Coinductive Field of Exact Real Numbers and General Corecursion

Milad Niqui [1,2]

Institute for Computing and Information Sciences, Faculty of Science
Radboud University Nijmegen
Toernooiveld 1, 6525 ED, Nijmegen, The Netherlands

Abstract

In this article we present a method to define algebraic structure (field operations) on a representation of real numbers by coinductive streams. The field operations will be given in two algorithms (homographic and quadratic algorithm) that operate on streams of Möbius maps. The algorithms can be seen as coalgebra maps on the coalgebra of streams and hence they will be formalised as *general corecursive functions*. We use the machinery of *Coq* proof assistant for coinductive types to present the formalisation.

Key words: Coinductive types, exact real arithmetic, general corecursion

1 Introduction

Coalgebras present a suitable semantics for working with infinite phenomena where only partial observations are available. Various reincarnation of infinite objects such as real numbers, labelled transition systems, object oriented modularity and dynamical systems can be studied in the framework of coalgebras. The case of real numbers is particularly interesting because of various reasons. First, the ubiquity and theoretical importance of real numbers makes them an important candidate for showing the expressibility of the framework of coalgebras. This will pave the way for classifying computable coalgebra maps which in turn help elevate the status of coalgebraic semantics for programming with infinite objects. Furthermore it enables us to use coalgebraic semantics for *exact real arithmetic*, a precision driven approach to real number computation whose applications in the field of numerical computations is increasing. On the other hand, specific properties of real numbers (such as

[1] Research supported by the Netherlands Organisation for Scientific Research (NWO).
[2] Email: M.Niqui@cs.ru.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the need to have a redundant representation) once translated into coalgebraic language give valuable insight into various notions of coinduction principles and bisimulation equivalence that can be of use elsewhere.

The final coalgebras corresponding to real numbers have relatively simple structure. In the literature they are usually presented as the final coalgebra of the *list functor* $X \longmapsto 1+A\times X$ or the *stream functor* $X \longmapsto A\times X$ [29,28] and as the final coalgebra of the *expression trees with A-elements and B-operations functor* $X \longmapsto A \times X + B \times X^2$ by the author [23,24]. However, such a simple structure in itself does not capture the full power of mathematical analysis: one needs to be able to equip the coalgebra of reals with algebraic structure (field operations) and order structure (Cauchy completeness).

Pavlović and Pratt [29] study the order properties of the continuum as the final coalgebra for the list functor and stream functor in category **Set** by specifying Cantor space and Baire space in terms of these functors. However, by characterising the continuum only up to its order type, their construction does not address the algebraic properties of real numbers.

Freyd gives another characterisation of Dedekind reals (see [20, § D4.7]) in terms of the diagonalisation of a 'wedge' functor in a category of posets. This idea is pursued further by Escardó and Simpson [15], where they characterise the Cauchy completeness of rationals within Dedekind reals obtaining a universal property that allows for an analogous of primitive recursion. These characterisations recover some of the algebraic properties of the real numbers while relying on topos logic to achieve this.

In the present work we plan to achieve a similar goal: a coalgebraic view of real numbers in which computation is done very close to the actual implementations of exact arithmetic. We start by expressing some algorithms on stream of real numbers by giving their specification in a syntax based on *Haskell* programming language, and would like to find their corresponding coalgebra maps. But this task is not easy mainly due to the problem of *productivity* [11]. Productive functions are those functions on infinite objects that produce provably infinite output.

The basic *coiteration scheme* which is obtained from the universality of final coalgebra can be used to directly define plenty of standard stream functions; but it has its limitations [31]. To overcome this restriction several more powerful schemes has been proposed each including an ever expanding class of specifications. Among these schemes are *corecursion* [16], *dual of course of value recursion* [31], *T-coiteration for pointed functors* [21], *λ-coiteration for distributive laws* [2] and *bialgebraic T-coiteration* [6]. But these schemes have one thing in common: they all impose some syntactic criterion for the class of specifications that they are capable of handling; while the productivity of the algorithms on real numbers cannot be syntactically detected. In fact the productivity of the standard filter function on stream of natural numbers with the following specification is also not decidable (here P is a boolean predicate

on natural numbers).

$$\mathsf{filter}\quad (x\colon xs) := \begin{cases} x\colon \quad \mathsf{filter}\quad xs & \textbf{if } P(x)\ , \\ \mathsf{filter}\quad xs & \textbf{otherwise.} \end{cases}$$

By suitably choosing P one can reduce the problem of the productivity of the above function to an open problem in mathematics; see [23, Example 4.7.6] for a choice of P which shows that the productivity of the above function is equivalent to whether there are infinitely many twin prime numbers.

Therefore it seems that providing syntactic productivity tests cannot cover the most general class of recursive specifications for infinite objects. Hence one has to adhere to semantic means in order to be able to use the framework of coalgebras for programming in general, and for programming on real numbers in particular. For instance, for the case of filter on prime numbers, one has to (1) consider a number theoretic constructive proof of the infinitude of primes, (2) from this proof extract a function $\varkappa$ that returns the nth prime number, (3) use $\varkappa$ to rewrite filter in a way that it passes syntactic tests of productivity, i.e., using one of the above syntactic schemes [23, § 4.7].

Formalising step (1) in the language of coalgebras requires one to either restrict itself to categories where some notion of logic or reasoning is present. There are several categorical ways to achieve this: one is to work in a category where the proofs of continuity and approximation are built-in [26]. Another, more general way is to work in a topos and use the internal logic. This is similar to the approach taken by Freyd et al [20,15]. The other closely related possibility would be to work in a category of types and use constructive type theory where propositions and computational objects coexist. This way one can use the machinery of constructive type theory for reasoning and formalising the 'semantic' proofs of productivity, themselves objects of the category.

In this article we take the last approach, i.e., we present a coalgebraic formalisation of our algorithms in a version of constructive type theory extended by coinductive types. Coinductive types are added to type theory for dealing with infinite objects [22]. Usually, in the categories modelling type theory, they are interpreted as (weakly)[3] final coalgebras. Weak finality is used for intensional type theories, where uniqueness cannot be expressed by reduction rules. Working extensionally, or in a setoid category (with bisimulation as setoid equality), one can formulate the uniqueness property [7, p. 74].

Our works is similar to the formalisation in [8,3] where the stream of real numbers is formalised as a coinductive type together with algebraic structure (field operations). However our works is different in many aspects: we use the more general setting of Edalat et al [14] for simultaneously defining algebraic operations of $+,\times,-$ and division in one algorithm (*quadratic* algorithm). As a result our method relies on formalising a non-syntactically productive function

[3] A final coalgebras with the uniqueness property dropped.

for which we use a method that we call *general corecursion*. This is related to (but different from) the method presented by Bertot in [4] for formalising Eratosthenes' sieve. Moreover, our formalisation of the homographic and quadratic algorithm are the first step towards the formalisation of the very powerful *normalisation algorithm* of Edalat and Potts [14,30,24] that gives all the elementary functions on real numbers.

The material in this article has been implemented in *Coq* proof assistant. We start by presenting the implementation of coinductive types in *Coq* in Section 2. In Section 3 we present the homographic and quadratic algorithm as *Haskell*-like specifications. In Section 4 we present the formalisation of those algorithms as a coalgebra map on the coalgebra of streams, using the general corecursion method. In Section 5 we conclude the article by presenting some directions for further research. Throughout the article we use a syntax loosely based on *Coq* syntax, adapted for presenting in an article. The actual code of the formalisation including the proof of all necessary *Coq* lemmas are available online at [25].

2 Coinductive Types in *Coq*

Coq proof assistant [9] is an implementation of Calculus of Inductive Constructions (CIC) extended with coinductive types. This is is an extension of Martin-Löf intensional type theory. Coinductive types were added to *Coq* by Giménez [18]. Their implementation follows the same philosophy as that of inductive types in CIC, namely there is a general scheme that allows for formation of coinductive types if their *constructors* are given, and if these constructors satisfy the *strict positivity* condition. The definition of strictly positive constructor is identical for inductive and coinductive types and similar to that of a monomial endofunctor (i.e., an endofunctor involving products and exponentials). Intuitively a constructor c is strictly positive with respect to x if x does not appear to the left of an $\rightarrow$ in c. A formal definition can be found in [27]. This means that the following forms an inductive (resp. coinductive) type I in *Coq*, provided that the keyword `Indutive` (resp. `CoInductive`) is given and that all c_i's are strictly positive constructors with respect to I.

$$
\begin{array}{l}
\texttt{(Co)Inductive}\ I\ \ \texttt{(}x_1\texttt{:}\ X_1\texttt{)}\ldots\texttt{(}x_i\texttt{:}\ X_i\texttt{)}\texttt{:}\ \forall\texttt{(}y_1\texttt{:}\ Y_1\texttt{)}\ldots\texttt{(}y_m\texttt{:}\ Y_m\texttt{)},\ s\texttt{:=} \\
\texttt{|}\,c_1\texttt{:}\ \forall\texttt{(}z_{11}\texttt{:}\ Z_{11}\texttt{)}\ldots\texttt{(}z_{1k_1}\texttt{:}\ Z_{1k_1}\texttt{)},\ I\ \ t_{11}\ \ \ldots\ \ t_{1m} \\
\qquad\quad \vdots \\
\texttt{|}\,c_n\texttt{:}\ \forall\texttt{(}z_{n1}\texttt{:}\ Z_{n1}\texttt{)}\ldots\texttt{(}z_{nk_n}\texttt{:}\ Z_{nk_n}\texttt{)},\ I\ \ t_{11}\ \ \ldots\ \ t_{nm}\texttt{.}
\end{array}
$$

In such a declaration s is a sort,i.e., $s \in \{\,\mathsf{Set}, \mathsf{Prop}, \mathsf{Type}\,\}$. Moreover x_is (resp. y_is) are *general* (resp. *recursive*) parameters of I.

For example one can define the final coalgebra of streams as

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

Note that this is a polymorphic type forming the streams of elements of its general parameter.

After a coinductive type is defined one can introduce its inhabitants and functions into it (i.e., elements of the final coalgebra). Such definitions are given by a *cofixed point* operator. This is an operator similar to the fixed point operator for structural recursion. This operator when given a well-typed definition that satisfies a *guardedness condition* will introduce an element of the weakly final coalgebra.

The typing rule for this operator is given by the following judgement (here, let I be a coinductive type with parameters $P_0, \ldots, P_i$).

$$\textsf{cofix}\ \textit{rule} \qquad \frac{\Gamma, f\colon B \vdash N\colon B \quad B \equiv \forall x_0\colon X_0, \ldots, x_j\colon X_j, (I \quad P_0 \quad \ldots \quad P_i) \quad \mathsf{G}(f, B, N)}{\Gamma \vdash \textsf{cofix}\ f\colon B := N\colon B}$$

According to this rule, if f, B and N satisfy the side condition G then $\textsf{cofix}\ f$ is an inhabitant of type B which is a coinductively defined type. In this case N is the body of the definition which may contain f. The side condition $\mathsf{G}(f, B, N)$ is called the *Coq* guardedness condition and is yet another syntactic criterion that is intended to ensure the productivity of infinite objects. This condition checks whether the declaration of f is guarded by constructors of I. This means that every occurrence of f in the body of f should be direct argument of one of the constructors of I. This condition is due to Giménez [18] and is based on earlier work of Coquand [10]. A precise definition of G can be found in [18, p. 175]. Like other syntactic extensions of coiteration scheme, the guardedness condition of *Coq* is too restrictive a requirement to allow for formalisation of all productive functions.

Finally we mention the reduction (in fact expansion) rule corresponding to the $\textsf{cofix}$ operator. Let $F \equiv \textsf{cofix}\ g\colon B := N$. Then the *cofixed point expansion* is the following rule.

$$\begin{array}{l}\textbf{match}\ (F \quad P_0 \ldots P_j)\colon X\ \textbf{with}\ \mid r_0 \Rightarrow R_0 \mid \ldots \mid r_k \Rightarrow R_k\ \textbf{end} \rightsquigarrow \\ \quad \textbf{match}\ (N[g \leftarrow F] \quad P_0 \ldots P_j)\colon X\ \textbf{with}\ \mid r_0 \Rightarrow R_0 \mid \ldots \mid r_k \Rightarrow R_k\ \textbf{end}\ .\end{array}$$

Thus, the expansion of a cofixed point is only allowed when a case analysis of the cofixed point is done.

From a coalgebraic point of view this treatment of coinductive types by means of constructors and cofixed point operator might seem unnatural: final coalgebras are about observations and not constructions; final coalgebra should be given using its destructor. Nevertheless, presenting the coinductive types in the *Coq* way, is much closer to the syntax of lazy functional program-

ming languages such as *Haskell* [4] and hence very useful for many applications. Moreover, as we show in Section 4, one can use *Coq* to define a general form of productive functions, allowing one to build more complicated coalgebraic structures. In any case, theoretically this does not change the coalgebraic semantics and the coinductive types can still be interpreted as weakly final coalgebras in any categorical model of CIC (see [1] where a stronger results is proven). Furthermore, the usual coiteration and corecursion schemes can be derived in terms of cofix operator [17]. Therefore in this article we present our method using the language of *Coq* with the understanding that it can be easily translated in the standard categorical notations in any categorical model of CIC [5].

3 Homographic and Quadratic algorithm: Specification

In this section we present the algorithms for field operations on a stream representation for compactification of positive real numbers, i.e., (extended) real numbers in $[0, +\infty]$. The algorithms are similar to Gosper's algorithm [19] for addition and multiplication on continued fractions. They also form the basis of Edalat and Potts approach to lazy exact real arithmetic [14,30].

Here we use a representation which is much simpler than continued fractions and is redundant enough to ensure the productivity. In fact we could abstract away both the digit set and the compact subinterval of $[-\infty, +\infty]$ that we use but then the algorithms would become too higher order. A treatment of the general case can be found in [23, § 5]. Thus, for presentational purposes, we consider a fixed representation for $[0, +\infty]$ containing 3 digits, each of which a Möbius map. *Möbius maps* are maps of the form

$$x \longmapsto \frac{ax + b}{cx + d} \ ,$$

where $a, b, c, d \in \mathbb{Z}$ and $ad - bc \neq 0$. A Möbius map is *refining* if it maps the closed interval $[0, +\infty]$ to itself. Möbius maps are usually denoted by the matrix of their coefficients.

For our representation, we consider the set $\mathbf{DIG} = \{\, \mathbf{L}, \mathbf{R}, \mathbf{M} \,\}$ and denote the set of streams over **DIG** by $\mathbf{DIG}^{\omega}$. We interpret each digit by a refining Möbius map as follows.

$$\mathbf{L} = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right] \ , \quad \mathbf{R} = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right] \ , \quad \mathbf{M} = \left[\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right] \ .$$

[4] Note that in *Haskell*— where there is no distinction between inductive and coinductive types— all data-types can be considered to be potentially infinite and hence correspond to *Coq* coinductive types.

[5] In fact, in the present article we do not need the *universes* in CIC and therefore categorical models of simpler extensions of Martin-Löf type theory — such as Martin-Löf categories of Abbott et al [1] — will suffice.

The fact that $\mathbf{DIG}^\omega$ is a representation (based on the Stern–Brocot tree) for $[0,+\infty]$ with enough redundancy is proven in [23, § 5.7]. This means that there exists a *representation* (i.e., a total surjective map) ρ from $\mathbf{DIG}^\omega$ to $[0,+\infty]$ such that for all $f_0 f_1 \cdots \in \mathbf{DIG}^\omega$ we have

$$\{\, \rho(f_0 f_1 \ldots) \,\} = \bigcap_{i=0}^{\infty} f_0 \circ \ldots f_i([0,+\infty]) \ .$$

In what follows we assume $\mu = \left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ to be a refining Möbius map . The *homographic algorithm* is the algorithm that given μ and a stream $\alpha \in \mathbf{DIG}^\omega$ outputs a stream δ such that $\mu(\rho(\alpha)) = \rho(\delta)$. In order to present the homographic algorithm we need to define an *emission condition* for a digit $d = \left[\begin{smallmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{smallmatrix}\right] \in \mathbf{DIG}$ and μ as the following predicate

$$\mathbf{Incl}(\mu, \mathrm{d}) := ad_{21} \leq cd_{11} \wedge dd_{12} \leq bd_{22} \wedge cd_{12} \leq ad_{22} \wedge bd_{21} \leq dd_{11} \ .$$

The intuition is that we are checking the inclusion of intervals $\mu([0,+\infty]) \subseteq d([0,+\infty])$ which are two intervals with rational endpoints (considering $+\infty = 1/0$ as a pseudofraction). This predicate enables us to state the homographic algorithm:

$$\mathsf{homographic} \quad \mu \quad (x\colon xs) := \begin{cases} \mathbf{L}\colon \mathsf{homographic} \quad (\mathbf{L}^{-1} \circ \mu) \quad (x\colon xs) & \textbf{if } \mathbf{Incl}(\mu, \mathbf{L}) \ , \\ \mathbf{R}\colon \mathsf{homographic} \quad (\mathbf{R}^{-1} \circ \mu) \quad (x\colon xs) & \textbf{else if } \mathbf{Incl}(\mu, \mathbf{R}) \ , \\ \mathbf{M}\colon \mathsf{homographic} \quad (\mathbf{M}^{-1} \circ \mu) \quad (x\colon xs) & \textbf{else if } \mathbf{Incl}(\mu, \mathbf{M}) \ , \\ \mathsf{homographic} \quad \mu \circ x \quad xs & \textbf{otherwise.} \end{cases}$$

Here d^{-1} and $\circ$ denote the usual matrix inversion and matrix product. The intuition behind the algorithm is that we start by considering an infinite product of Möbius maps, of which all but the first one are digits. We start pushing μ towards infinity by absorbing digits (hence obtaining a new refining Möbius map) and emitting digits whenever the emission condition holds, i.e., whenever the range of Möbius map applied to the interval $[0,+\infty]$ fits inside the range of a digit.

$$\mu \circ d_0 \circ d_1 \circ \cdots \quad \rightsquigarrow \quad d \circ (d^{-1} \circ \mu) \circ d_0 \circ d_1 \circ \cdots \quad \textbf{if } \mathbf{Incl}(\mu, \mathrm{d}) \ .$$

For a more formal semantics for the algorithm see the semantical proof of correctness that is given in [23, § 5.6].

To compute field operations we consider the *quadratic map* which is a map

$$\xi(x,y) := \frac{axy + bx + cy + d}{exy + fx + gy + h} \ ,$$

with $a, b, c, d, e, f, g \in Z$ and can be denoted by its $2 \times 2 \times 2$ tensor of coefficients. A quadratic map is *nonsingular* if all of the six matrices that constitute

the faces of the tensor of coefficients are nonsingular. A *refining* quadratic map is a nonsingular quadratic map ξ such that $\xi([0,+\infty],[0,+\infty]) \subseteq [0,+\infty]$.

In the remainder of this section we assume $\xi = \left[\begin{smallmatrix} a & b & c & d \\ e & f & g & h \end{smallmatrix}\right]$ is a refining quadratic map. The *quadratic algorithm* is an algorithm that given ξ and two streams $\alpha, \beta \in \mathbf{DIG}^\omega$, outputs a stream δ such that $\xi(\rho(\alpha), \rho(\beta)) = \rho(\delta)$. The algorithm uses an emission condition for quadratic maps that is given below.

$$\begin{aligned}\mathbf{Incl}(\xi, \mathrm{d}) := ed_{12} \leq ad_{22} \wedge fd_{12} \leq bd_{22} \wedge gd_{12} \leq cd_{22} \wedge hd_{12} \leq dd_{22} \wedge \\ ad_{21} \leq ed_{11} \wedge bd_{21} \leq fd_{11} \wedge cd_{21} \leq gd_{11} \wedge dd_{21} \leq hd_{11}.\end{aligned}$$

We use the notation $d \circ \xi$ to denote the composition of a Möbius map d and a quadratic map ξ (note that the outcome is again a quadratic map). Moreover we use $\xi \bullet_1 d$ and $\xi \bullet_2 d$ to denote the two different ways of composing a quadratic map and a Möbius map by considering the Möbius map as its first (resp. second) argument. With this notation we can present the quadratic algorithm:

$$\begin{aligned}&\mathsf{quadratic}\ \ \xi\ \ (x\colon xs)\ \ (y\colon ys) := \\ &\begin{cases} \mathbf{L}\colon \mathsf{quadratic}\ \ (\mathbf{L}^{-1} \circ \xi)\ \ (x\colon xs)\ \ (y\colon ys) & \textbf{if } \mathbf{Incl}(\xi, \mathbf{L})\ , \\ \mathbf{R}\colon \mathsf{quadratic}\ \ (\mathbf{R}^{-1} \circ \xi)\ \ (x\colon xs)\ \ (y\colon ys) & \textbf{else if } \mathbf{Incl}(\xi, \mathbf{R})\ , \\ \mathbf{M}\colon \mathsf{quadratic}\ \ (\mathbf{M}^{-1} \circ \xi)\ \ (x\colon xs)\ \ (y\colon ys) & \textbf{else if } \mathbf{Incl}(\xi, \mathbf{M})\ , \\ \mathsf{quadratic}\ \ (\xi \bullet_1 x \bullet_2 y)\ \ xs\ \ ys & \textbf{otherwise.} \end{cases}\end{aligned}$$

The intuition behind this algorithm is similar to the homographic algorithm. Both homographic and quadratic algorithm are the base case of normalisation algorithm of Edalat and Potts [30,23]. Homographic algorithm can be used to compute linear fractional transforms of a real number, while quadratic map can be used for binary field operations (e.g. taking $\xi := \left[\begin{smallmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{smallmatrix}\right]$ it gives the multiplication).

Note also that here we are concerned with the (extended) positive reals. Adding a (redundant) sign bit is quite straightforward and enables the computation on entire real line [30].

4 Homographic and Quadratic algorithm: General Corecursive Version

Algorithms of the previous section specify morphisms on the final coalgebra of streams. Translating this specification into the categorical language means that we should ensure that the codomain is indeed the final coalgebra, i.e., the outcome of the algorithms is an infinite stream. Both algorithms resemble the general shape of the filter algorithm (see Section 1). Hence the usual syntactic schemes do not suffice. What we need is to incorporate a categorical object that captures the semantic proof of the infinitude of the outcome. Such a

proof relies on the fact that at every step in the algorithm after absorbing a finite number of digits the emission condition certainly holds and hence we output a digit [23, Lemma 5.6.10]. We plan to capture this inside a recursive function that at each step outputs the next digit, serving as a modulus for productivity. The original algorithms will then call this function at every step to obtain the next digit while keeping track of the new arguments that should be passed to future step. This idea is used by Bertot [4] to give a general method for defining filter in *Coq*. In this section we apply a modification of Bertot's method for our algorithms of exact arithmetic.

4.1 Homographic Algorithm

We work in a categorical model of CIC. Let $\mathbb{M}$ (resp. $\mathbb{T}$) be objects denoting the set of Möbius maps (resp. quadratic maps) in this category [6]. We are seeking to define a coalgebra map $h\colon \mathbb{M} \times \mathbf{DIG}^\omega \longrightarrow \mathbf{DIG}^\omega$, that corresponds to the homographic algorithm. But h is a partial function and might not be productive at every point. So instead of defining h we shall define a map $\bar{h}\colon \Pi(\mu\colon \mathbb{M})(\alpha\colon \mathbf{DIG}^\omega).P_h(\mu,\alpha) \longrightarrow \mathbf{DIG}^\omega$ where $P_h(\mu,\alpha)$ is a predicate (i.e., a term of type Prop) with the intended meaning that the specification of homographic algorithm is productive when applied to μ and α. In other words it specifies the domain of the partial function h.

The definition of P_h is based on the modulus of productivity. This modulus is a recursive function $m_h\colon \mathbb{M}\times\mathbf{DIG}^\omega \longrightarrow \mathbf{DIG}\times\mathbb{M}\times\mathbf{DIG}^\omega$ with the intended meaning that $m_h(\mu,\alpha) = \langle d, \langle \mu', \alpha'\rangle\rangle$ if and only if

$$\mathsf{homographic}\quad \mu\quad \alpha \qquad \rightsquigarrow \qquad d\colon \mathsf{homographic}\quad \mu'\quad \alpha'\ ,$$

where '$\rightsquigarrow$' denotes multiple reduction steps after which d is output (so after output of d there are no more digits absorbed in μ'). We would like this to be a function with recursive calls on α, but this is not possible. The reason is that α is an element of the final coalgebra while in the recursion scheme we need an element of an initial algebra. In other words we need to accommodate the domain of the function m_h with an inductively defined argument which will be used for recursive calls.

This situation is similar to the case of partial recursive functions or recursive functions with non-structurally recursive arguments. In order to formalise such function in constructive type theory, there is a method of adding an inductive domain predicate introduced in [12] and extensively developed by Bove and Capretta [5]. According to this method we need to define an inductively defined predicate $E_h(\mu,\alpha)$ with the intended meaning that μ and α are in the domain of m_h which in turn means that the homographic algorithm should emit at least one digit when applied on μ and α.

[6] They can be considered as $\mathbb{Z}^4$ and $\mathbb{Z}^8$ respectively, forgetting about the refining and nonsingular properties. Adding nonsingularity and refining criteria is trivial by considering records (Σ-types).

We define E_h as the initial algebra for the following strictly positive Prop-valued functor

$$\begin{aligned}\mathrm{F}(X) := \Pi(\mu:\mathbb{M})(\alpha:\mathbf{DIG}^\omega), \mathbf{Incl}(\mu,\mathbf{L}) + \\ \Pi(\mu:\mathbb{M})(\alpha:\mathbf{DIG}^\omega), \mathbf{Incl}(\mu,\mathbf{R}) + \\ \Pi(\mu:\mathbb{M})(\alpha:\mathbf{DIG}^\omega), \mathbf{Incl}(\mu,\mathbf{R}) + \\ \Pi(\mu:\mathbb{M})(\alpha:\mathbf{DIG}^\omega), \neg\mathbf{Incl}(\mu,\mathbf{L}) \times \neg\mathbf{Incl}(\mu,\mathbf{R}) \times \neg\mathbf{Incl}(\mu,\mathbf{M}) \times \\ X(\mu\circ(\mathsf{hd}\alpha))(\mathsf{tl}\alpha)\ .\end{aligned}$$

In the *Coq* syntax this would be given as

```
Inductive E_h: M → DIG^ω → Prop :=
|E_hL: ∀(μ: M)(α: DIG^ω), Incl(μ, L) → E_h μ α
|E_hR: ∀(μ: M)(α: DIG^ω), Incl(μ, R) → E_h μ α
|E_hM: ∀(μ: M)(α: DIG^ω), Incl(μ, M) → E_h μ α
|E_hab: ∀(μ: M)(α: DIG^ω),¬Incl(μ, L) → ¬Incl(μ, R) → ¬Incl(μ, M) →
                                E_h (μ∘(hd α)) (tl α) → E_h μ α.
```

Here hd and tl are destructors of the stream coalgebra and E_{hL}, E_{hR}, E_{hM} and E_{hab} are constructors of E_h. This allows us to define the modulus of productivity, i.e., a recursive function $\overline{m}_h : \Pi(\mu:\mathbb{M})(\alpha:\mathbf{DIG}^\omega).E_h(\mu,\alpha) \longrightarrow \mathbf{DIG}\times\mathbb{M}\times\mathbf{DIG}^\omega$ as follows.

```
Fixpoint m̄_h(μ: M)(α: DIG^ω)(t: E_h μ α){struct t}: DIG*(M*DIG^ω):=
match Incl_dec(μ, L) with
| left _⇒ ⟨L, ⟨L^-1 ∘ μ, α⟩⟩
| right t_l⇒
   match Incl_dec(μ, R) with
   | left _⇒ ⟨R, ⟨R_-1 ∘ μ, α⟩⟩
   | right t_r⇒
      match Incl_dec(μ, M) with
      | left _⇒ ⟨M, ⟨M_-1 ∘ μ, α⟩⟩
      | right t_m⇒ m̄_h (μ∘(hd α)) (tl α) (E_hab_inv μ α t_l t_r t_m t)
      end
   end
end.
```

Here `Fixpoint` (resp. `struct`) are *Coq* keywords to denote a recursive definition (resp. recursive argument of structural recursive calls). Moreover, in the body of the definition two terms $\mathbf{Incl}_{\mathrm{dec}}$ and E_{hab}_inv are used. Both terms can be proven as lemmas in *Coq*. The first lemma is the following.

```
Lemma Incl_dec : ∀ (μ: M) (d: DIG), Incl(μ, d) ⊕ ¬Incl(μ, d).
```

This term extracts the informative computational content of the predicate **Incl** which is a term of type Prop. This is necessary because in CIC one cannot

obtain elements of type Set by pattern matching on propositions. Thus we have to use $\oplus$: Prop×Prop ⟶ Set — with `left` and `right` its coprojections— to transfer propositions into a boolean sum on which we can pattern match. Hence the above lemma is inevitable, although its proof is quite trivial [7].

The second lemma states the inverse of the last constructor of E_{hab}.

```
Lemma E_hab_inv: ∀(μ: 𝕄)(α: DIG^ω),
      ¬Incl(μ, L) → ¬Incl(μ, R) → ¬Incl(μ, M) → E_h μ α →
                                    E_h (μ∘(hd α)) (tl α).
```

This lemma can be easily proven because E_{hab} is an inductive type and hence all its canonical objects should be generated by one of its constructors.

Note that in $\overline{m}_h$ the output is independent of the proof t. The term t only serves as a catalyst that allows for using recursion where all the other arguments are not inductive. Thus we should be able to prove a *proof irrelevance* result for $\overline{m}_h$.

```
Lemma m̄_h_PI: ∀(μ: 𝕄)(α: DIG^ω)(t₁ t₂: E_h μ α),
                                    m̄_h μ α t₁ = m̄_h μ α t₂.
```

The proof of the above lemma is based on a dependent induction scheme for E_h that is more specialised than the usual induction scheme obtained from initiality: the ordinary induction scheme can be used to prove a property R: $\mathbb{M}\to\mathbf{DIG}^\omega\to$Prop while the dependent induction scheme can be used to prove a property R: $\Pi(\mu\colon \mathbb{M})(\alpha\colon \mathbf{DIG}^\omega).E_h(\mu,\alpha)\to$Prop.

The Lemma $\overline{m}_h$_PI enables us to prove the *fixed point equations* of the $\overline{m}_h$ function. The fixed point equations are in fact unfolding of the body of the definition of $\overline{m}_h$; they will be used later on in proving a similar result for the homographic algorithm. Hence we mention them here:

```
Lemma m̄_hL: ∀(μ: 𝕄)(α: DIG^ω)(t: E_h μ α),
    Incl(μ, L) → m̄_h μ α t = ⟨L, ⟨L^-1 ∘ μ, α⟩⟩.
Lemma m̄_hR: ∀(μ: 𝕄)(α: DIG^ω)(t: E_h μ α),
    ¬Incl(μ, L) → Incl(μ, R) → m̄_h μ α t = ⟨R, ⟨R^-1 ∘ μ, α⟩⟩.
Lemma m̄_hM: ∀(μ: 𝕄)(α: DIG^ω)(t: E_h μ α),
    ¬Incl(μ, L) → ¬Incl(μ, R) → Incl(μ, M) →
                                    m̄_h μ α t = ⟨M, ⟨M^-1 ∘ μ, α⟩⟩.
Lemma m̄_hab: ∀(μ: 𝕄)(α: DIG^ω)(t: E_h μ α),
    ¬Incl(μ, L) → ¬Incl(μ, R) → ¬Incl(μ, M) →
     ∀(t': E_h μ∘(hd α) (tl α)), m̄_h μ α t = m̄_h μ∘(hd α) (tl α) t'.
```

Note that the last lemma states a more general fact than just the fourth

[7] The proof of this as well as all the following lemmas are implemented in *Coq* and can be found in [25].

branch of the recursive definition of $\overline{m}_h$ because the proof obligation t' is abstracted. Nevertheless its proof is similar to the other three.

Having defined $\overline{m}_h$ we need one more auxiliary predicate before defining P_h. This auxiliary predicate is an inductive predicate that ensures that E_h holds for any finite iteration of $\overline{m}_h$ (here π_{ij} denotes the i-th projection of a j-tuple).

```
Inductive Ψh : ℕ → 𝕄 → DIGω → Prop :=
| Ψh0 : ∀(μ : 𝕄)(α : DIGω), Eh μ α → Ψh 0 μ α
| ΨhS : ∀(n : ℕ)(μ : 𝕄) (α : DIGω) (t : Eh μ α),
        Ψh n (π23(m̄h μ α t)) (π33(m̄h μ α t)) → Ψh (S n) μ α.
```

We use the above predicate to define P_h, a predicate that captures the productivity of the homographic algorithm. This predicate will be an inductive type with one constructor.

```
Inductive Ph : 𝕄 → DIGω → Prop :=
| Phab : ∀(μ : 𝕄)(α : DIGω),(∀(n : ℕ), Ψh (S n) μ α) → Ph μ α.
```

The sole constructor of this type ensures that after each emission, which occurs because of E_h, the new Möbius map passed to the homographic algorithm results in a new emission. This fact is implicit in the following two properties of P_h that are needed in the definition of the homographic algorithm. First lemma states the relation between P_h and E_h:

```
Lemma Ph_Eh : ∀(μ : 𝕄)(α : DIGω), Ph μ α → Eh μ α.
```

The second lemma relates $\overline{m}_h$ and P_h, and shows that P_h is indeed passed to the future arguments.

```
Lemma m̄h_Ph : ∀(μ : 𝕄)(α : DIGω)(t : Eh μ α),
  let μ' :=π23(m̄h μ α t) in
    let α' :=π33(m̄h μ α t) in Ph μ α → Ph μ' α'.
```

The proof of both of the above lemmas is based on the inverse of the constructors of Ψ_h, namely the following two lemmas.

```
Lemma Ψh0_inv : ∀(n : ℕ)(μ : 𝕄) (α : DIGω), Ψh n μ α → Eh n μ α.
Lemma ΨhS_inv : ∀(n : ℕ)(μ : 𝕄) (α : DIGω) (t : Eh μ α),
        Ψh (S n) μ α → Ψh n (π23(m̄h μ α t)) (π33(m̄h μ α t)).
```

Lemmas Ψ_{h0}_inv and Ψ_{hS}_inv are in turn consequence of the proof irrelevance of $\overline{m}_h$ that we stated above as $\overline{m}_h$_PI.

Finally we are ready to define the homographic algorithm as the function $\bar{h}\colon \Pi(\mu\colon \mathbb{M})(\alpha\colon \mathbf{DIG}^\omega).P_h(\mu,\alpha) \longrightarrow \mathbf{DIG}^\omega$ that accommodates the proof of

its own productivity as one of its arguments. Here `CoFixpoint` denotes that we are using the cofix rule (see Section 2).

```
CoFixpoint h̄ (μ: 𝕄) (α: DIG^ω) (p: P_h μ α) : DIG^ω :=
  Cons π13(m̄_h μ α (P_h_E_h μ α p))
       (h̄ π23(m̄_h μ α (P_h_E_h μ α p))
          π33(m̄_h μ α (P_h_E_h μ α p))
          (m̄_h_P_h μ α (P_h_E_h μ α p) p)).
```

This definition passes the guardedness condition of *Coq*. Thus we have tackled the problem of productivity by changing the domain of the function and adding a proof obligation.

Next we show that the $\bar{h}$ satisfies the specification of the homographic algorithm. At this point we need to use an extensional equality on final coalgebra of streams, which is also a bisimulation equality. Here we state it for the special case of $\mathbf{DIG}^\omega$ and denote it by $\cong$.

```
CoInductive ≅ : DIG^ω → DIG^ω → Prop :=
|  ≅_c : ∀(α1 α2: DIG^ω), hd α1 = hd α2 → tl α1 ≅ tl α2 → α1 ≅ α2.
```

Note that the sole constructor of $\cong$ has the shape of a bisimulation relation. The proof that this is an equivalence relation can be found in the standard library of *Coq* [9]. Moreover, Giménez shows that this is a bisimulation equivalence relation and derives the usual principle of coinduction [18, § 4.2].

We use $\cong$ to prove an extensional proof irrelevance for $\bar{h}$. Proof of this lemma uses the proof irrelevance $\overline{m}_h$_PI for the modulus function.

```
Lemma h̄_EPI: ∀(μ μ': 𝕄)(α α': DIG^ω)(p: P_h μ α)(p': P_h μ' α'),
                                  μ=μ' → α=α' → h̄ μ α p ≅ h̄ μ' α' p'.
```

Subsequently, we use the above lemma together with the fixed point equations of $\overline{m}_h$ to prove that $\bar{h}$ satisfies the specification of the homographic algorithm. We call these the *cofixed point equations* of the homographic algorithm because they can be considered as the dual of the fixed point equations for recursive functions.

```
Lemma h̄_L: ∀(μ: 𝕄)(α: DIG^ω)(p: P_h μ α),
    Incl(μ, L) → h̄ μ α p ≅ Cons L (h̄ (L^-1 ∘ μ) α).
Lemma h̄_R: ∀(μ: 𝕄)(α: DIG^ω)(p: P_h μ α),
    ¬Incl(μ, L) → Incl(μ, R) → h̄ μ α p ≅ Cons R (h̄ (R^-1 ∘ μ) α).
Lemma h̄_M: ∀(μ: 𝕄)(α: DIG^ω)(p: P_h μ α),
    ¬Incl(μ, L) → ¬Incl(μ, R) → Incl(μ, M) →
                                  h̄ μ α p ≅ Cons M (h̄ (M^-1 ∘ μ) α).
Lemma h̄_ab: ∀(μ: 𝕄)(α: DIG^ω)(p: P_h μ α),
```

```
¬Incl(μ, L) → ¬Incl(μ, R) → ¬Incl(μ, M) →
    ∀(p' : P_h μ∘(hd α) (tl α)), h̄ μ α p ≅ h̄ μ∘(hd α) (tl α) p'.
```

Hence we have shown that our function $\bar{h}$ satisfies the specification of Section 3 and is indeed a formalisation of the homographic algorithm.

Of course we have only tackled the formalisation of the homographic algorithm as a productive coalgebra map, and *not* its correctness. In fact the above algorithm (that does not put any condition on μ) is not productive for non-refining or singular Möbius maps. It is important to have in mind that we have separated the issue of productivity and correctness. This is in accordance with separation of *termination* and correctness in the method of Bove–Capretta for general recursion [5]. In order to prove the correctness we need to define a suitable semantics (for example use another model of real numbers) and prove that the effect of the above algorithm is equivalent to the effect of Möbius maps in the field of real numbers. This is an ongoing project of the author and will be presented in a future work.

4.2 Quadratic Algorithm

In the case of quadratic algorithm we follow the same method that we used for homographic algorithm. We start by defining the initial algebra for the domain of the modulus function.

```
Inductive E_q : 𝕋 → DIG^ω → DIG^ω → Prop :=
| E_qL : ∀(ξ : 𝕋)(α β : DIG^ω), Incl(ξ, L) → E_q ξ α β
| E_qR : ∀(ξ : 𝕋)(α β : DIG^ω), Incl(ξ, R) → E_q ξ α β
| E_qM : ∀(ξ : 𝕋)(α β : DIG^ω), Incl(ξ, M) → E_q ξ α β
| E_qab : ∀(ξ : 𝕋)(α β : DIG^ω), ¬Incl(ξ, L) → ¬Incl(ξ, R) → ¬Incl(ξ, M) →
        E_q (ξ •1(hd α)) •2(hd β) (tl α) (tl β) → E_q ξ α β.
```

Using this we define the modulus function by structural recursion on a term of the above type. Note that in this case the modulus function $\overline{m}_q$ returns a quadruple $\langle d, \langle \xi', \langle \alpha', \beta' \rangle \rangle \rangle$ consisting of the emitted digit, the new quadratic map passed to the continuation of quadratic algorithm and the remainder (unabsorbed part) of two the streams of digits.

```
Fixpoint m̄_q (ξ : 𝕋) (α β : DIG^ω) (t : E_q ξ α β) {struct t}
        : DIG*(𝕋*(DIG^ω*DIG^ω)) :=
match Incl_dec(ξ, L) with
| left _ ⇒ ⟨L, ⟨L^-1 ∘ ξ, ⟨α, β⟩⟩⟩
| right t_l ⇒
   match Incl_dec(ξ, R) with
   | left _ ⇒ ⟨L, ⟨R^-1 ∘ ξ, ⟨α, β⟩⟩⟩
   | right t_r ⇒
      match Incl_dec(ξ, M) with
```

```
      | left _ ⇒ ⟨L, ⟨M⁻¹ ∘ ξ, ⟨α, β⟩⟩⟩
      | right t_m ⇒ m̄_q (ξ •1(hd α)) •2(hd β) (tl α) (tl β)
                        (E_qab_inv ξ α β t_l t_r t_m t)
      end
  end
end.
```

Here E_{qab}_inv is the inverse of the last constructor of the inductive type E_q(cf. E_{hab}_inv for homographic algorithm). Furthermore we have to prove the proof irrelevance and the fixed point equations for $\overline{m}_q$. For brevity we do not mention them here but their statement and proofs can be found in [25].

Next we define the inductive predicate Ψ_q that ensures the validity of E_q for finite iterations of $\overline{m}_q$:

```
Inductive Ψ_q : ℕ → 𝕋 → DIGω → DIGω → Prop :=
|Ψ_q0: ∀(ξ: 𝕋) (α β: DIGω), E_q ξ α β → Ψ_q 0 ξ α β
|Ψ_qS: ∀(n: ℕ)(ξ: 𝕋) (α β: DIGω) (t: E_q ξ α β),
   Ψ_q n (π24(m̄_q ξ α β t)) (π34(m̄_q ξ α β t)) (π44(m̄_q ξ α β t))→
                                                Ψ_q (S n) ξ α β.
```

This allows us to define P_q as an inductive predicate:

```
Inductive P_q : 𝕋 → DIGω → DIGω → Prop :=
|P_qab: ∀(ξ: 𝕋) (α β: DIGω), (∀ (n: ℕ), Ψ_q n ξ α β) → P_q ξ α β.
```

Once again we need to prove two lemmas relating P_q with E_q and $\overline{m}_q$.

```
Lemma P_q_E_q: ∀ (ξ: 𝕋) (α β: DIGω),P_q ξ α β → E_q ξ α β.
Lemma m̄_q_P_q: ∀ (ξ: 𝕋) (α β: DIGω) (t: E_q ξ α β),
   let ξ':=π24(m̄_q ξ α β t) in let α':=π34(m̄_q ξ α β t) in
     let β':=π44(m̄_q ξ α β t) in P_q ξ α β → P_q ξ' α' β'.
```

Finally we can define the quadratic algorithm as a function into the final coalgebra of streams $\bar{q}$: $\Pi(\xi\colon \mathbb{T})(\alpha\ \beta\colon \mathbf{DIG}^\omega).P_q(\xi,\alpha,\beta) \longrightarrow \mathbf{DIG}^\omega$ using the cofixed point operator of *Coq*:

```
CoFixpoint q̄ (ξ: 𝕋) (α β: DIGω) (p: P_q ξ α β) : DIGω :=
   Cons π14(m̄_q ξ α β (P_q_E_q ξ α β p))
        (q̄ π24(m̄_q ξ α β (P_q_E_q ξ α β p))
           π34(m̄_q ξ α β (P_q_E_q ξ α β p))
           π44(m̄_q ξ α β (P_q_E_q ξ α β p))
           (m̄_q_P_q ξ α β (P_q_E_q ξ α β p) p)).
```

To prove that $\bar{q}$ satisfies the specification of the quadratic algorithm we first need the extensional proof irrelevance:

$$\texttt{Lemma}\ \bar{q}\_\mathrm{EPI}\colon \forall(\xi\xi'\colon \mathbb{T})(\alpha\alpha'\beta\beta'\colon \mathtt{DIG}^\omega)(p\colon P_q\ \xi\ \alpha\ \beta)(p'\colon P_q\ \xi'\ \alpha'\ \beta'),\\ \xi{=}\xi' \to\ \alpha{=}\alpha' \to\ \beta{=}\beta' \to\ \bar{q}\ \xi\ \alpha\ \beta\ p \cong \bar{q}\ \xi'\ \alpha'\ \beta'\ p'.$$

Applying this and the fixed point equations of $\overline{m}_q$ we can prove the cofixed point equations of $\bar{q}$.

$$\begin{array}{l}
\texttt{Lemma}\ \bar{q}_L\colon \forall(\xi\colon \mathbb{M})(\alpha\ \beta\colon \mathtt{DIG}^\omega)(p\colon P_q\ \xi\ \alpha\ \beta),\\
\quad \mathbf{Incl}(\xi,\mathtt{L}) \to\ \bar{q}\ \xi\ \alpha\ \beta\ p \cong \bar{q}\ \mathtt{Cons\ L}\ (\bar{q}\ (\mathtt{L}^{-1}\circ\xi)\ \alpha\ \beta).\\
\texttt{Lemma}\ \bar{q}_R\colon \forall(\xi\colon \mathbb{M})(\alpha\ \beta\colon \mathtt{DIG}^\omega)(p\colon P_q\ \xi\ \alpha\ \beta),\\
\quad \neg\mathbf{Incl}(\xi,\mathtt{L}) \to \mathbf{Incl}(\xi,\mathtt{R}) \to\ \bar{q}\ \xi\ \alpha\ \beta\ p \cong \bar{q}\ \mathtt{Cons\ M}\ (\bar{q}\ (\mathtt{M}^{-1}\circ\xi)\ \alpha\ \beta).\\
\texttt{Lemma}\ \bar{q}_M\colon \forall(\xi\colon \mathbb{M})(\alpha\ \beta\colon \mathtt{DIG}^\omega)(p\colon P_q\ \xi\ \alpha\ \beta),\\
\quad \neg\mathbf{Incl}(\xi,\mathtt{L}) \to \neg\mathbf{Incl}(\xi,\mathtt{R}) \to \mathbf{Incl}(\xi,\mathtt{M}) \to\\
\qquad\qquad \bar{q}\ \xi\ \alpha\ \beta\ p \cong \bar{q}\ \mathtt{Cons\ M}\ (\bar{q}\ (\mathtt{M}^{-1}\circ\xi)\ \alpha\ \beta).\\
\texttt{Lemma}\ \bar{q}_{ab}\colon \forall(\xi\colon \mathbb{M})(\alpha\ \beta\colon \mathtt{DIG}^\omega)(p\colon P_q\ \xi\ \alpha\ \beta),\\
\quad \neg\mathbf{Incl}(\xi,\mathtt{L}) \to \neg\mathbf{Incl}(\xi,\mathtt{R}) \to \neg\mathbf{Incl}(\xi,\mathtt{M}) \to\\
\quad \forall(p'\colon P_q\ (\xi \bullet_1(\mathtt{hd}\ \alpha))\bullet_2(\mathtt{hd}\ \beta)\ (\mathtt{tl}\ \alpha)\ (\mathtt{tl}\ \beta)),\\
\qquad\qquad \bar{q}\ \xi\ \alpha\ \beta\ p \cong \bar{q}\ (\xi \bullet_1(\mathtt{hd}\ \alpha))\bullet_2(\mathtt{hd}\ \beta)\ (\mathtt{tl}\ \alpha)\ (\mathtt{tl}\ \beta).
\end{array}$$

Hence $\bar{q}$ agrees with the specification of the quadratic algorithm. As we mentioned quadratic algorithm can be used to evaluate field operations on streams representing real numbers. Thus we have shown how to equip the final coalgebra of stream with a field structure.

4.3 Analysis of the Method

Evidently the method for formalising the quadratic algorithm mimics precisely the one used for homographic algorithm. This suggests that one can generalise this method to obtain a scheme in style of [6] for formalising specification of partial functions on final coalgebras. Such a method would be the dual of the Bove–Capretta [5] for general recursion method and the term *general corecursion* seems suitable. In this article we have not developed such a scheme, as our focus lies on the special case of exact arithmetic algorithms for the coalgebra of reals. Nevertheless, all the intermediate inductive predicates and recursive functions can be obtained by following the shape of the specification. Therefore we consider the method to be generic enough for formalising arbitrary partial coalgebra maps for strictly positive functors in any category modelling CIC.

In fact the method might work in categories for simpler extensions of Martin-Löf type theory. This is because the method does not rely on properties peculiar to CIC; even the distinction between Set and Prop is not necessary and we could put all the inductive predicates in Set. However, with an eye on program extraction, we prefer to keep the distinction between informative and non-informative objects. Note that if we extract the function $\bar{h}$ the argument $P_h(\mu,\alpha)$ will be discarded, resulting in a function $\hat{h}\colon \mathbb{M}\times\mathbf{DIG}^\omega \longrightarrow \mathbf{DIG}^\omega$

which is only different from the original specification modulo unfolding (see the discussion by Bertot [4]).

It remains to be seen whether the method can be applied in categories other than those modelling some extensions of Martin-Löf type theory.

Finally we compare our method with the one given by Bertot [4]. Both in our work and in [4] the same idea of dualising Bove–Capretta's method is pursued. One difference between our work and [4] is that we consider P_h to be an inductive type while Bertot uses a coinductive predicate F_infinite. But the two predicates seem to be extensionally equal. Moreover the use of an extra inductive predicate Ψ_h to capture the iteration of $\bar{m}_h$ does not occur in Bertot's method for filter. This is due to the slight difference between homographic algorithm and the general form of filter function: in homographic algorithm the property **Incl** is a dynamic property because the Möbius map, being passed to future steps of the function, is changing all the time; therefore the property that states the productivity should keep track of this. Another notable difference is our use of bisimulation equality and proof of extensional cofixed point equations which is not present in [4] where instead another coinductive predicate is used to describe the *connectedness* of a stream with respect to a given property.

5 Conclusions and Further work: Elementary Functions and need for Coinduction–Recursion

We have shown how to equip the coalgebra of real numbers with a field structure and in the process have presented a method that can be applied to other partial infinite objects. From the field of real numbers the next step would be to define elementary functions on this final coalgebra. Homographic and quadratic algorithm are the base case of Edalat and Potts' *normalisation algorithm* on coalgebra of expression trees [23, § 5]. Therefore if we could apply the method of previous section to formalise this algorithm we could obtain all the elementary functions. Unfortunately this does not seem to be possible: the method of the previous section needs a more complicated machinery than that of CIC to be applicable to normalisation algorithm. This is because the normalisation algorithm is a nested algorithm and therefore applying our method the modulus of productivity $\overline{m}_h$ will be a nested function too. It is well-known that applying Bove–Capretta method for formalising nested recursive functions requires the presence of *inductive–recursive* types [5,13]. In this case the inductive domain predicate will become an inductive–recursive predicate that is defined simultaneously with the nested function. A similar phenomenon happens in our method, in the sense that we need a notion similar to induction–recursion that would allow for simultaneous definition of an inductive predicate together with a cofixed point. The author is exploring the possibility of defining such a notion.

Acknowledgement

The author wishes to thank the anonymous referees for their comments.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoret. Comput. Sci.*, 342:3–27, Sept. 2005.

[2] F. Bartels. Generalised coinduction. In A. Corradini, M. Lenisa, and U. Montanari, editors, *Proc. of 4th Workshop on Coalgebraic Methods in Computer Science, CMCS'01, Genova, Italy, 6–7 Apr. 2001*, volume 44(1) of *Electron. Notes Theor. Comput. Sci.* Elsevier Science Publishers, Amsterdam, 2001.

[3] Y. Bertot. CoInduction in Coq. In *Lecture Notes of TYPES Summer School 2005, August 1526 2005, Göteborg, Sweden. vol II*, 2005. http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/lectnotes_vol2.pdf, [cited 19 May 2006].

[4] Y. Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Comput. Sci.*, pages 102–115. Springer-Verlag, 2005.

[5] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Comput. Sci.*, pages 121–135. Springer-Verlag, 2001.

[6] D. Cancila, F. Honsell, and M. Lenisa. Generalized coiteration schemata. In P. Gumm, editor, *Proc. of 6th Workshop on Coalgebraic Methods in Computer Science, CMCS'03, Warsaw, 5-6 Apr. 2003*, volume 82(1) of *Electron. Notes Theor. Comput. Sci.* Elsevier Science Publishers, Amsterdam, 2003.

[7] V. Capretta. *Abstraction and Computation.* PhD thesis, Katholieke Universiteit Nijmegen, Apr. 2002.

[8] A. Ciaffaglione and P. Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoret. Comput. Sci.*, 351(1):39–51, Feb. 2006.

[9] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.0.* INRIA, Apr. 2004. http://coq.inria.fr/doc/main.html, [cited 19 May 2006].

[10] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Comput. Sci.*, pages 62–78. Springer-Verlag, 1994.

[11] E. W. Dijkstra. On the productivity of recursive definitions. Personal note EWD 749, http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF, [cited 19 May 2006], Sept. 1980.

[12] C. Dubois and V. V. Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. In M. Kerber, editor, *Proc. Workshop on Mechanization of Partial Functions, July 5 1998, Lindau, Germany*, pages 53–62, 1998. available at ftp://ftp.cs.bham.ac.uk/pub/authors/M.Kerber/98-CADE-WS/dubois-donzeau.ps.gz, [cited 19 May 2006].

[13] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symbolic Logic*, 65(2):525–549, 2000.

[14] A. Edalat and P. J. Potts. A new representation for exact real numbers. In S. Brookes and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference (MFPS XIII), Carnegie Mellon University, Pittsburgh, PA, USA, March 23–26, 1997*, volume 6 of *Electron. Notes Theor. Comput. Sci.* Elsevier Science Publishers, 1997.

[15] M. H. Escardó and A. K. Simpson. A universal characterization of the closed Euclidean interval. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 115–128. IEEE Computer Society, 2001.

[16] H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proc. of Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 193–217. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992. available at http://citeseer.ist.psu.edu/geuvers92inductive.html, [cited 19 May 2006].

[17] E. Giménez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6–10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Comput. Sci.*, pages 39–59. Springer-Verlag, 1995.

[18] E. Giménez. *Un Calcul de Constructions Infinies et son Application a la Verification des Systemes Communicants*. PhD thesis PhD 96-11, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Dec. 1996.

[19] R. W. Gosper. HAKMEM, Item 101 B. http://www.inwap.com/pdp10/hbaker/hakmem/cf.html#item101b, [cited 19 May 2006], Feb. 29 1972. MIT AI Laboratory Memo No. 239.

[20] P. T. Johnstone. *Sketches of an Elephant. A Topos Theory Compendium, vol 2*. Number 44 in Oxford Logic Guides. Oxford University Press, 2002.

[21] M. Lenisa. From set-theoretic coinduction to algebraic coinduction: Some results, some problems. In B. Jacobs and J. Rutten, editors, *Proc. of 2nd Int. Wksh. on Coalgebraic Methods in Computer Science, CMCS'99, Amsterdam, The Netherlands, 20–21 March 1999*, volume 19 of *Electron. Notes Theor. Comput. Sci.* Elsevier Science Publishers, North-Holland, 1999.

[22] P. Martin-Löf. Mathematics of infinity. In P. Martin-Löf and G. Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Comput. Sci.*, pages 149–197. Springer-Verlag, 1990.

[23] M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud Universiteit Nijmegen, Sept. 2004.

[24] M. Niqui. Formalising exact arithmetic in type theory. In S. B. Cooper, B. Löwe, and L. Torenvliet, editors, *New Computational Paradigms: First Conference on Computability in Europe, CiE 2005, Amsterdam, The Netherlands, June 8–12, 2005. Proceedings*, volume 3526 of *Lecture Notes in Comput. Sci.*, pages 368–377. Springer-Verlag, 2005.

[25] M. Niqui. http://www.cs.ru.nl/~milad/ETrees/coinductive-field/, [cited 19 May 2006], Jan. 2006. Files under Coq 8.0pl3.

[26] D. Pattinson. Computable functions on final coalgebras. In *Proc. of 6th Workshop on Coalgebraic Methods in Computer Science, CMCS'03, Warsaw, 5-6 Apr. 2003*, volume 82, 1 of *Electron. Notes Theor. Comput. Sci.* Elsevier Science Publishers, 2003.

[27] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. Research report RR 92-49, Laboratoire de l'Informatique du Parallélisme, Ecole normale supérieure de Lyon, Dec. 1992.

[28] D. Pavlović and M. H. Escardó. Calculus in coinductive form. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, pages 408–417, 1998.

[29] D. Pavlović and V. Pratt. On coalgebra of real numbers. In B. Jacobs and J. Rutten, editors, *Coalgebraic Methods in Computer Science, CMCS'99*, volume 19 of *Electron. Notes Theor. Comput. Sci.*, page 15 pages. Elsevier Science Publishers, 2000.

[30] P. J. Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, University of London, Imperial College, July 1998.

[31] T. Uustalu and V. Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10(1):5–26, 1999.