assume that `two` is a subtype of `three` in the obvious way (so we shall not bother to include explicit coercions between them). The type constructors we use are `A` $\to$ `B`, function space, `A` $\times$ `B`, cartesian product, and `A stream`. Function application is assumed to be lazy. Mainly for denotational simplicity, we interpret $\times$ as a lazy product (thus a pair may converge in one component but not the other). The behaviour of streams is best explained via the denotational semantics.

For the denotational semantics, we use directed-complete partial orders with least element (henceforth cpos) for interpreting datatypes, and continuous functions between them for interpreting programs (see e.g. [12]). In Sec. 3 we refer to cpos as topological spaces, understanding them as carrying the Scott topology.

Given a set X, we write $X_\perp$ for the flat cpo with least element $\perp$ and with all other elements taken from the set X. Basic types are interpreted as flat cpos by: $[\![\texttt{int}]\!] = \mathbb{Z}_\perp$; $[\![\texttt{bool}]\!] = \mathbb{B}_\perp$ where $\mathbb{B} = \{true, false\}$; $[\![\texttt{two}]\!] = \mathbf{2}_\perp$ where $\mathbf{2} = \{0,1\}$; and $[\![\texttt{three}]\!] = \mathbf{3}_\perp$ where $\mathbf{3} = \{\perp 1, 0, 1\}$. The function space is interpreted as the cpo, $[\![\texttt{A} \to \texttt{B}]\!]$, of all continuous functions from $[\![\texttt{A}]\!]$ to $[\![\texttt{B}]\!]$ ordered pointwise.The interpretation of the product type, $[\![\texttt{A} \times \texttt{B}]\!]$, is the straightforward cartesian product of $[\![\texttt{A}]\!]$ and $[\![\texttt{B}]\!]$ (as partially ordered sets).

Streams will be denoted by possibly infinite sequences, so we develop some notation for these. For a set X we write: X^* for the set of finite sequences of its elements; X^ω for the set of infinite sequences; and X^∞ for the set of all sequences, i.e. $X^\infty = X^* \cup X^\omega$. For any sequence α, we write $|\alpha|$ for the (possibly infinite) length of α and $\alpha(i)$ (where $0 \leq i < |\alpha|$) for the $(i+1)$-th element of α. We use textual juxtaposition, $\alpha\beta$, for the concatenation of a finite sequence α with an arbitrary sequence β. We write $\alpha\lceil_n$ for the largest finite prefix, β, of α such that $|\beta| \leq n$. For $x \in X$ we write $\overline{\overline{x}}$ for the infinite constant sequence. In the paper, we shall only ever use streams formed from base types. These have a straightforward interpretation. If $[\![\texttt{A}]\!] = \texttt{X}_\perp$ then $[\![\texttt{A stream}]\!] = X^\infty$ with $\alpha \leq \beta$ if and only if α is a prefix of β. We write $hd : X^\infty \to X_\perp$ and $tl : X^\infty \to X^\infty$ for the evident head and tail functions. The "cons" operation on streams (written `::` in our pseudocode) is the evident left-strict function from $X_\perp \times X^\infty$ to X^∞.

3 Real Numbers: Representation and Semantics

In order to write algorithms for functions and functionals on the reals, we first need to choose a representation for real numbers. It is well known that the standard base n notation for reals does not provide an adequate representation, as many simple functions (e.g. addition) are not computable exactly. However, many alternative choices of adequate representation are available. There are discussions of these issues in e.g. [6, 9].

We shall use one of the simplest possible representations: a modification of the standard binary representation using negative digits. We consider an infinite sequence $\alpha \in \mathbf{3}^\omega$ (recall that $\mathbf{3} = \{\perp 1, 0, 1\}$) as representing the real number

$$q(\alpha) = \textstyle\sum_{i=0}^{\infty} \alpha(i) \times 2^{\perp(i+1)}$$

This defines a surjective function q from $\mathbf{3}^\omega$ to $\mathbb{I}$, where we write $\mathbb{I}$ for the closed interval $[\bot 1, 1]$. The whole real line can be represented using a mantissa from $\mathbf{3}^\omega$ and an exponent from $\mathbb{Z}$, thus $(z, \alpha) \in \mathbb{Z} \times \mathbf{3}^\omega$ represents the real number $2^z \times q(\alpha)$. This representation will be used in the full version of the paper, but, for lack of space, is not considered further in this conference version.

We use the natural type definition to implement the representation.

```
type interval = three stream
```

There is, however, a mismatch between the datatype and the representation of reals. We have that $[\![\texttt{interval}]\!] = \mathbf{3}^\infty$, whereas only elements in the subset $\mathbf{3}^\omega$ have been given interpretations as real numbers.

Just as not all values of type `interval` represent real numbers, neither do all functions of type `interval` $\to$ `interval` represent functions on real numbers. We use the denotational semantics to distinguish those that do. For greater generality we work with n-ary functions.

An arbitrary function $\phi : (\mathbf{3}^\infty)^n \to \mathbf{3}^\infty$ is said to be *total* if it restricts to a function $\bar{\phi} : (\mathbf{3}^\omega)^n \to \mathbf{3}^\omega$. Clearly $\bar{\phi}$, when it exists, is unique. Similarly, a function $\theta : (\mathbf{3}^\omega)^n \to \mathbf{3}^\omega$ is said to be *real* if there exists a function $\tilde{\theta} : \mathbb{I}^n \to \mathbb{I}$ such that, for all $\alpha_1, \ldots, \alpha_n \in \mathbf{3}^\omega$, it holds that $q(\theta(\alpha_1, \ldots, \alpha_n)) = \tilde{\theta}(q(\alpha_1), \ldots, q(\alpha_n))$. Again $\tilde{\theta}$ is uniquely determined (because q is surjective). Putting the two together, we say that $\phi : (\mathbf{3}^\infty)^n \to \mathbf{3}^\infty$ is *real-total* if it is total and $\bar{\phi}$ is real, in which case we write $\tilde{\phi} : \mathbb{I}^n \to \mathbb{I}$ for the unique induced function.

A functional program of type `interval` $\to$ `interval` will always be denoted by a *continuous* $\phi : \mathbf{3}^\infty \to \mathbf{3}^\infty$. By topological trivialities, if we endow $\mathbf{3}^\omega$ with the subspace topology of the Scott topology on $\mathbf{3}^\infty$, and we endow $\mathbb{I}$ with the quotient topology of $\mathbf{3}^\omega$ under q, then, for any continuous real-total ϕ, we have that $\bar{\phi}$ and $\tilde{\phi}$ are continuous. The proposition below makes this observation more interesting.

Proposition 1.

1. *The induced topologies on $\mathbf{3}^\omega$ and $\mathbb{I}$ are the product and Euclidean topologies respectively.*
2. *For any continuous $f : \mathbb{I}^n \to \mathbb{I}$, there exists a real $\theta : (\mathbf{3}^\omega)^n \to \mathbf{3}^\omega$ such that $f = \tilde{\theta}$.*
3. *For any continuous $\theta : (\mathbf{3}^\omega)^n \to \mathbf{3}^\omega$ there exists a total $\phi : (\mathbf{3}^\infty)^n \to \mathbf{3}^\infty$ such that $\theta = \bar{\phi}$.*

In the full version of the paper the definitions and results in this section will be related to work on totality in domain theory [2, 3, 16], and to topological injectivity (and projectivity) results [11].

4 Moduli of Continuity and Stream Quantifiers

Consider any continuous function $\phi : \mathbf{2}^\infty \to X_\bot$ where X is any set. We say that f is *total* if, for all $\alpha \in \mathbf{2}^\omega$, it holds that $\phi(\alpha) \in X$.

Proposition 2. *For any total* $\phi : \mathbf{2}^\infty \to X_\perp$ *there exists* $n \in \mathbb{N}$ *such that, for all* $\alpha \in \mathbf{2}^\infty$*, it holds that* $\phi(\alpha) = \phi(\alpha\lceil_n)$.

We call the least n satisfying the property stated in the proposition the *intensional modulus of continuity* of ϕ, and we write $imc(\phi)$ for it.

Corollary 1. *For any total* $\phi : \mathbf{2}^\infty \to X_\perp$ *there exists* $n \in \mathbb{N}$ *such that, for all* $\alpha, \beta \in \mathbf{2}^\omega$*, it holds that* $\alpha\lceil_n = \beta\lceil_n$ *implies* $\phi(\alpha) = \phi(\beta)$.

We call the smallest such n the *extensional modulus of continuity* of ϕ, and we write $emc(\phi)$ for it. Obviously $emc(\phi) \leq imc(\phi)$. In the full version of the paper there will be a discussion of the relative benefits of the two notions of modulus.

Our first application, due to Berger [2], is to provide a universal quantifier for total predicates on `two stream`. The algorithm is presented in Fig. 1 below.

```
witness-not: (two stream → bool) → two stream
witness-not (P) =
  lazylet w = witness-not (λv. P(0 :: v))
     in if P(0 :: w) then 1 :: witness-not (λv. P(1 :: v))
                     else 0 :: w

forall : (two stream → bool) → bool
forall (P) = P (witness-not P)
```

Fig. 1. Algorithms for the stream quantifier

Proposition 3. *For any total* $\phi : \mathbf{2}^\infty \to \mathbb{B}_\perp$*:*

$$[\![\texttt{forall}]\!](\phi) = \begin{cases} \mathit{true} & \mathit{if, for\ all}\ \alpha \in \mathbf{2}^\omega,\ \phi(\alpha) = \mathit{true} \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

Proof. One proves, by induction on $imc(\phi)$, that, for all total $\phi : \mathbf{2}^\infty \to \mathbb{B}_\perp$: if there exists $\alpha \in \mathbf{2}^\omega$ such that $\phi(\alpha) = \mathit{false}$ then $[\![\texttt{witness-not}]\!](\phi)$ is one such α; otherwise $[\![\texttt{witness-not}]\!](\phi) = \overrightarrow{\perp}$. The proposition follows easily. □

5 Functional Algorithms for Maximum and Integration

The denotation of every program of type `interval → interval` will be a continous function ϕ from $\mathbf{3}^\infty$ to $\mathbf{3}^\infty$. If ϕ is real-total then there is a corresponding continuous $\tilde{\phi} : \mathbb{I} \to \mathbb{I}$. Our goal in this paper is to show how Berger's algorithms can be applied to the practical problem of computing the values of functionals acting on continuous functions on $\mathbb{I}$.

We shall concentrate on two basic and useful functionals: the functional that finds the maximum value attained by a continuous function over the closed interval $[0, 1]$, and the function that computes the definite integral of a continuous function over $[0, 1]$. That such maximum values and definite integrals exist for all continuous functions are very basic results in analysis. Observe that both operations return values in $\mathbb{I}$.

5.1 Maximum

The algorithm for the functional `max-fun` is presented in Fig. 2. A first lemma states the important properties of the main auxiliary function defined there.

```
sub-one: interval → interval
sub-one (1 :: r)  = −1 :: r
sub-one (0 :: r)  = −1 :: sub-one(r)
sub-one (−1 :: r) = 1⃗

max-real: interval × interval → interval
max-real (d1 :: r1, d2 :: r2) =
  let d = d1 − d2 in case d of  2 then d1 :: r1
                                1 then d1 :: max-real(r1, sub-one(r2))
                                0 then d1 :: max-real(r1, r2)
                               −1 then d2 :: max-real(sub-one(r1), r2)
                               −2 then d2 :: r2

max-fun: (interval → interval) → interval
max-fun (f) =
 let d = head (f(1⃗)) in if forall (λv. head(f(v)) = d)
                        then d :: (max-fun(λv. tail(f(v))))
                        else max-real (max-fun (λv. f(0 :: v)),
                                       max-fun (λv. f(1 :: v)) )
```

Fig. 2. Maximum-value algorithm

Lemma 1. $[\![\texttt{max-real}]\!]$ *is real-total with:* $\widetilde{[\![\texttt{max-real}]\!]}(x, y) = max(x, y)$ *Moreover, for all* $\alpha, \beta \in \mathbf{3}^\infty$, $|[\![\texttt{max-real}]\!](\alpha, \beta)| \geq min(|\alpha|, |\beta|)$.

Observe that the lemma includes the intensional information that `max-real` only examines n digits of the input streams in order to produce n digits of output. This is crucial in the proof of the proposition below, which states the correctness of `max-fun`.

Proposition 4. *For any real-total* ϕ, *it holds that* $[\![\texttt{max-fun}]\!](\phi) \in \mathbf{3}^\omega$ *and*

$$q([\![\texttt{max-fun}]\!](\phi)) = max\{\tilde{\phi}(x) \mid 0 \leq x \leq 1\}.$$

To prove Proposition 4, we prove, by induction on $n \in \mathbb{N}$, that, for all real-total $\phi : \mathbf{3}^\infty \to \mathbf{3}^\infty$, it holds that $[\![\texttt{max-fun}]\!](\phi)\lceil_n = d_1 \ldots d_n \in \mathbf{3}^n$ such that:

$$\mid max\{\tilde{\phi}(x) \mid 0 \leq x \leq 1\} \perp \sum_{i=1}^{n} d_i . 2^{\perp i} \mid \;\leq\; 2^{\perp n} \tag{1}$$

The base case, $n = 0$, is trivial. When $n > 0$, consider $h(\phi) : \mathbf{2}^\infty \to \mathbf{3}_\perp$ defined by $h(\phi)(\alpha) = hd(\phi(\alpha))$. Because ϕ is real-total, we have that $h(\phi)$ is total. The required inequality (1) is now proved by an inner induction on $emc(h(\phi))$.

Briefly, if $emc(h(\phi)) = 0$ then (1) is proved using the outer induction hypothesis on n and the general equality, valid for any continuous $f : \mathbb{I} \to \mathbb{I}$:

$$max\{f(x) + c \mid 0 \leq x \leq 1\} = max\{f(x) \mid 0 \leq x \leq 1\} + c. \tag{2}$$

When $emc(h(\phi)) > 0$ then (1) is proved using the induction hypothesis on the extensional modulus of continuity (the intensional information of Lemma 1 is needed) together with the general equality, valid for any continuous $f : \mathbb{I} \to \mathbb{I}$:

$$\begin{aligned} max\{f(x) \mid 0 \leq x \leq 1\} = max(\ & max\{f(x/2) \mid 0 \leq x \leq 1\}, \\ & max\{f((x+1)/2) \mid 0 \leq x \leq 1\}\ ). \end{aligned} \tag{3}$$

5.2 Integration

Integration can be performed by much the same method. Observe that integration enjoys the following equalities, for any continuous $f : \mathbb{I} \to \mathbb{I}$:

$$\int_0^1 f(x) + c \ \mathrm{d}x = \int_0^1 f(x) \ \mathrm{d}x + c$$

$$\int_0^1 f(x) \ \mathrm{d}x = \int_0^1 f(x/2) \ \mathrm{d}x \ \oplus \ \int_0^1 f((x+1)/2) \ \mathrm{d}x,$$

where $\oplus : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ computes the average of two reals. The above equations are wholly analogous to (2) and (3) for maximum. Indeed we shall obtain an integration algorithm by replacing the binary `max-real` used in `max-fun` with a function computing the average of two reals. However, the translation is not completely straightforward. Recall that the intensional information of Lemma 1 was crucial to the proof of Proposition 4. This contrasts with the easy:

Proposition 5. *There is no real-total $\phi : \mathbf{3}^\infty \to \mathbf{3}^\infty$ such that, for all $x, y \in \mathbb{I}$, $\tilde{\phi}(x, y) = x \oplus y$ and, for all $\alpha, \beta \in \mathbf{2}^\infty$, $|\phi(\alpha, \beta)| \geq min(|\alpha|, |\beta|)$.*

The observed problem is a quirk of the particular representation of real numbers we are using. A neat way of solving it is to use a second representation. Recall that the set of dyadic rationals is $\mathbb{Q}_d = \{m/2^n \mid m, n \in \mathbb{Z}\}$. We write $\mathbb{D}$ for the set $\mathbb{Q}_d \cap [\bot 1, 1]$, which we call the set of *dyadic digits*. We consider an infinite sequence $\gamma \in \mathbb{D}^\omega$ as representing the real number $q'(\gamma) = \sum_{i=0}^{\infty} \gamma(i) \times 2^{\bot(i+1)}$. This defines a surjective function $q' : \mathbb{D}^\omega \to [\bot 1, 1]$ extending $q : \mathbf{3}^\omega \to [\bot 1, 1]$.

In order to write algorithms working with dyadic digits we assume an implemented datatype `dyadic` of dyadic digits, complete with the associated operations for the basic arithmetic operations on dyadic rationals. Then we simply define a new datatype for the interval $[\bot 1, 1]$ in terms of dyadic streams:

```
type q-interval = dyadic stream
```

Semantically we assume that $[\![\texttt{dyadic}]\!] = \mathbb{D}_\bot$, so $[\![\texttt{q-interval}]\!] = \mathbb{D}^\infty$. The notions of a function $\phi : \mathbb{D}^\infty \to \mathbb{D}^\infty$ being *total* and *real-total* are defined entirely analogously to the cases for $\mathbf{3}^\infty$.

The full algorithm for integration is presented in Fig. 3. For convenience we assume that `three` is a subtype of `dyadic` and (hence) `interval` is a subtype of `q-interval`.

```
coerce: q-interval → interval
coerce (qd1 :: qd2 :: qr) =
 let qc = (2×qd1)+qd2 in case qc < −1   then −1::coerce((qc+2) :: qr)
                              qc > 1    then  1::coerce((qc−2) :: qr)
                              otherwise then  0::coerce(qc :: qr)

q-avg: q-interval × q-interval → q-interval
q-avg (qd1 :: qr1, qd2 :: qr2) = (qd1+qd2)/2 :: q-avg (qr1, qr2)

q-int: (interval → interval) → q-interval
q-int (f) = let d = head (f(1⃗)) in if forall (λv. head(f(v)) = d)
                                    then d :: (q-int(λv. tail(f(v))))
                                    else q-avg (q-int (λv. f(0 :: v)),
                                                q-int (λv. f(1 :: v)))

integrate: (interval → interval) → interval
integrate (f) = coerce (q-int(f))
```

Fig. 3. Integration algorithm

Lemma 2.

1. *For any* $\gamma \in \mathbb{D}^\omega$, *it holds that* $[\![\texttt{coerce}]\!](\gamma) \in \mathbf{3}^\omega$ *and* $q([\![\texttt{coerce}]\!](\gamma)) = q'(\gamma)$.
2. *The function* $[\![\texttt{q-avg}]\!] : \mathbb{D}^\infty \to \mathbb{D}^\infty$ *is real-total with* $[\![\texttt{q-avg}]\!](x, y) = x \oplus y$. *Moreover, for all* $\gamma, \gamma' \in \mathbb{D}^\infty$, $|[\![\texttt{q-avg}]\!](\gamma, \gamma')| \geq min(|\gamma|, |\gamma'|)$.

Proposition 6. *For any real-total* ϕ, *it holds that* $[\![\texttt{integrate}]\!](\phi) \in \mathbf{3}^\omega$ *and*

$$q([\![\texttt{integrate}]\!](\phi)) = \int_0^1 \tilde{\phi}(x)\ dx.$$

The proof structure closely follows that of Proposition 4.

6 Further Developments

In the full version of the paper an extension of the integration algorithm will be presented that integrates, over any closed interval, functions defined from the interval to the whole real line. This makes use of the mantissa-exponent representation of the real line mentioned briefly in Sec. 3.

The algorithms in this extended abstract were implemented by Reinhold Heckmann in Gofer in summer 1997. The extensions to functions from an arbitrary closed interval to the full real line have recently been implemented in Haskell by David Plume. The integration algorithm performs abysmally on any interesting functions. The maximum algorithm performs a little better. A partial quantitative analysis of this situation will appear in the full version of the paper. The intrinsic intractibility of the operations of integration and finding maximum values is to be expected from the work of Ko [13].

Acknowledgements

I have benefited from discussions with Pietro Di Gianantonio, Gordon Plotkin and, especially, Martín Escardó. I thank Ieke Moerdijk, Jaap van Oosten and Harold Schellinx for their hospitality in Utrecht, where the paper was written with financial support from an NWO Pionier Project.

References

1. J.-C. Bajard, D. Michelucci, J.-M. Moreau, and J.-M. Muller. Introduction to special issue: "Real Numbers and Computers". *Journal of Universal Computer Science*, 1(7):436–438, 1995.
2. U. Berger. *Totale Objecte und Mengen in Bereichstheorie*. PhD Thesis, University of Munich, 1990.
3. U. Berger. Total objects and sets in domain theory. *Journal of Pure and Applied Logic*, 60:91–117, 1993.
4. H.J. Boehm and R. Cartwright. Exact real arithmetic: Formulating real numbers as functions. In D. Turner, editor, *Research Topics in Functional Programming*, pages 43–64. Adison-Wesley, 1990.
5. H.J. Boehm, R. Cartwright, M. Riggle, and M.J. O'Donnel. Exact real arithmetic: a case study in higher order programming. In *ACM Symposium on LISP and Functional Programming*, 1986.
6. P. Di Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD Thesis, University of Pisa, 1993.
7. P. Di Gianantonio. An abstract data type for real numbers. In *Proceedings of ICALP-97*, pages 121–131. Springer LNCS 1256, 1997.
8. A. Edalat and M.H. Escardó. Integration in Real PCF. *Information and Computation*, To appear, 1998.
9. A. Edalat and P.J. Potts. *Exact Real Computer Arithmetic*. Presented at workshop: New Paradigms for Computation on Classical Spaces, Birmingham, 1997.
10. M.H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, 1996.
11. M.H. Escardó. Properly injective spaces and function spaces. *Topology and its Applications*, To appear, 1998.
12. C.A. Gunter. *Semantics of Programming*. MIT Press, 1992.
13. Ker-I Ko. *Complexity Theory of Real Functions*. Birkhauser, Boston, 1991.
14. V. Menissier-Morain. Arbitrary precision real arithmetic: Design and algorithms. *Journal of Symbolic Computation*, Submitted, 1996.
15. G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(1):223–255, 1977.
16. G.D. Plotkin. Full abstraction, totality and PCF. *Math. Struct. in Comp. Sci.*, To appear, 1998.
17. J. Vuillemin. Exact real arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, 1990.